INTEGRATED DEVICE TECHNOLOGY

1995

RISC Microprocessor Application Guide

RISC APPLICATIONS GUIDE

AUGUST 1995

Integrated Device Technology

Integrated Device Technology, Inc.

INTRODUCTION

This manual is a collection of various applications notes and conference papers written to describe the behavior and use of the 32-bit family of RISController™ devices and 64-bit ORION™ family of devices.

The application notes include descriptions of design techniques, development environments, and software development tools. The reader is encouraged to review the introduction of the various application notes as a brief summary of the topic of that paper.

This manual is complemented by other documentation, also available from your IDT sales representative. These documents include:

- The RISC data book, which contains data sheets for these devices. Also included are the electrical specifications, pinout, current speed grades, and package dimensions.
- The R3041 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R3041.
- The R36100 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R36100.
- The R3051 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R3051 and R3052.
- The R3071/R3081 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R3071 and R3081.
- The R4650 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R4650.
- The R4600/R4700 Hardware User's Manual, which contains a detailed description of the hardware and software interface of the R4600 and R4700.
- The various user's manuals on the IDT software tools, and the user's manual for the IDT79S341, IDT79S385A, IDT79S381, IDT79S460, and IDT79S464 Evaluation Boards.
- The IDT Advantage Catalog, detailing various third-party tools, such as real-time OS, in-circuit emulation, logic analyzer support, and program development tools available to support applications development around the IDT RISController™ and ORION™ family devices.

1995 RISC APPLICATIONS GUIDE
TABLE OF CONTENTS

Integrated Device Technology, Inc.

IDT79R3051™ SYSTEM DESIGN EXAMPLE

APPLICATION NOTE AN-86

by Andrew Ng

INTRODUCTION

This application note describes a memory evaluation board that is an example of many of the design considerations for systems based on an IDT79R3051™ RISController™ family CPU.

The memory board, illustrated in Figure 1, consists of:

- An R3051 CPU
- Reset circuitry
- An address de-multiplexer
- A data transceiver
- Wait-state and memory control logic
- 128K bytes of SRAM
- 128K bytes of EPROM
- A dual channel UART
- A real time counter
- An interrupt controller

In addition, an expansion connector supplies all the CPU signals for the addition of external modules such as DRAM memory systems or other application specific I/O systems. The memory and I/O system on the example board are compatible with the IDT7RS382 R3000 Evaluation Board. Thus 7RS382 software such as the IDT/sim PROM Debug Monitor can run on the example board. The board is typical of an embedded controller core such as for LAN adapters, laser printers, facsimiles, and avionics applications. The differences would appear in which peripherals are used and memory type, size, and speed requirements.

The board was designed as a generic example of the construction of a system using the IDT79R3051 RISController with both low parts count and cost sensitive requirements. However, since many generalities were taken into consideration, many systems can reduce both parts count and cost even further. Although the board is not populated with parts that have the highest performance achievable, its design can be easily modified to do so. In addition, PAL™ support for further experiments with optimizations and trade-offs can be done to accommodate different kinds and speeds of memory and I/O. While the board is designed with SRAM for the simplicity of a design example, the extension to a DRAM system with CAS before RAS refresh is only slightly more complex.

THE R3051 RISCONTROLLER CPU

The IDT79R3051 family is a series of high-performance 32-bit microprocessor RISControllers designed to bring the high-performance inherent in the MIPS™ RISC architecture into low cost, simplified, and power sensitive applications.

The instruction set is compatible with the 79R3000A and 79R3001 RISC CPUs. Features of the R3051 family include:

- 4kB (R3051) to 8kB (R3052) of Instruction Cache on-chip
- 2kB of Data Cache on-chip
- Clocked from a single, double-frequency clock input
- On-chip 4 deep read and write buffer
- On-chip DMA arbiter
- Flexible burst/simple block bus interface
- Multiplexed address and data bus for low cost packaging, simplicity of use
- Base versions use fixed address translation to simplify software
- Extended architecture versions use 64-entry, fully associative Translation Lookaside Buffer (TLB) to support page mapping and virtual memory

The R3051 RISController combines a similarly featured R3000A CPU system consisting of over 50 LSI/MSI parts into a single integrated chip.

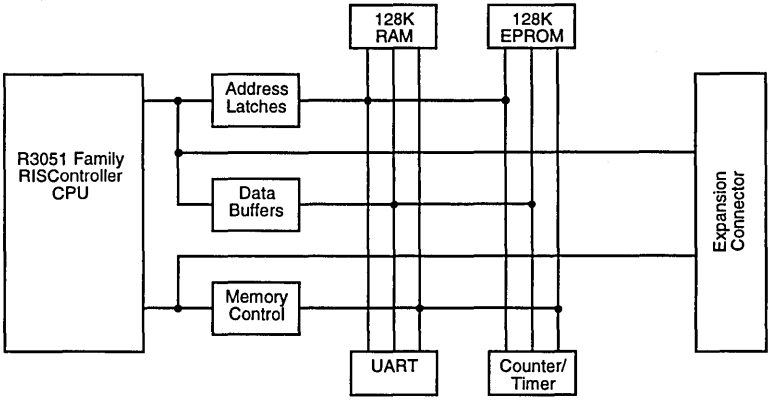


Figure 1. System Block Diagram

DETAILED DESIGN REVIEW

The following sections give a detailed review of how each functional block relates specifically to designing with the R3051 RISController. Particular attention is focused on alternative design strategies that could reduce parts count and improve performance as well as on a description of the original design. The subsystem block designs include:

- Analog reset logic
- A PAL-based memory controller (3x PALs)
- Address de-multiplexer (4x IDT74FCT373T)
- Data transceiver (4x IDT74FCT623T)
- 128kB of SRAM (4x IDT71256 32kx8 45ns SRAM)
- 128kB of EPROM (4x 27256 32kx8 125ns EPROM)
- 68681 DUART
- 8254 Timer
- Interrupt controller (1x PAL)
- Off-card connector

Reset, Reset Vector, and Clock Buffer Circuitry

The $\overline{\text{Reset}}$ signal is based on a linear integrated circuit, a TI TL7705A supply voltage supervisor with a Power-On Reset Generator. A 1 µF capacitor is used to program the reset generator for a 13 ms Reset period.

Note that because the R3051 synchronizes the $\overline{\text{Reset}}$ input signal internally, an RC circuit can be used instead. An example is to pull $\overline{\text{Reset}}$ high with a resistor of about 10K Ohms, tie $\overline{\text{Reset}}$ to a 22 µF capacitor which is tied to ground, and tie $\overline{\text{Reset}}$ to a push button switch that is tied to ground. The example board can be reprogrammed and populated to experiment with $\overline{\text{Reset}}$.

Certain configuration options (the reset vector) are selected in the R3051 by using the interrupt pins at the rising edge of $\overline{\text{Reset}}$. On the example board, the interrupt pins are simply pulled up (or down) since $\overline{\text{SInt}}$(2:0) are not used in this system (software can permanently mask these interrupt inputs in the Status Register). However, if they are used (via the expansion connector) they would need to be multiplexed with the reset function. There are a number of techniques to perform this multiplexing: for example, if the interrupting agent is not capable of tri-stating its interrupt during $\overline{\text{Reset}}$, an external multiplexer such as an IDT74FCT257T can be used, with the enable always tied active and the select tied to $\overline{\text{Reset}}$. If the interrupting agent tri-states its interrupt during Reset, then using simple pull-ups or pull-downs will still operate properly.

The clocks on the board are buffered by an IDT74FCT240C(T) inverting tri-state buffer. This buffer was selected partially to provide a board testability path for injecting a test clock, as well as to buffer the signals to increase their drive. The primary reason for the buffer, however, is to invert $\overline{\text{SysClk}}$ to form SysClk, the signal that is used to clock the state machines on this board. Buffer output pins closest to the ground pin (pins with the lowest pin inductance) were used first to help lessen potential noise and ground bounce problems. The Clk2xIn oscillator is socketed, so that the board may be populated with different speed parts.

In this design, the FCT240C(T) enables are pulled down to be active all of the time. Since $\overline{\text{SysClk}}$ does not tri-state when $\overline{\text{Tri-State}}$ ($\overline{\text{SInt}}$(1)) is active during the reset vector, it is helpful to an ATE programmer to be able to tri-state the inverter.

Memory Controller

The example board's Memory Controller consists of three 22V10 PALs. The first PAL is used for address decoding, the second for wait state and cycle counting, and the third for byte enables. The PALs are functionally described in the following paragraphs. The PAL equations are included in the appendices. The PALs are all placed in sockets, and thus can easily be reprogrammed for various experiments.

Address Decoder

The Address Decoder PAL, MEMDEC.JED, uses Address(31:17) to generate chip selects. The chip selects are decoded according to the 7RS382 address map as described in the 7RS382 Hardware User's Guide. Three spare I/O pins are provided, which could be used to decode additional chip selects. These spare outputs are in place of the 'USER CS1X*' chip selects provided for on the 7RS382 board, but not explicitly supplied by this example board.

The address decoder does not wait for ALE to begin generating the chip-select outputs. It does this so that maximum performance may be achieved, since the Chip Select outputs will be generated earlier in the transfer. However, as a result, the CS outputs may tend to "glitch" as a valid address is driven. Thus, the Read Enable and Write Enable seen in the memory system must be synchronized so that they are valid only within the time that the CPU is attempting a read or write transfer. This combination allows maximum performance: address and chip enables are seen early in the transfer, but the Read and Write signals are generated synchronously to insure proper system operation.

One of the extra I/O pins can be used as a test enable input to tri-state the outputs for board level ATE. Some systems will not need to decode as many address bits or may have a fixed map, and thus may able to use FCT138's or 16V8's to do the address decoding instead of the relatively expensive 22V10 part.

Memory Cycle Controller

The purpose of the Memory Cycle Controller is to provide a wait-state generator which stalls the R3051's Bus Interface Unit, so that various types and speeds of memory can be used. The Memory Cycle Controller is implemented with a 22V10 PAL called MEMCONT.JED. Note that this PAL was selected in order to make the PAL equations more readable. A lower cost solution may implement the state machine in two 16R8 PALs.

The Memory Cycle Controller allows various speeds of memory devices to be used, by using the throttled read supported by the R3051 bus interface. Other kinds of transactions are treated as simplified cases of the throttled read.

The basic state machine looks for the start of a read or write transaction by looking for an asserting edge of $\overline{\text{Rd}}$ or $\overline{\text{Wr}}$. When a transaction is begun, the state machine starts a 5-bit binary up counter, C(4:0). C(4:0) then increments on each SysClk rising edge. C(4:0) is used as the basic timing master for all

of the other control signals generated in the state machine.

In the memory scheme used here, rather than search for the negating edge of $\overline{Rd}$ or $\overline{Wr}$ at the end of the transaction, a $\overline{CycEnd}$ synchronous decoder is used to tell the C counter when the end of the memory cycle occurs. This type of strategy is used because the de-asserting edges of $\overline{Rd}$ and $\overline{Wr}$ occur within the setup and hold times of a buffered/inverted (FCT240C(T)) $\overline{SysClk}$. Typically, the de-asserting edge of $\overline{Rd}$, $\overline{Wr}$, and $\overline{Burst}$ should not be used to control a SysClk based state machine. Similarly, the rapid negation of ALE by the processor makes it difficult to synchronously sample ALE when using a state machine driven by a buffered clock.

$\overline{CycEnd}$ serves to synchronously reset the state machine when a de-asserting $\overline{Rd}$ or $\overline{Wr}$ edge is expected, whether or not the $\overline{Rd}$ or $\overline{Wr}$ de-asserting edge meets the setup and hold times of the state machine. Another output, $\overline{EnStart}$ is used to start the byte enables by waiting a number of cycles before asserting. The amount of time the transfer waits is used to allow drivers used in the previous transfer to tri-state, and may be necessary in systems which employ devices whose output disable time is long relative to the system clock frequency.

Other outputs from the Memory Cycle Controller PAL include the R3051 transfer termination inputs $\overline{RdCEn}$, $\overline{Ack}$, and $\overline{BusError}$. On a read transfer, $\overline{Burst}$ and one of the Chip Enable inputs from the Address Decoder are used to determine the timing and quantity of $\overline{RdCEn}$ signals to be asserted for this transfer (according to the requested transfer size and the memory device speed).

$\overline{Ack}$ is asserted at the end of a write cycle to indicate completion of the transfer, and optionally towards the end of a Quad Word (Burst) read cycle. A description of the various kinds and options of read and write cycles is thoroughly explained in the R3051 Family Hardware User's Guide. The number of cycles before and between the assertion of $\overline{Ack}$ and $\overline{RdCEn}$ is programmable, allowing flexibility for various types of memories.

Finally, the $\overline{BusError}$ output is used to end an undecoded memory cycle. In the R3051, $\overline{Rd}$ is negated one-half cycle after the $\overline{BusError}$ input is asserted.

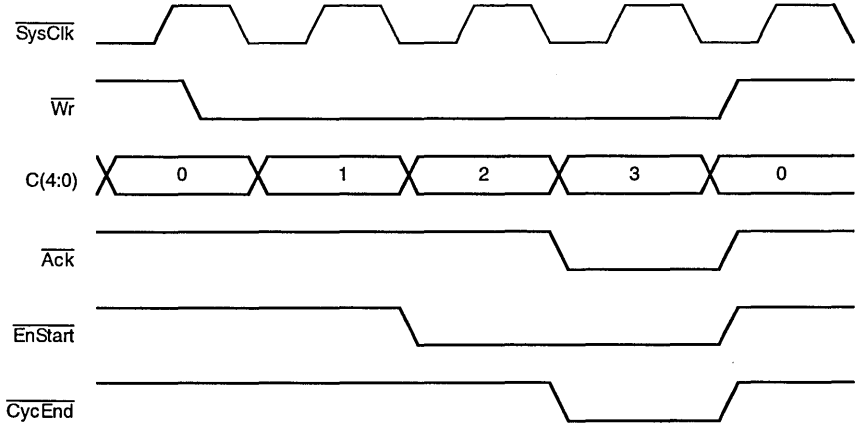


Figure 2. Timing of CycEnd

Other Approaches

Of course, alternative methods and techniques to memory interfacing with an R3051 family CPU exist. Four approaches easily implemented in discrete components include:

- using a SysClk based $\overline{CycEnd}$ counter (as used in this example)
- using asynchronously resettable registers for the counter
- using interlocking SysClk and $\overline{SysClk}$ registers
- using an unbuffered $\overline{SysClk}$

All of these methods can be used to design for the clocking scheme of the R3051 Family, which uses both the rising and falling edges to control its outputs. The use of both edges of the clock allows the R3051 to mitigate the 1 clock inter-transaction latency that is associated with most other CPUs that need the extra clock to fixup and start new memory cycles. However,because the R3051 Family asserts and de-asserts its edges the same way on both $\overline{Rd}$ and $\overline{Wr}$ cycles, specific methods can be employed so that the memory system is always clocked from one edge of $\overline{SysClk}$. An example of this is the $\overline{CycEnd}$ method used on this board, which ignores the edges that are not synchronized with the state machine. Although traditional high-performance CPUs require complex state machines to operate efficiently, the beauty of the R3051 family is the simplicity of its interface. Memory control state machines for the R3051 family are really only minor variations on traditional wait-state machines, and can also easily take advantage of the 1/2 clock inter-transaction savings provided by the CPU interface.

Each of the four approaches has advantages as well as drawbacks relative to each other. The following paragraphs will give a brief description of each technique. Each of the methods could be used by themselves or combined with one

or more of the other methods, to achieve the optimal price/performance/parts count for a given application. Systems employing dedicated interface chips (such as the IDT R372x family, or customer specific ASIC or Gate Array devices), may choose to make different trade-offs than those using discrete component based solutions.

Using SysClk and generating a Cycle End indicator

The SysClk based CycEnd approach as described above is straightforward because of its similarity to traditional wait-state machines. As mentioned above, it does not require the terminating edge of Rd or Wr to complete a transaction.

The system implemented in this design example is limited in speed by:

tclk/2 >= t240 + tpalco + t3051setup + tcap + twire

which works out to 28 MHz for a 10 nsec 16V8, over 40 MHz for a 5 nsec 16R8 PAL, and 33 MHz for a 10 nsec 22V10 PAL.

Using Asynchronous Reset to terminate the Cycle Counter

The second potential method, which uses an asynchronous reset to terminate the cycle, requires AND'ing together Rd and Wr into the the reset line of the counter C(4:0) and can be demonstrated by reprogramming the PAL on the example board. The reset-to-valid output, reset width, and the reset recovery time to clock are among the speed limiting paths in this approach when implemented in PALs. Unfortunately, the reset-to-output delay of a PAL is usually less optimized and relatively slow.

tasyncreset <= tclk/2 - trdn - tcap - twire

For example, a 20 MHz system would require a reset-to-output delay of 17ns, which can be found in a 10 nsec 22V10 PAL (with a 15 nsec reset to valid output data time).

Using interlocking PALs clocked on opposite edges

The third potential approach uses a SysClk based register to detect asserting edges and a SysClk based register to detect de-asserting edges. The outputs of each of the PALs interlock by controlling the outputs of the other PALs. This allows the flexibility of seeing all edges and being able to control outputs optimally by using any 1/2 clock edge (such as output enables). Such an approach obviously requires more PALs, and is somewhat speed limited by:

tclk/2 >= t240 + tpalco + tpalsetup + tcap + twire

which works out to 20 MHz for a 10 nsec 16V8 PAL.

In systems using chips designed specifically to interface to the R3051 family (such as the IDT R3721 DRAM controller), this approach is simpler to implement and leads to the highest levels of performance.

Using an unbuffered SysClk

The fourth potential approach uses an unbuffered SysClk based state machine. This leads to the requirement of having 0 hold time on the registers as well as a 2 nsec minimum propagation delay time to meet the R3051 timing requirements (note that using a buffered SysClk instead of the unbuffered version would require negative hold time on the registers). Despite these restrictions, some PALs can be found that meet all of these requirements. This approach leads to a one cycle latency in reacting to R3051 output assertions. An asserting Rd or Wr would be seen a clock too late to bring RdCEn or Ack low during their first possible sampling clock. Using an unbuffered SysClk has a speed advantage over the other techniques:

tclk >= tpalco + t3051setup + tcap + twire

tclk/2 >= t3051prop + tpalsetup + tcap + twire

which can support designs of 35 MHz for a 10 nsec 16V8 PAL and well over 40 MHz with a 7.5 nsec 16R8 PAL.

An additional consideration relative to using an unbuffered SysClk is the amount of loading placed on the clock, and the impact of additional loading on R3051 AC parameters. Of course, when using a single chip memory controller such as the IDT R3721 or a customer designed ASIC, these loading considerations are minimal.

In summary, the R3051 Family uses both edges of the clock to assert control signals in order to reduce inter-transaction delay between external bus cycles. However, by using one or a combination of the above techniques in a design, a traditional wait-state machine can still be used with the addition of only minor variations.

Read and Write Enables

The Read and Write Enables PAL, MEMEN.JED, uses EnStart and CycEnd to control the initiation and length of the output enable and write enable assertions. Rd and Wr are used to select between read and write cycles. Note that it would have been possible to combine individual bank selects with the address decoder PAL, rather than use a distinct PAL to control the timing of the assertion of Write and Read Byte Strobes.

On read cycles, RdEn is asserted as the system's primary output enable signal. RdDataEn is used to enable the FCT623T data transceiver bank. RdDataEn in most systems would simply be 'DataEn' as supplied straight from the processor. This system provides RdDataEn in case other transceiver banks are added to the system.

The byte enables are used to support partial word writes which are used during byte, halfword, and tri-byte operations. Write cycles combine the byte enables, BE(3:0), with Wr, EnStart, and CycEnd to form the write enable outputs WrEn(D:A) which are attached to the byte banks within the memory system. Whether or not the system is Little or Big Endian, WrEn(A) is always attached to the LSB. WrEn(D:A) can also be implemented using an FCT257T multiplexer. WrDataEn is used to control the FCT623T data transceiver bank and must be held extra long to provide memory data hold time.

Finally, the Byte Enable PAL also has a synchronized PowReset output called Reset and a "guarded" GUARTCS. The guarded chip select, GUARTCS is an example of interfacing R3051 signals to a Motorola-type I/O Device as opposed to an Intel-type I/O Device.

Motorola-type devices multiplex their read/write input pin and expect a data strobe pin to validate the data out or to latch the data in, while Intel-type devices have separate read and write strobes. Since the MC68681 DUART is a Motorola device, the data strobe must start late and end early, so that read/write is held throughout that period. Additionally, the MC68681 uses its chip select pin as a data strobe. As a data strobe, it is important not to have decoder glitches on the chip select since reads in I/O devices are often used to update

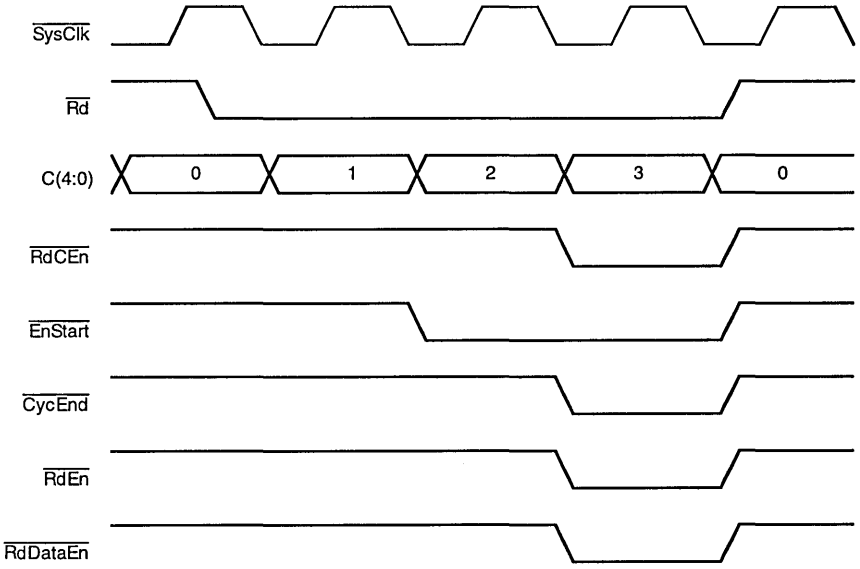


Figure 3. Timing Diagram of RdEn

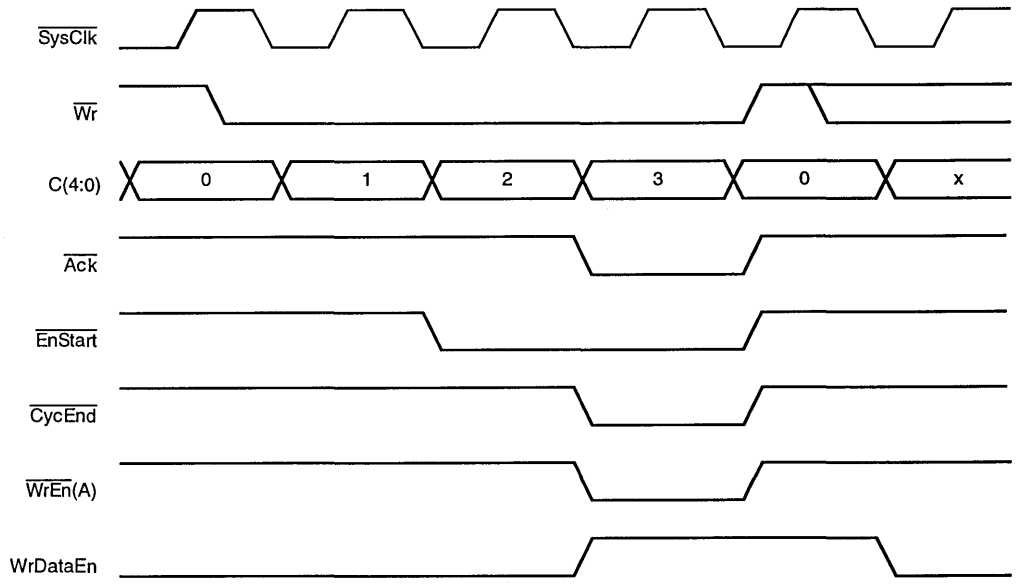


Figure 4. Timing Diagram of WrEn(A)

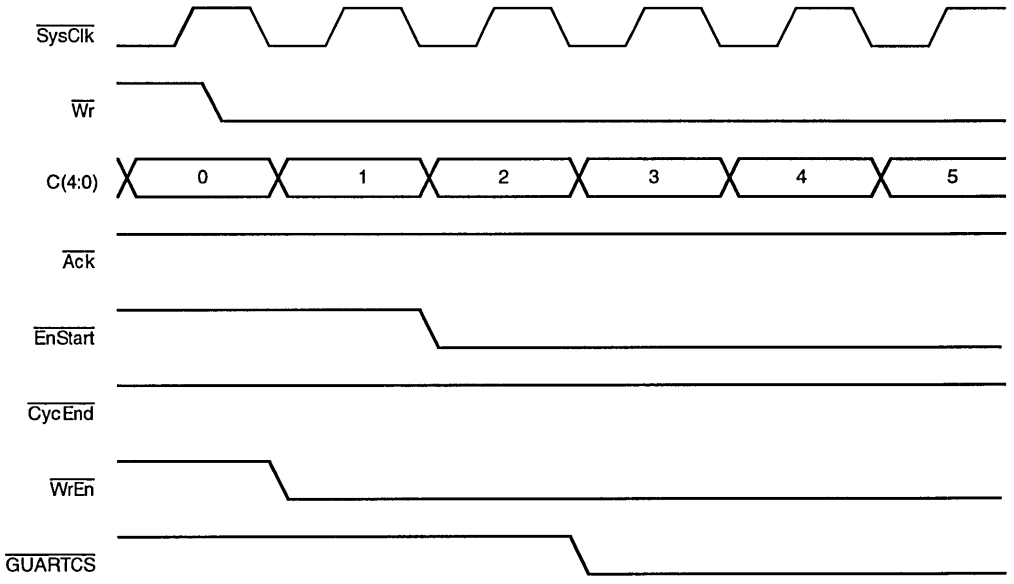


Figure 5. Timing Diagram of Start of GUARTCS

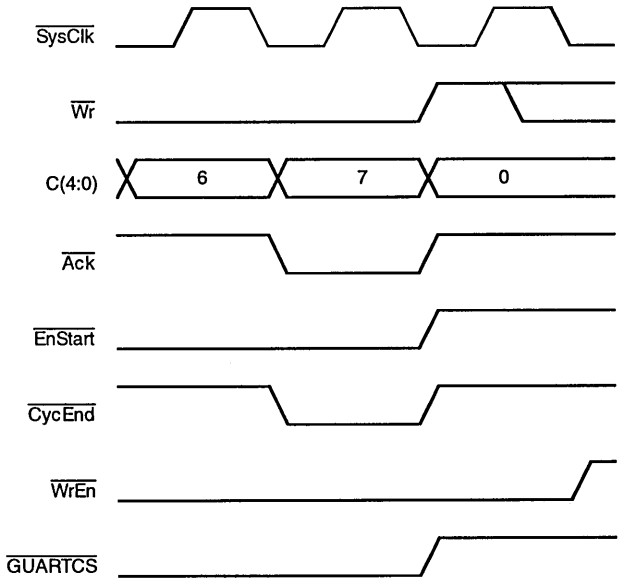


Figure 6. Timing Diagram of End of GUARTCS

FIFO pointers. Thus, the guarded GUARTCS uses EnStart and CycEnd to shorten up UARTCS. Finally, WrEn is provided to extend Wr to allow additional data hold time at the end of the write cycle. WrEn could easily be inserted with another OR term into WrEn(A).

Address Latch and Transceiver De-multiplexer

The address latch bank consists of four FCT373T 8-bit transparent latches. ALE is used for the latch enable on the FCT373T's. The transparent phase allows extra address decoding time during the time that ALE is high; the outputs of the latches are fed directly to the address decode PAL and to the memory devices. In order to insure that address hold time to the latches are met, it is important to take care with the use of the ALE signal. The number and length of the ALE traces is critical and should be kept to a minimum.

Rather than use FCT373's, DRAM systems may want to use FCT821's or FCT823's, which are wider latches. RAS/CAS address multiplexing can be performed by sequencing the output enables of the latches and having the outputs of the latches tied together and driving the DRAM address bus.

The data transceiver bank on the example board uses four FCT623T 8-bit transceivers. FCT623T's were chosen over the similar 10-bit FCT861's and 9-bit FCT863's simply to reduce pin count. The FCT861/3's provide a more conventional interface, since both output enables are active low, instead of one enable active high, and the other active low as in the FCT623T's. However, since this system uses PALs to control the transceivers, the use of FCT623's poses no additional complexity to the design.

FCT623T's were selected instead of FCT245's because of the ease of interfacing to dual output enable pins instead of a direction and enable pins as in the FCT245. Interfacing with FCT245 controls would ideally require that the direction control only be changed when the output enable is disabled. This requires extending a combined (latched) Rd and Wr based signal for an extra cycle at the end of a memory transaction, which may be the beginning of the next memory cycle. Unless the direction pin is controlled with a SysClk based state machine, a signal like EnStart would be necessary to keep the enable pin de-asserted in the subsequent cycle until the direction pin control becomes valid. Some systems with high noise tolerance, e.g., IBM-PC adapter boards, forgo the extra cycle ideal and simply bus contend for a very short time (a few ns) into its memory system by having the read strobe directly control the direction. DataEn, output from the CPU, can be used in such systems to simplify control signal generation.

When there are no pending DMA, read, or write requests, the R3051 tri-states the A/D(31:0) bus during these non-bus clock cycles to reduce power consumption. One can optionally add external pullup or pulldown resistors so that the A/D(31:0) bus is always defined for board level ATE and so that the input pins of the latches and transceivers are stabilized.

Finally, systems that can output disable (oe to Z-state) all memory readable devices within:

tdisable < tclk/2 - t3051dataenn + taddr - tcap - twire

might not require the transceiver bank and thus could reduce the parts count by 4.

EPROM and Static RAM Memory

The memory on the example board is populated with 125 nsec Erasable PROMs (EPROMs) and 45 nsec Static RAMs (SRAMs). Four 27C256 32Kx8 EPROMs are used to form 128K bytes of ROM. The EPROMs are placed in sockets and thus can easily be removed for reprogramming or replacement; alternative designs may wish to add circuitry to allow in-board programming of the EPROMs (e.g. Flash Erase EPROMs).

The EPROMs have a relatively long output disable time (oe to z-state), typical of ROMs and thus require data buffers to prevent contention on the multiplexed AD(31:0) bus, since the following equation is not met:

tclk/2 >= tdisablecontrol + tdisable - taddr + tcap + twire

In addition, the disable time for these EPROMs is long enough that, except for relatively slow systems (under 20 MHz), extra clocks need to be added to the next bus cycle to prevent bus contention with other memory banks. This is determined by:

tclk >= tdisablecontrol + tdisable - tdata + tcap + twire

The SRAM bank is formed using four IDT71256 32Kx8 SRAMs for a total of 128K bytes. The RAM chips have common data I/O pins, separate read and write strobes, and chip selects. RAMs without a separate read strobe (output enable pin) may require more complex address decoding when used in a multiple bank configuration.

DUART, Timer, and Interrupt Controller

An MC68681 DUART and an MAX235 RS232 transceiver are used to form two RS232 serial communication links. The DUART control registers are word addressed, but only D(7:0) are used. The MC68681 is an example of a Motorola-type I/O interface as explained above.

An iP8254 timer/counter chip is used for a real-time clock or timer. The iP8254 is an example of an Intel-type I/O interface. The iP8254's need for separate read and write strobes matches up well with the R3051.

Software control of these chips is best described by their respective data sheets. Typically, most software programs for the 7RS382 have used the DUART in a polling mode and the timer in a square wave mode. Interrupts Int(5:3) are controlled by UARTIntOC, Timer OutB, and Timer OutA respectively from MSB to LSB. The 16R8 PAL, called MEMINT.JED, is used to control these interrupts latches in the assertion transition of the original interrupt lines.

The controller holds the interrupt line to the processor for Timer A and Timer B until they are acknowledged (as required by the R3051). Acknowledgement is indicated by reading the interrupt controller at Virtual Address BF800010 and BF800014 (Physical Address 1F800010 and 1F800014) respectively. This action incidentally reads extraneous data from the Timer chip itself on D(7:0). The DUART interrupt must be acknowledged by using the DUART control registers.

The output disable to data in z-state time for these I/O peripherals is relatively long, as is typical for I/O devices. This forms the critical timing path for the placement of $\overline{EnStart}$ in the Memory Controller and Memory Enable PALs.

Expansion Connector

Two 50-pin connectors are provided which bring out the R3051 RISController pins to allow off-board expansion. The $\overline{BusReq}$ and $\overline{BusGnt}$ pins are not presently used on this board. If DMA is to be used, the R3051 control outputs $\overline{Rd}$, $\overline{Wr}$, $\overline{Burst}$, $\overline{DataEn}$, and ALE are pulled high or low so that they remain inactive when tri-stated.

SCHEMATICS AND PAL EQUATIONS

Appendices include the System Design Example Board Schematics and the PAL equations.

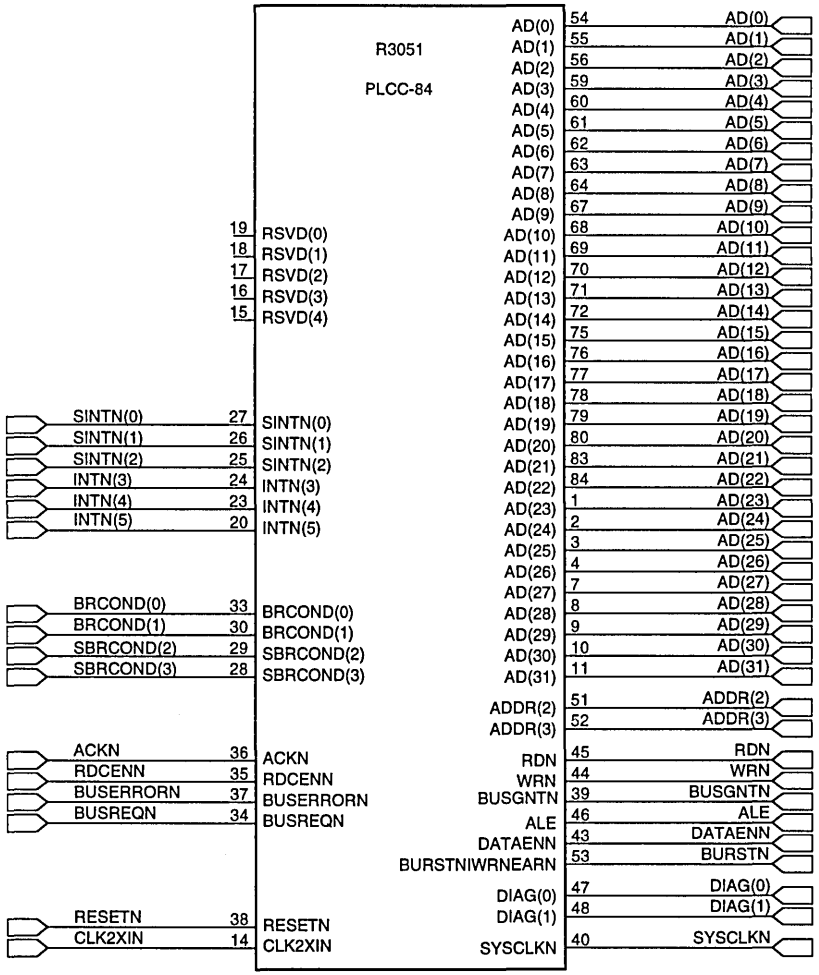


Figure 7. R3051 RISController

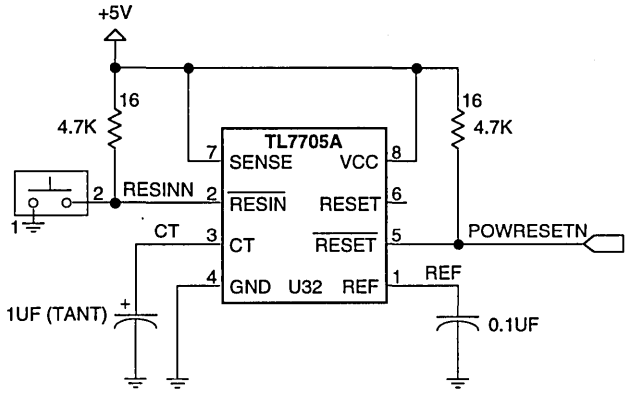


Figure 8. Reset Logic

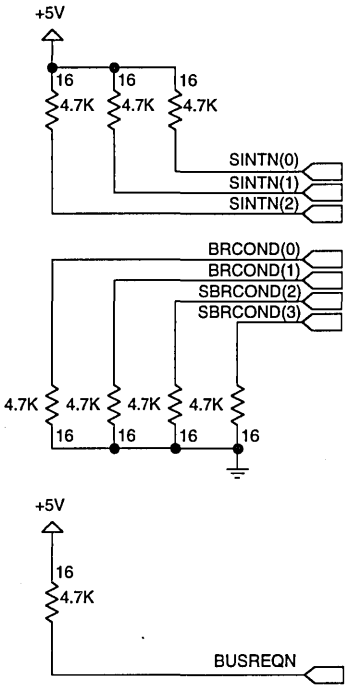


Figure 9. Unused Inputs

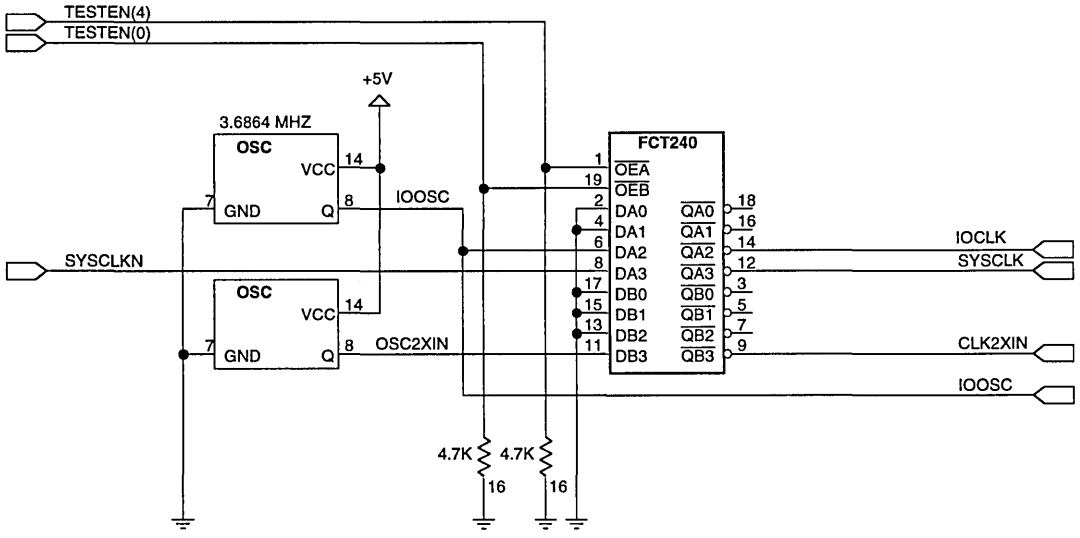


Figure 10. Clock Logic

NOTES:
MEMSPARE0 -- CARDCSN I XCSN0
MEMSPARE1 -- C4 I WRLASTN I WORLDBOOTN
MEMSPARE2 -- TESTEN I SHADOW RAM I DATAENN I XCSN1

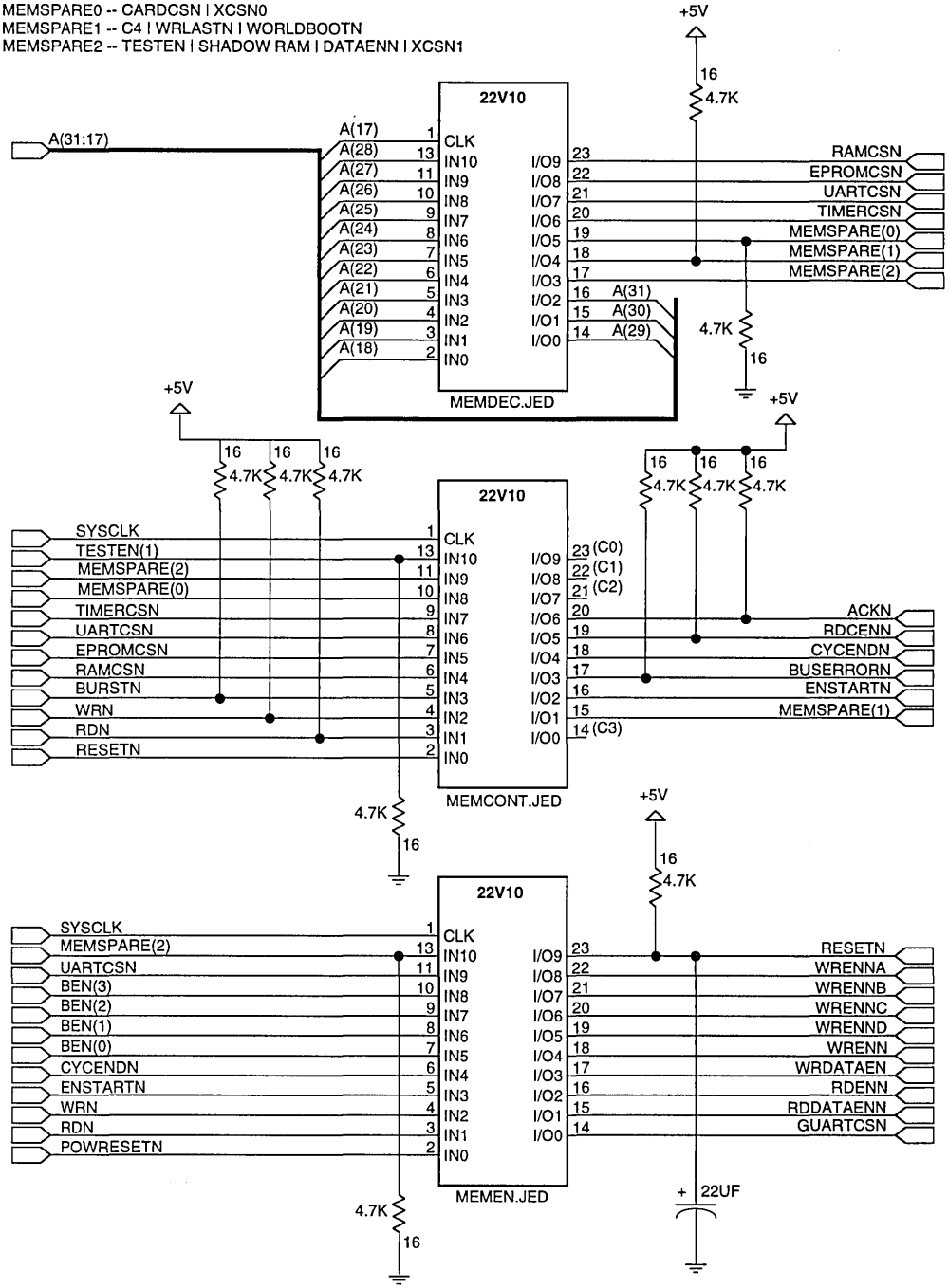


Figure 11. Memory Controller

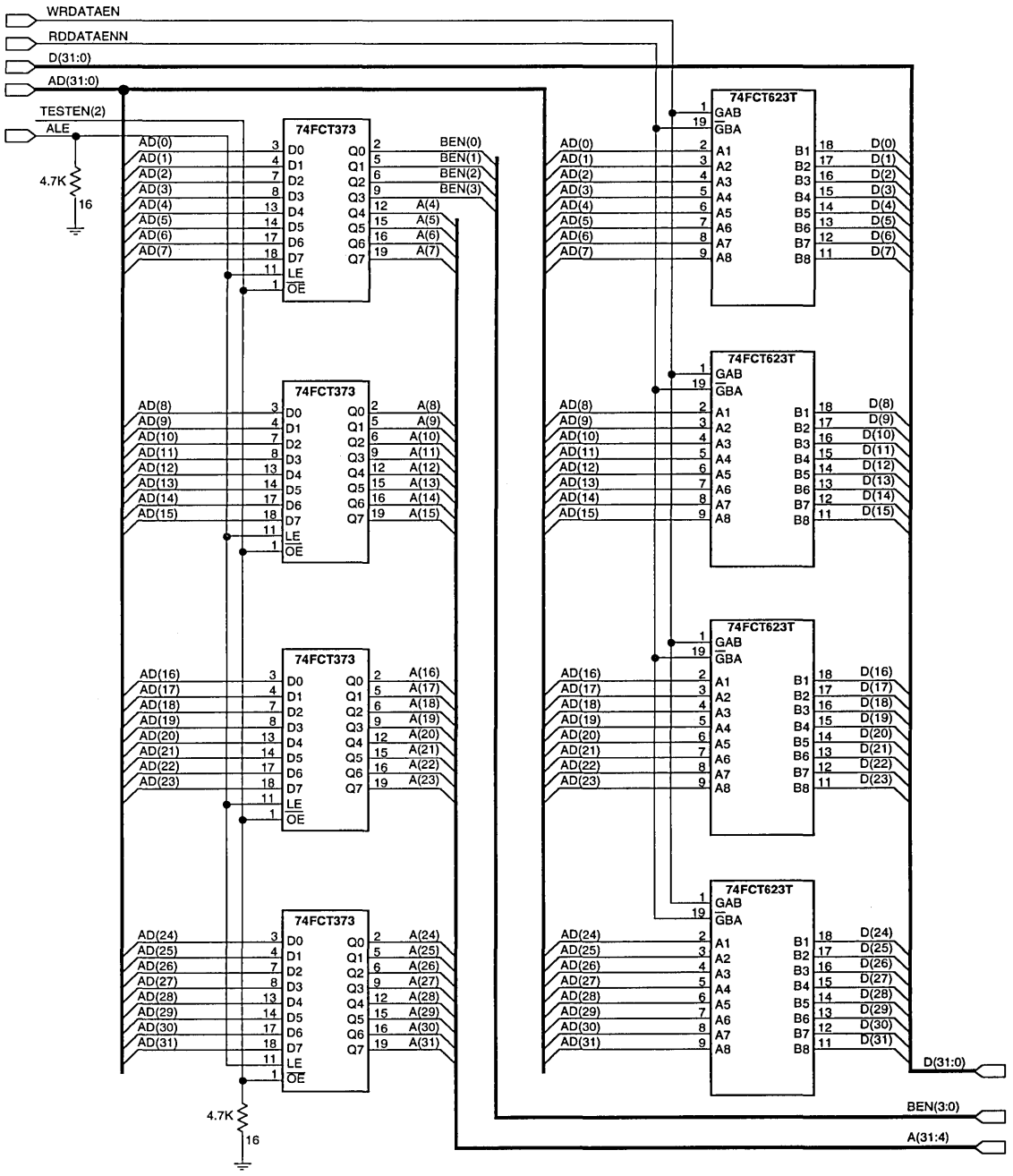
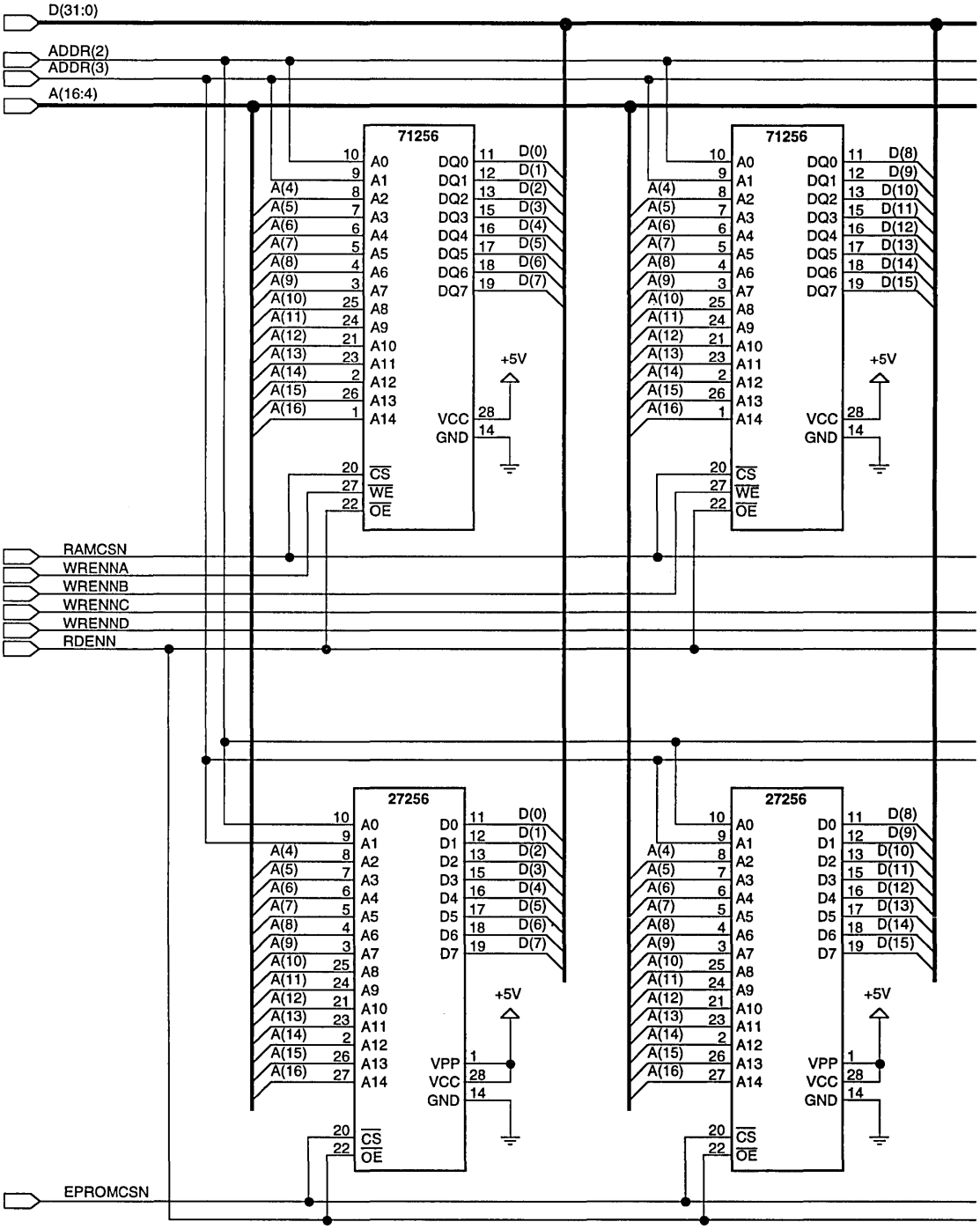


Figure 12. Address Latch Data Transceiver Demultiplexer



NOTE: BANK A - - LITTLE ENDIAN LSB BYTE 0
- - BIG ENDIAN LSB BYTE 3

Figure 13. ROM and Static RAM Memory

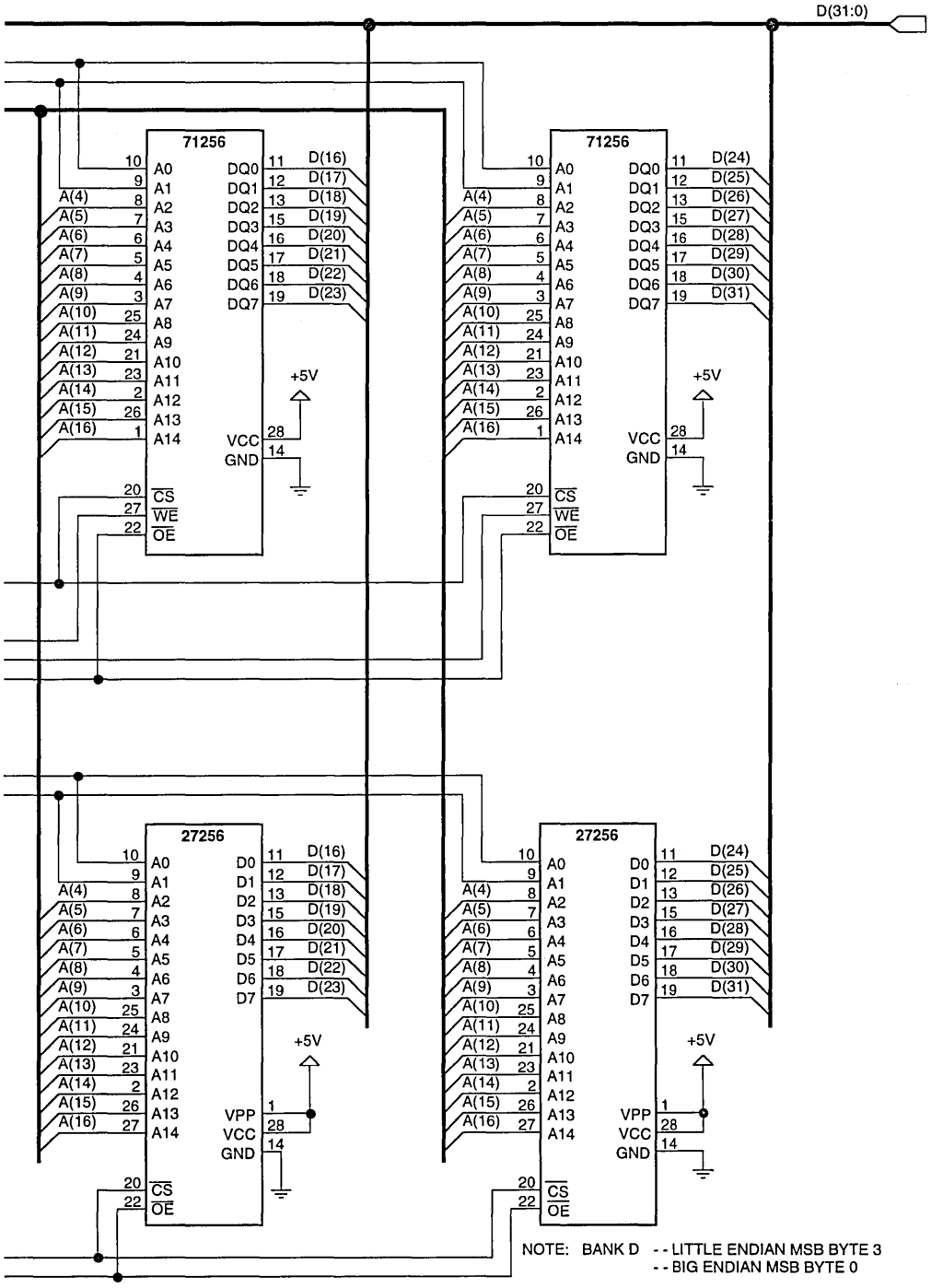


Figure 13. ROM and Static RAM Memory

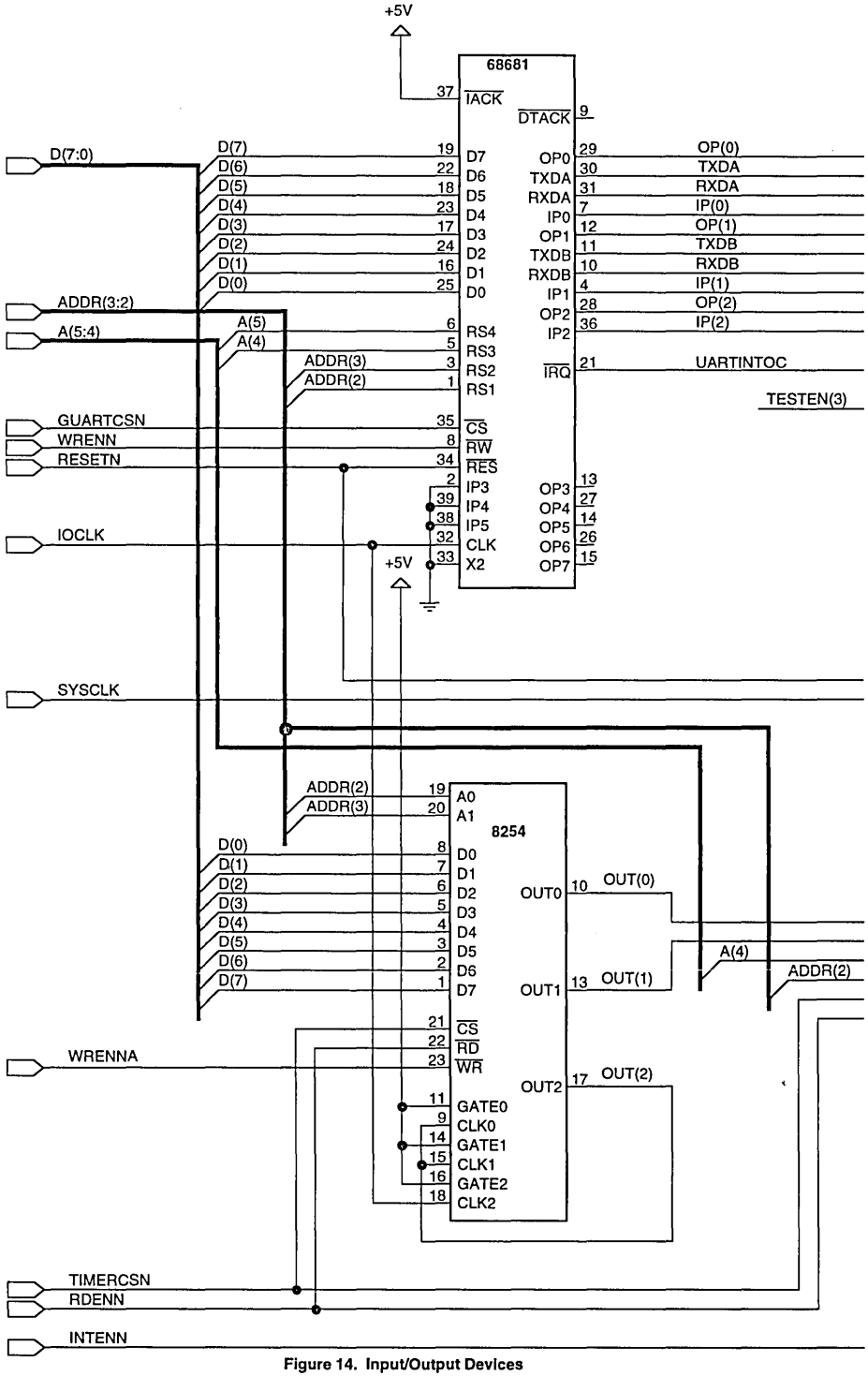


Figure 14. Input/Output Devices

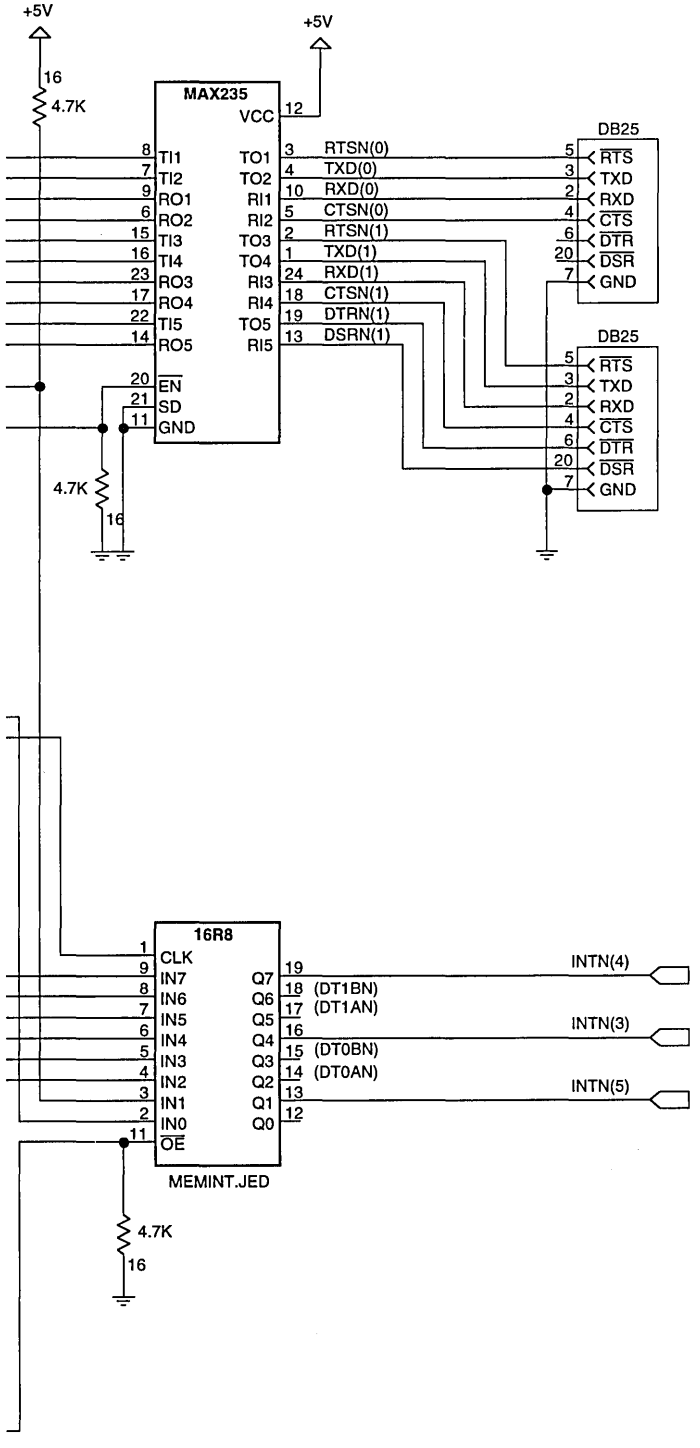


Figure 14. Input/Output Devices

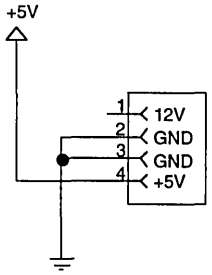


Figure 15. Power Connector

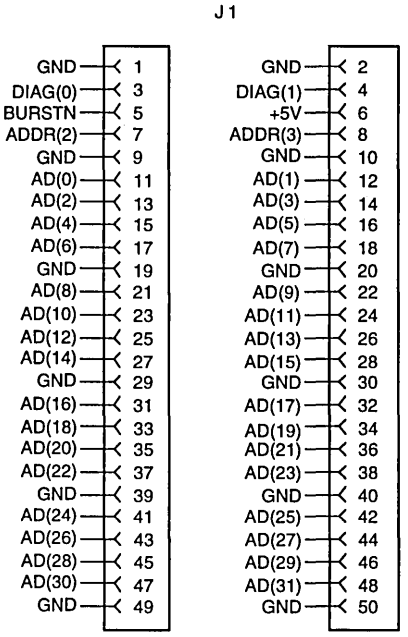


Figure 16. 50-Pin Connector

Figure 17. Spares

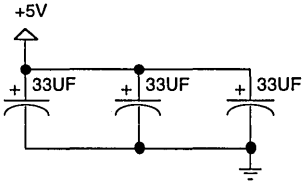


Figure 18. Primary Power Decoupling Capacitors

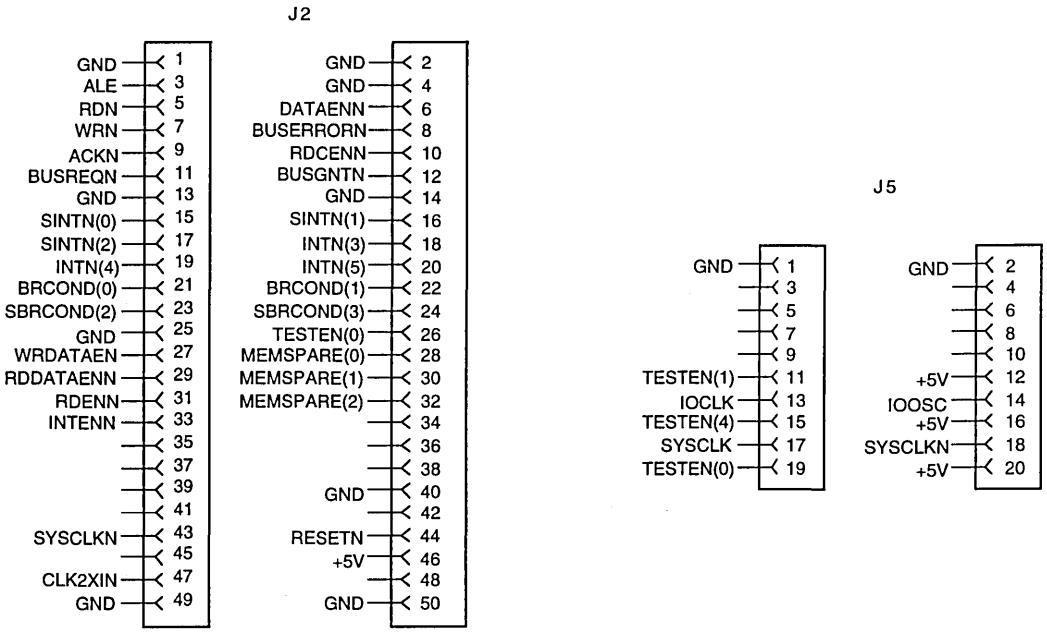


Figure 19. 50-Pin Connector

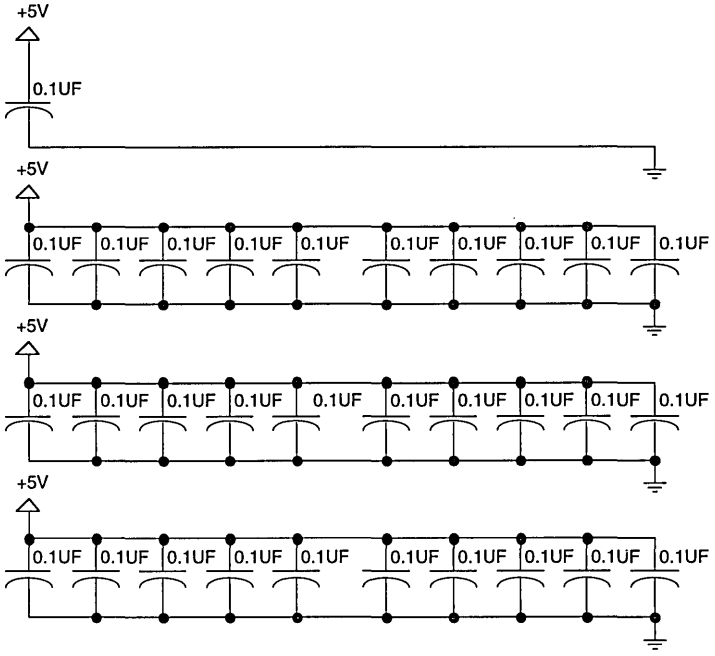


Figure 21. Decoupling Capacitors

```
{ TITLE   : MEMDEC.LPLC
        UPAL1 MEMORY AND I/O ADDRESS DECODER PAL FOR THE R305X
        BEHAVIORAL BUS EMULATOR MEMORY EVALUATION BOARD
  PURPOSE : DECODES DEMULTIPLEXED ADDRESS TO GENERATE CHIP SELECTS.
  LANG    : LPLC — TM OF CAPILANO COMPUTING SYSTEMS
  AUTHOR  : ANDY NG, IDT INC.
  UPDATES : C2503 03-18-91 AP NOTE FIRST RELEASE
}

MODULE UPAL1    ;
TITLE  UPAL1    ;
TYPE   AMD 22V10 ;

INPUTS ;
     { DEMULTIPLEXED MEMORY ADDRESS LINES }
     A17          NODE[PIN1] ;        { MSB ADDRESS LINES 31-17     }
     A18          NODE[PIN2] ;
     A19          NODE[PIN3] ;
     A20          NODE[PIN4] ;
     A21          NODE[PIN5] ;
     A22          NODE[PIN6] ;
     A23          NODE[PIN7] ;
     A24          NODE[PIN8] ;
     A25          NODE[PIN9] ;
     A26          NODE[PIN10] ;
     A27          NODE[PIN11] ;
     A28          NODE[PIN13] ;


     { OUTPUT FEEDBACK NODES (NEEDED FOR LPLC'ISM) }
     A29          NODE[PIN16] ;
     A30          NODE[PIN15] ;
     A31          NODE[PIN14] ;
     MEMSPARE0    NODE[PIN19] ;
     MEMSPARE1    NODE[PIN18] ;
     MEMSPARE2    NODE[PIN17] ;

OUTPUTS ; { ATTRIBUTES C — COMBINATIONAL, R — REGISTERED, H — HIGH, L — LOW }

     { CHIP SELECTS }
     RAMCSN       NODE[PIN23] ATTR[CL] ; { STATIC RAM CHIP SELECT      }
     EPROMCSN     NODE[PIN22] ATTR[CL] ; { EPROM CHIP SELECT           }
     UARTCSN      NODE[PIN21] ATTR[CL] ; { UNGATED UART CHIP SELECT    }
     TIMERCSN     NODE[PIN20] ATTR[CL] ; { TIMER CHIP SELECT           }

     { I/O PINS USED AS INPUTS }
     A29          NODE[PIN14] ATTR[CL] ; { MSB ADDRESS LINES 31-17     }
     A30          NODE[PIN15] ATTR[CL] ;
     A31          NODE[PIN16] ATTR[CL] ;
     MEMSPARE0    NODE[PIN19] ATTR[CL] ;
     MEMSPARE1    NODE[PIN18] ATTR[CL] ;
     MEMSPARE2    NODE[PIN17] ATTR[CL] ;

     { OUTPUT ENABLES }
     RAMCSNEN     NODE[PIN23EN] ;
     EPROMCSNEN   NODE[PIN22EN] ;
```

```
UARTCSNEN     NODE[PIN21EN] ;
TIMERCSNEN    NODE[PIN20EN] ;
A29EN         NODE[PIN14EN] ;
A30EN         NODE[PIN15EN] ;
A31EN         NODE[PIN16EN] ;
MEMSPARE0EN   NODE[PIN19EN] ;
MEMSPARE1EN   NODE[PIN18EN] ;
MEMSPARE2EN   NODE[PIN17EN] ;

{ ASYNCHRONOUS RESET AND SYNCHRONOUS PRESET NODES }
RESETEN       NODE[RESET]  ;
PRESETEN      NODE[PRESET] ;


{ 7RS382 COMPATIBLE PHYSICAL ADDRESS DECODE MAP }
{  RAM     00000000H — 0001FFFFH   32K  }
{  EPROM   1FC00000H — 1FC1FFFFH   32K  }
{  UART    1FE00000H — 1FE0003FH        }
{  TIMER   1F800000H — 1F80002CH        }


TERMS ; { LPLC "TABLE" ALGORITHM TAKES TOO LONG TO COMPILE }

{ NOTES: MEMSPARE0 IS BEING USED FOR A BOARD CHIP SELECT
     DRIVABLE BY ANOTHER MEMORY SYSTEM.  WITHOUT IT
     ASSERTED LOW, THIS BOARD WILL NOT ISSUE ANY MEMORY
     SIGNALS NOR OUTPUT ENABLE SHARED CONTROL PINS.      }
{ NOTES: MEMSPARE1 IS NOT BEING USED.  IT COULD BE USED AS AN
     OUTPUT IF IT OR THE UPAL2 OUTPUT IT IS CONNECTED TO IS
     TRISTATED.                                          }
{ NOTES: MEMSPARE2 IS BEING USED AS A TESTEN INPUT PIN TO
     TRISTATE THE OUTPUTS DURING BOARD TESTING.  ANOTHER
     USE WOULD BE FOR A BOARD CHIP SELECT — MEMCSN.
     MEMSPARE2 IS CONNECTED TO A UPAL3 INPUT PIN.        }

{ I/O PINS USED ONLY AS INPUTS }
A29EN         = 0 ;
A30EN         = 0 ;
A31EN         = 0 ;
MEMSPARE0EN   = 0 ;
MEMSPARE1EN   = 0 ;
MEMSPARE2EN   = 0 ;
A29       NOT = 0 ;
A30       NOT = 0 ;
A31       NOT = 0 ;
MEMSPARE0 NOT = 0 ;
MEMSPARE1 NOT = 0 ;
MEMSPARE2 NOT = 0 ;

{ RESET AND PRESET ARE NOT USED IN THIS PAL. }
RESETEN  = 0 ;
PRESETEN = 0 ;


RAMCSNEN      = !MEMSPARE2 ;
RAMCSN NOT    = !MEMSPARE0 AND
                !A31 AND !A30 AND !A29 AND !A28
               AND !A27 AND !A26 AND !A25 AND !A24
               AND !A23 AND !A22 AND !A21 AND !A20
```

```
                AND !A19 AND !A18 AND !A17
    ;


    EPROMCSNEN      = !MEMSPARE2 ;
    EPROMCSN NOT    = !MEMSPARE0 AND
                      !A31 AND !A30 AND !A29 AND  A28
                    AND  A27 AND  A26 AND  A25 AND  A24
                    AND  A23 AND  A22 AND !A21 AND !A20
                    AND !A19 AND !A18 AND !A17
    ;


    UARTCSNEN       = !MEMSPARE2 ;
    UARTCSN NOT     = !MEMSPARE0 AND
                      !A31 AND !A30 AND !A29 AND  A28
                    AND  A27 AND  A26 AND  A25 AND  A24
                    AND  A23 AND  A22 AND  A21 AND !A20
                    AND !A19 AND !A18 AND !A17
    ;


    TIMERCSNEN      = !MEMSPARE2 ;
    TIMERCSN NOT    = !MEMSPARE0 AND
                      !A31 AND !A30 AND !A29 AND  A28
                    AND  A27 AND  A26 AND  A25 AND  A24
                    AND  A23 AND !A22 AND !A21 AND !A20
                    AND !A19 AND !A18 AND !A17
    ;


END;
END UPAL1.
```

```
{ TITLE : MEMCONT.LPLC
        UPAL2 MEMORY CONTROLLER PAL FOR THE R305X BEHAVIORAL BUS EMULATOR
        MEMORY EVALUATION BOARD
  PURPOSE: PRODUCES READ, WRITE, AND BUS ERROR ACKNOWLEDGE CONTROLS (RDCENN,
        ACKN, BUSERRORN) BASED ON A 4 OR 5 BIT COUNTER AND CYCLE END
        STALL CYCLE (WAIT STATE) EQUATIONS.
  LANG  : LPLC — TM OF CAPILANO COMPUTING SYSTEMS
  AUTHOR : ANDY NG, IDT INC.
  UPDATES: C4B28 03-18-91 AP NOTE FIRST RELEASE
}

MODULE UPAL2   ;
TITLE  UPAL2   ;
TYPE   AMD 22V10 ;

INPUTS ;
     { REGULAR INPUT PINS }
     SYSCLK        NODE[PIN1] ;        { UN-INVERTED SYSTEM CLOCK     }
     RESETN        NODE[PIN2] ;        { MASTER RESET                 }
     RDN           NODE[PIN3] ;        { READ                         }
     WRN           NODE[PIN4] ;        { WRITE                        }
     BURSTN        NODE[PIN5] ;        { BURST READ I WRITE NEAR      }
     RAMCSN        NODE[PIN6] ;        { RAM CHIP SELECT              }
     EPROMCSN      NODE[PIN7] ;        { EPROM CHIP SELECT            }
     UARTCSN       NODE[PIN8] ;        { UART CHIP SELECT             }
     TIMERCSN      NODE[PIN9] ;        { TIMER CHIP SELECT            }
     MEMSPARE0     NODE[PIN10] ;       {                              }
     MEMSPARE2     NODE[PIN11] ;       {                              }
     TESTEN        NODE[PIN13] ;       { TEST PIN TO Z-STATE OUTPUTS  }

     { REGISTER FEEDBACK PINS }
     C WIDTH[5]    NODE[PIN15,PIN14,PIN21,PIN22,PIN23] ;
     ENSTARTN      NODE[PIN16] ;
     CYCENDN       NODE[PIN18] ;
     RDCENN        NODE[PIN19] ;
     ACKN          NODE[PIN20] ;
     BUSERRORN     NODE[PIN17] ;


OUTPUTS ; { ATTRIBUTES C — COMBINATIONAL, R — REGISTERED, H — HIGH, L — LOW  }

     { REGISTERED OUTPUT PINS }
     { BINARY UP COUNTER INPUTS MSB TO LSB C4, C3, C2, C1, C0 }
     C WIDTH[5]    NODE[PIN15,PIN14,PIN21,PIN22,PIN23] ATTR[RL] ;
     ENSTARTN      NODE[PIN16] ATTR[RL] ; { READ/WRITE OUTPUT ENABLE START }
     CYCENDN       NODE[PIN18] ATTR[RL] ; { CYCLE END (COMPOSITE ACK)     }
     RDCENN        NODE[PIN19] ATTR[RL] ; { R305X READ BUFFER CLOCK ENABLE }
     ACKN          NODE[PIN20] ATTR[RL] ; { R3050X ACKNOWLEDGE            }
     BUSERRORN     NODE[PIN17] ATTR[RL] ; { R305X BUS ERROR               }

     { OUTPUT ENABLES }
     CEN WIDTH[5]  NODE[PIN15EN,PIN14EN,PIN21EN,PIN22EN,PIN23EN] ;
     ENSTARTNEN    NODE[PIN16EN] ;
     CYCENDNEN     NODE[PIN18EN] ;
     RDCENNEN      NODE[PIN19EN] ;
     ACKNEN        NODE[PIN20EN] ;
     BUSERRORNEN   NODE[PIN17EN] ;
```

```
    { ASYNCHRONOUS RESET AND SYNCHRONOUS PRESET NODES }
    RESETEN      NODE[RESET] ;
    PRESETEN     NODE[PRESET] ;


TABLE ;

    { RESET AND PRESET ARE NOT BEING USED.                    }
    RESETEN  = 0 ;
    PRESETEN = 0 ;

    { PURPOSE: PROVIDES REGISTERED VERSION OF RDN AND WRN.

      NOTE:   QRDN AND QWRN ARE KEPT LOW ONE EXTRA CLOCK BY CYCENDN.
              THIS IS BECAUSE THE RISING EDGE OF RDN OR WRN MAY NOT
              HAVE ENOUGH HOLD TIME FROM THE RISING EDGE OF
              (BUFFERRED) SYSCLK.
      NOTE:   QRDN AND QWRN DO NOT NECESSARILY TRANSITION BACK HIGH
              BETWEEN CONSECUTIVE MEMORY CYCLES, E.G., WRITE FOLLOWED
              BY A WRITE.                                     }

    { QRDN NOT    :=  RESETN AND (!RDN OR (!QRDN AND !CYCENDN)) ; }
    { QWRN NOT    :=  RESETN AND (!WRN OR (!QWRN AND !CYCENDN)) ; }


    { PURPOSE: C[4]-C[0] PROVIDES A 5-BIT BINARY UP COUNTER.  IT IS RESET
              ANYTIME RESETN IS ASSERTED AND AT THE END
              OF EVERY MEMORY CYCLE AFTER CYCENDN IS ASSERTED.
              IT BEGINS COUNTING UP WHEN A READ OR WRITE CYCLE IS
              INITIATED.
      NOTE:   CYCENDN IS ASSUMED TO ASSERT WITH THE LAST RDCENN
              ON READS AND WITH ACKN ON WRITES.  THUS CYCENDN WILL CLEAR
              THE COUNTER WHETHER OR NOT RDN OR WRN HIGH TRANSITION
              MEETS THE REGISTER SETUP AND HOLD TIME REQUIREMENTS.      }

    { NOTE:   TO ADD A GENERAL PURPOSE READY (A.K.A. BUSYN AND WAITN)
              INPUT, CHANGE EACH OF THE COUNTER C[4:0] EQUATIONS SO
              THAT THEIR VALUE CAN BE HELD WITH AN ADDITIONAL TERM, E.G.:
              C[0]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
                         AND (   (C[0] XOR 1)
                              OR (C[0] AND !READY) ) ;
              A READY INPUT CAN BE USED FOR DUAL-PORT MEMORY INTERFACING,
              EEPROM WRITE INTERFACING, ETC.
    }

    CEN[0]  =  !TESTEN ;
    CEN[1]  =  !TESTEN ;
    CEN[2]  =  !TESTEN ;
    CEN[3]  =  !TESTEN ;
    CEN[4]  =  !TESTEN ;

    C[0]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
               AND (C[0] XOR 1) ;
    C[1]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
               AND (C[1] XOR C[0]) ;
    C[2]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
               AND (C[2] XOR (C[1] AND C[0])) ;
    C[3]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
```

```
            AND (C[3] XOR (C[2] AND C[1] AND C[0])) ;
C[4]   :=  RESETN AND CYCENDN AND (!RDN OR !WRN)
            AND (C[4] XOR (C[3] AND C[2] AND C[1] AND C[0])) ;


{ PURPOSE: ENSTARTN OUTPUT PROVIDES THE TIMING FOR THE LEADING
       EDGE OF OEN AND WEN STROBES SO THAT 1. THE ADDRESS LINES HAVE
       TIME TO BE DECODED AND 2. OE/DATA PINS HAVE TIME TO Z-STATE
       FROM READS ON THE PRECEDING CYCLE.  THE CYCENDN TERM IS
       NEEDED TO HOLD OFF A CONSECUTIVE MEMORY CYCLE, E.G., WHEN
       WRITE DEASSERTS AND REASSERTS WITHIN THE SAME CLOCK.
       ENSTARTN SHOULD NOT BE USED TO END WRITE TRANSCEIVER
       ENABLES AS IT DEASSERTS WITH THE WRITE LINE INSTEAD OF
       HOLDING FOR ONE MORE 1/2 CLOCK.                    }

ENSTARTNEN    = !TESTEN ;
ENSTARTN  NOT := !MEMSPARE0 AND RESETN AND (C >= 1) AND CYCENDN ;


{ PURPOSE: CYCLE END GOES LOW (SYNCHRONOUSLY) DURING THE LAST RDCENN ON
       READS AND DURING ACKN ON WRITES.  IT RETURNS HIGH
       SYNCHRONOUSLY BY INTERLOCKING ON THE COUNTER OUTPUTS
       WHICH COUNT ONE GREATER THAN THE ASKED FOR VALUE BEFORE
       RESETTING BACK TO ZERO (VIA CYCENDN).  THUS CYCENDN WILL
       DEASSERT ON THE SAME CLOCK AS THE RDN, WRN, OR BURSTN RISING
       EDGES REGARDLESS OF WHETHER OR NOT THOSE RISING EDGES MEET
       THE REGISTER'S SETUP AND HOLD TIMES.                }
{ NOTE:   TO FIT CYCENDN INTO A 16V8, TWO OUTPUTS MAY BE NEEDED.    }

CYCENDNEN    = !TESTEN ;
CYCENDN NOT := RESETN AND CYCENDN AND (
           (!RAMCSN   AND (C == 02H) AND !RDN AND  BURSTN)
         OR (!RAMCSN   AND (C == 08H) AND !RDN AND !BURSTN)
         OR (!RAMCSN   AND (C == 03H) AND !WRN          )
         OR (!EPROMCSN AND (C == 03H) AND !RDN AND  BURSTN)
         OR (!EPROMCSN AND (C == 0CH) AND !RDN AND !BURSTN)
         OR (!UARTCSN  AND (C == 06H) AND         BURSTN)
         OR (!TIMERCSN AND (C == 06H) AND         BURSTN)
         OR ( {!BUSERRORN} (C == 1FH)                   )
);


{ NOTE: IN THIS EXPERIMENT MEMSPARE0 IS PULLED LOW AND CAN BE
     USED TO DISABLE THIS CONTROLLER'S RDCENN, ACKN, AND BUSERRORN.
     SINCE MEMSPARE0 IS ATTACHED TO THE MEMDEC.LPLC PAL, THE
     MEMDEC PAL COULD COMBINE THE CSN'S SO THAT THESE SIGNALS
     ARE ONLY DRIVEN WHEN NEEDED.                       }

{ NOTE: ANOTHER POSSIBILITY IS TO USE MEMSPARE0 AS AN EXTRA CHIP
     SELECT.                                  }


{ PURPOSE: READ BUFFER CLOCK ENABLE IS USED BY THE R305X TO STROBE
       DATA INTO ITS INTERNAL READ BUFFERS.                }
{ NOTE:   IT IS ASSUMED THAT THE UART AND TIMER ARE
       IN UNCACHABLE MEMORY SPACE AND WILL NOT BE BURST READ.
       IF THEY ARE BURST READ, THE STATE MACHINE LOOPS 4 TIMES.    }

RDCENNEN    = !MEMSPARE0 ;
```

```
RDCENN NOT  := RESETN AND CYCENDN AND (
          (!RAMCSN AND !RDN
            AND (           (C == 02H)
                OR (!BURSTN AND (C == 04H))
                OR (!BURSTN AND (C == 06H))
                OR (!BURSTN AND (C == 08H))
              )
          )
        OR (!EPROMCSN AND !RDN
            AND (           (C == 03H)
                OR (!BURSTN AND (C == 06H))
                OR (!BURSTN AND (C == 09H))
                OR (!BURSTN AND (C == 0CH))
              )
          )
        OR (!UARTCSN AND !RDN
            AND (           (C == 06H)
              )
          )
        OR (!TIMERCSN AND !RDN
            AND (           (C == 06H)
              )
          )
);


{ PURPOSE: ACKNOWLEDGE IS PRIMARILY USED TO END WRITE CYCLES.  IT
        SHOULD BE PULSED ONE (HALF) CLOCK CYCLE BEFORE THE WRITE
        STROBE IS NEEDED.  ON READ CYCLES, ACKNOWLEDGE WILL
        IMPLICITLY BE GENERATED BY THE R305X, HOWEVER, IF OPTIMAL
        TIMING IS DESIRED, ACK SHOULD BE DRIVEN NO SOONER THAN 1
        CLOCK BEFORE THE END OF A SINGLE READ AND FOR BURSTS NO
        SOONER THAN 4 CLOCKS BEFORE THE END OF THE LAST READ.      }

ACKNEN      = !MEMSPARE0 ;
ACKN NOT    := RESETN AND CYCENDN AND (
           (!RAMCSN AND !WRN                    { WRITE CYCLE }
             AND (          (C == 03H)
               )
           )
         OR (!RAMCSN AND !RDN AND !BURSTN       { READ  CYCLE }
             AND (          (C == 05H)
               )
           )
         OR (!EPROMCSN AND !RDN AND !BURSTN     { READ  CYCLE }
             AND (          (C == 09H)
               )
           )
         OR (!UARTCSN AND !WRN AND BURSTN       { WRITE CYCLE }
             AND (          (C == 06H)
               )
           )
         OR (!TIMERCSN AND !WRN                 { WRITE CYCLE }
             AND (          (C == 06H)
               )
           )
);
```

```
    { PURPOSE: BUSERRORN SIMPLY ENDS A WAYWARD UNDECODED BUS CYCLE.  ON
          READS IT CAUSES AN EXCEPTION.  ON WRITES IT DOES NOT CAUSE
          AN EXCEPTION CONDITION FOR THE PROCESSOR.  TO DO THAT, LATCH
          BUSERRORN AND FEED IT TO AN INTERRUPT PIN OR A BRANCH
          CONDITION PIN.                                    }

    BUSERRORNEN    = !MEMSPARE0 ;
    BUSERRORN NOT := RESETN AND CYCENDN AND (
                          (C == 1FH)

    ) ;


END ;
END UPAL2.
```

```
{ TITLE   : MEMEN.LPLC
        UPAL3 MEMORY READ AND WRITE ENABLE PAL FOR THE R305X BEHAVIORAL BUS
        EMULATOR MEMORY EVALUATION BOARD
  PURPOSE : GENERATES READ AND WRITE ENABLES FOR MEMORY CONTROLS.
  LANG    : LPLC — TM OF ©APILANO COMPUTING SYSTEMS
  AUTHOR  : ANDY NG, IDT INC.
  UPDATES : C7C4F 03-18-91 AP NOTE FIRST RELEASE
}

MODULE UPAL3   ;
TITLE  UPAL3   ;
TYPE   AMD 22V10 ;

INPUTS ;
    { DEMULTIPLEXED MEMORY ADDRESS LINES }
    SYSCLK       NODE[PIN1] ;       { INVERTED SYSCLKN          }
    POWRESETN    NODE[PIN2] ;       { POWER UP RESET            }
    RDN          NODE[PIN3] ;       { READ LINE                 }
    WRN          NODE[PIN4] ;       { WRITE LINE                }
    ENSTARTN     NODE[PIN5] ;       { ENABLE START              }
    CYCENDN      NODE[PIN6] ;       { CYCLE END                 }
    BEN0         NODE[PIN7] ;       { BYTE ENABLE 0             }
    BEN1         NODE[PIN8] ;       { BYTE ENABLE 1             }
    BEN2         NODE[PIN9] ;       { BYTE ENABLE 2             }
    BEN3         NODE[PIN10] ;      { BYTE ENABLE 3             }
    UARTCSN      NODE[PIN11] ;      { UART CHIP SELECT          }
    MEMSPARE2    NODE[PIN13] ;      { SPARE INPUT               }

    { OUTPUT FEEDBACK NODES (NEEDED FOR LPLC'ISM) }
    RESETN       NODE[PIN23] ;
    WRENN        NODE[PIN18] ;
    WRDATAEN     NODE[PIN17] ;

OUTPUTS ; { ATTRIBUTES C — COMBINATIONAL, R — REGISTERED, H — HIGH, L — LOW }

    { WRITE ENABLES }
    WRENNA       NODE[PIN22] ATTR[RL] ; { WRITE ENABLE FOR BYTE 0    }
    WRENNB       NODE[PIN21] ATTR[RL] ; { WRITE ENABLE FOR BYTE 1    }
    WRENNC       NODE[PIN20] ATTR[RL] ; { WRITE ENABLE FOR BYTE 2    }
    WRENND       NODE[PIN19] ATTR[RL] ; { WRITE ENABLE FOR BYTE 3    }
    WRENN        NODE[PIN18] ATTR[RL] ; { WRITE ENABLE MOTO-TYPE I/O }
    WRDATAEN     NODE[PIN17] ATTR[RL] ; { WRITE DATA XCEIVER ENABLE  }

    { READ ENABLES }
    RDENN        NODE[PIN16] ATTR[RL] ; { READ OUTPUT ENABLE (FOR WORDS)}
    RDDATAENN    NODE[PIN15] ATTR[RL] ; { READ DATA XCEIVER ENABLE   }

    { MISCELLANEOUS CONTROLS }
    RESETN       NODE[PIN23] ATTR[RL] ; { SYNCHRONIZED RESET         }
    GUARTCSN     NODE[PIN14] ATTR[RL] ; { GATED/GUARDED UART CHIP SELECT}

    { I/O PINS USED AS INPUTS }
    { NONE }

    { OUTPUT ENABLES }
    WRENNAEN     NODE[PIN22EN] ;
    WRENNBEN     NODE[PIN21EN] ;
    WRENNCEN     NODE[PIN20EN] ;
```

```
    WRENNDEN      NODE[PIN19EN] ;
    WRENNEN       NODE[PIN18EN] ;
    WRDATAENEN    NODE[PIN17EN] ;
    RDENNEN       NODE[PIN16EN] ;
    RDDATAENNEN   NODE[PIN15EN] ;
    RESETNEN      NODE[PIN23EN] ;
    GUARTCSNEN    NODE[PIN14EN] ;

    { ASYNCHRONOUS RESET AND SYNCHRONOUS PRESET NODES }
    RESETEN       NODE[RESET]  ;
    PRESETEN      NODE[PRESET] ;

TABLE ;

    { RESET AND PRESET ARE NOT USED IN THIS PAL. }
    RESETEN  = 0 ;
    PRESETEN = 0 ;


    { PURPOSE: WRITE BYTE ENABLES AND WRITE WORD ENABLE ALLOW
          SUFFICIENT TIME FOR THE ADDRESS TO DECODE AND
          FOR A VALID CHIP SELECT BEFORE ENABLING THE
          WRITE STROBE FOR A SPECIFIC BYTE BANK.
      NOTE:   BANK A IS THE BIG ENDIAN'S LSB BYTE3 OR THE LITTLE
          ENDIAN'S LSB BYTE0.  IT ALWAYS HOLDS D(7:0).
          BANK D IS THE BIG ENDIAN'S MSB BYTE0 OR THE BIG
          ENDIAN'S MSB BYTE3.  IT ALWAYS HOLDS D(31:23).
                                              }

    WRENNAEN      = !MEMSPARE2 ;
    WRENNA    NOT := RESETN AND (
              !WRN AND !BEN0 AND !ENSTARTN AND CYCENDN
    );

    WRENNBEN      = !MEMSPARE2 ;
    WRENNB    NOT := RESETN AND (
              !WRN AND !BEN1 AND !ENSTARTN AND CYCENDN
    );

    WRENNCEN      = !MEMSPARE2 ;
    WRENNC    NOT := RESETN AND (
              !WRN AND !BEN2 AND !ENSTARTN AND CYCENDN
    );

    WRENNDEN      = !MEMSPARE2 ;
    WRENND    NOT := RESETN AND (
              !WRN AND !BEN3 AND !ENSTARTN AND CYCENDN
    );

    { PURPOSE: WRENN IS USED TO PROVIDE A WRITE LINE THAT HOLDS
          LOW FOR AN EXTRA CYCLE, SO THAT IT CAN BE USED FOR
          MOTOROLA-TYPE I/O DEVICES ON THEIR MULTIPLEXED
          READ/WRITE LINE.                    }

    WRENNEN       = !MEMSPARE2 ;
    WRENN     NOT := RESETN AND (
                (!WRN AND CYCENDN)
              OR (!WRENN AND !CYCENDN)
```

```
);

{ PURPOSE: WRDATAEN AND RDDATAENN DRIVE THE OUTPUT ENABLE
       CONTROLS ON A FCT623T TRANSCEIVER BANK FOR THE
       DATA BUS.  THE CONTROLS CAN BE USED FOR ANY
       DUAL-OUTPUT ENABLE TRANSCEIVER (1 FOR EACH
       DIRECTION.  OUTPUT ENABLE/DIRECTION CONTROLLED
       TRANSCEIVERS (FCT245) REQUIRE MORE INTERFACING
       IF OUTPUT CONTENTION IS TO BE AVOIDED BY
       ONLY CHANGING THE DIRECTION WHEN THE OUTPUTS ARE
       DISABLED.                                  }

{ NOTE:   WRITE DATA ENABLE DEASSERTS ONE CLOCK AFTER
       WRN DOES TO PROVIDE SUFFICIENT HOLD TIME FOR THE
       WRITE DATA INTO THE MEMORY (SEE UPAL2 QWRN FOR A
       MORE DETAILED EXPLANATION).
 NOTE:   WRDATAEN IS ACTIVE HIGH FOR THE FCT623T OUTPUT ENABLE
       CONTROL.  FOR THE FCT861 OUTPUT ENABLES, USE ACTIVE
       LOW.
 NOTES:  THE FIRST OR-TERM ASSERTS WRDATAEN WHILE THE SECOND
       OR-TERM DEASSERTS WRDATAEN.                }

WRDATAENEN     = !MEMSPARE2 ;
WRDATAEN       := RESETN AND (
                  (!WRN AND !ENSTARTN)
                OR (WRDATAEN AND (!ENSTARTN OR !CYCENDN))
);

RDENNEN        = !MEMSPARE2 ;
RDENN      NOT := RESETN AND (
                  !RDN AND !ENSTARTN AND CYCENDN
);

{ PURPOSE: RDDATAENN IS CONNECTED TO THE MEMORY BOARD'S
       DATA TRANSCEIVER OUTPUT ENABLE (FCT623T OR FCT861)
       AND ONLY ENABLES FOR THIS BOARD'S CHIP SELECTS.
       IF THE MEMORY CONTROLLER IS USED FOR ANOTHER
       BOARD'S MEMORY, THEN THE TRANSCEIVER OUTPUT ENABLE
       SHOULD BE DISABLED FOR THOSE CHIP SELECTS (VIA
       MEMSPARE2.                                 }

{ NOTE:   IN MOST SYSTEMS, R305X'S DATAENN OUTPUT CAN BE
       CONNECTED DIRECTLY TO THE TRANSCEIVER ENABLE PIN
       INSTEAD OF USING A SYNTHESIZED RDDATAENN.      }

RDDATAENNEN    = !MEMSPARE2 ;
RDDATAENN  NOT := RESETN AND (
                  !RDN AND !ENSTARTN AND CYCENDN
);


{ PURPOSE: RESET SYNCHRONIZES THE POWER UP RESET FOR THE
       MEMORY CONTROLLER STATE MACHINES AND FOR THE R305X. }

RESETNEN       = !MEMSPARE2 ;
RESETN     NOT := !POWRESETN ;


{ PURPOSE: GUARDED/GATED UART CHIP SELECT, GUARTCSN GATES
```

```
        UARTCSN BECAUSE THE UART BEING USED HAS A MOTOROLA-
        TYPE I/O DEVICE INTERFACE WHICH MULTIPLEXES ITS
        READ/WRITE INPUT PIN SUCH THAT THE CHIP SELECT MUST
        STROBE IN OR OUT DATA.  THIS IS IN CONTRAST TO AN
        INTEL-TYPE I/O DEVICE INTERFACE WHICH WOULD HAVE A
        SEPARATE READ STROBE AND WRITE STROBE AS WELL AS A
        CHIP SELECT.  IT IS IMPORTANT NOT TO HAVE A
        GLITCH (FROM ADDRESS DECODING THE CHIP SELECT) ON
        READS IN ORDER TO ALLOW THE I/O DEVICE TO UPDATE
        FIFO POINTERS, ETC.  THUS GUARTCSN STARTS LATE AND
        ENDS EARLY, SO THAT READ/WRITE IS HELD VALID
        THROUGHOUT THE CHIP SELECT.                    }

    GUARTCSNEN      = !MEMSPARE2 ;
    GUARTCSN    NOT := RESETN AND (
                    !UARTCSN AND !ENSTARTN AND CYCENDN
    );

END;
END UPAL3.
```

```
{ TITLE  : MEMINT.LPLC
        UPAL4 MEMORY I/O INTERRUPT CONTROLLER PAL FOR THE R305X BEHAVIORAL
        BUS EMULATOR MEMORY EVALUATION BOARD
  PURPOSE: REPLICATES THE TIMER/UART INTERRUPT CONTROLLER ON THE 7RS382 BOARD.
        ADDITIONAL FUSE BITS ADDED FOR 16V8 COMPATIBILITY.
  LANG   : LPLC — TM OF CAPILANO COMPUTING SYSTEMS
  AUTHOR : IDT INC.
  UPDATES: C3F98 01-04-91 16V8 PCB VERSION FIRST RELEASE A.N.
}


{ U24A_382 INTERRUPT PAL}
{ 1-2-90,12-14-89 }
{JEDEC file's CHECKSUM = 379E } { NOTE: 01-04-91 — NOT APPLICABLE TO 16V8 }

{    CONTROL PAL FOR 8254 TIMER'S AND UART INTERRUPT
     USED FOR EVALUATION BOARD 382 }

MODULE U24A_382;
TITLE U24A_382;
TYPE MMI 16R8;

{ FUSE BITS FOR 16V8 FAMILY ATTRIBUTES USED AS A 16R8 }
FUSE 2048..2079 00000000000000000000000000000000 ;
FUSE 2080..2111 00000000000000000000000000000000 ;
FUSE 2112..2143 00000000000000001111111111111111 ;
FUSE 2144..2175 11111111111111111111111111111111 ;
FUSE 2176..2193 11111111111111101            ;

INPUTS;

MRES/      NODE[PIN2];
UARTINT/   NODE[PIN3];
PMRD/      NODE[PIN4];
CSTIM/     NODE[PIN5];
EA02       NODE[PIN6];
EA04       NODE[PIN7];
OUT1       NODE[PIN8]; {input from Timer output OUT1}
OUT0       NODE[PIN9]; {input from Timer output OUT0}

DT0A/      NODE[PIN14]; {feedback}
DT0B/      NODE[PIN15]; {feedback}
T0INT/     NODE[PIN16]; {feedback}

DT1A/      NODE[PIN17]; {feedback}
DT1B/      NODE[PIN18]; {feedback}
T1INT/     NODE[PIN19]; {feedback}

OUTPUTS;

UINT5/     NODE[PIN13];
DT0A/      NODE[PIN14];
DT0B/      NODE[PIN15];
T0INT/     NODE[PIN16]; { goes to R3000's UINT3}

DT1A/      NODE[PIN17];
DT1B/      NODE[PIN18];
T1INT/     NODE[PIN19]; { goes to R3000's UINT4}
```

```
TABLE;
{   8254 TIMER generates 2 square-wave outputs OUT0 and OUT1.
    When OUT0 goes from high to low, this PAL asserts interrupt
    T0INT/, which will interrupt R3000 through UINT3.
    Same scheme applies to OUT1, T1INT/ and UINT4.
    Reading physical addresses 1F80 0010 and 1F80 0014 (which are
    virtual addresses BF80 0010 and BF80 0014 in this 382 board)
    will clear interrupt UINT3 and UINT4, respectively.

    This PAL also synchronizes UART interrupt signal }


DT0A/    :=   OUT0;     {delay TIMER's OUT0 through a register}
DT0B/    :=   DT0A/;    {delay again}
T0INT/ NOT :=   MRES/ AND
          ((NOT DT0A/ AND DT0B/) OR
          (NOT T0INT/ AND (NOT EA04 OR EA02 OR CSTIM/ OR PMRD/)));

DT1A/    :=   OUT1;
DT1B/    :=   DT1A/;
T1INT/ NOT :=   MRES/ AND
          ((NOT DT1A/ AND DT1B/) OR
          (NOT T1INT/ AND (NOT EA04 OR NOT EA02 OR CSTIM/ OR PMRD/)));

UINT5/   :=   UARTINT/ OR NOT MRES/ ;
          {put UART's interrupt through a register to synchronize
          it with R3000 clock }
END;
END U24A_382.
```

Integrated Device Technology, Inc.

DESIGNING A DISCRETE DRAM CONTROLLER FOR THE R3051 RISCONTROLLER™ FAMILY

APPLICATION NOTE AN-90

By Bob Napaa

INTRODUCTION

The IDT R3051™ RISController™ family utilizes a high-performance computing core to achieve high performance across a variety of applications. Further, the amount of cache incorporated in the R3051 family allow these CPUs to achieve very high performance even with simple, low speed, low cost memory sub-systems.

The R3051 RISController CPU family includes a full R3000A core RISC processor, and thus is fully software compatible with the standard MIPS processor. In order to provide high-bandwidth to the CPU core, the family also incorporates on-chip up to 8 kB of instruction cache and 2 kB of data cache. The external memory interface from the R3051 family is very flexible, and allows a wide variety of implementations according to the price / performance goals of the application. For a detailed reference to the system interface of the R3051 family, the reader is advised to refer to the "R3051 Family Hardware User's Manual".

This applications note is a design example on the interface to a non-interleaved DRAM memory sub-system. The goals of this sub-system are to provide a simple, extensible memory interface using off-the-shelf components, and to illustrate basic design techniques for systems using an R3051 family CPU.

GENERAL DESCRIPTION OF THE DRAM SYSTEM

Figure 1 illustrates a typical system based on the R3051 RISController family. The R3051 family uses a double-frequency input clock for its internal operation and provides a nominal frequency reference clock output for the external system. This output clock, $\overline{\text{SysClk}}$,synchronizes the external memory sub-systems to the R3051.

Memory transactions from the R3051 use a single, time multiplexed 32-bit address and data bus and a simple set of control signals. External logic then performs address demultiplexing and decoding, memory control, interface timing, and data path control.

The system shown in Figure 1 runs at 25 Mhz (2x clock = 50 Mhz). The R3051 interfaces to a DRAM system as the main

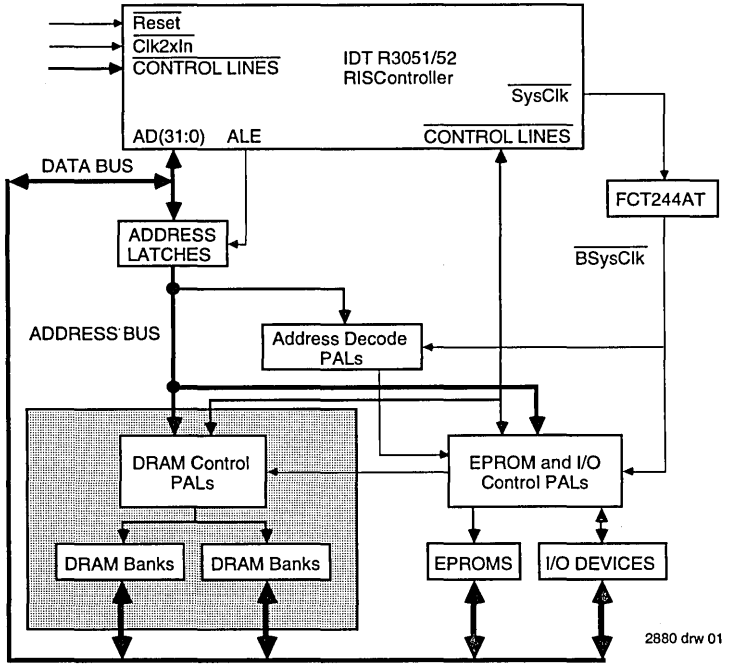


Figure 1. R3051 RISController Family Based System

memory, to an EPROM system and to various I/O devices and controllers. Address latches decouple the address bus from the data bus. Address decoders select among the various external modules. The output clock from the R3051 ($\overline{SysClk}$) is buffered ($\overline{BSysClk}$) to reduce the loading effect and to provide clock drive capability with minimum clock skew for the system. This applications note will focus on the DRAM control and data path sub-system.

The main DRAM memory system is based on 1 to 4 banks of non-interleaved DRAMs with 80 nsec of access time (t_{RAC} = 80 nsec). The density of the DRAMs used is 256K x 4 to provide a maximum memory space of 4 Mbytes. The DRAM memory space occupies the lower 4 Mbytes of the physical memory space (A21:A0). Figure 2 illustrates the architecture of the main DRAM memory system.

Table 1 illustrates the decoding scheme used in accessing the DRAM memory space. To simplify address decoding, software will insure that all references to the DRAM memory occur with address bit A(22) low, and thus only that bit will be used in the decoding. Address bits A(21:20) will select among the four banks, and the $\overline{Rd}$ and $\overline{Wr}$ outputs from the R3051 differentiate between read and write accesses.

Each 1MB bank of DRAMs is individually controlled by separate $\overline{RAS}$ and $\overline{CAS}$ control signals. Thus, each bank may be independently selected. The banks are arranged so that each bank represents a single, contiguous range of 1MB (as opposed to an interleaved memory structure).

Data buffers isolate the DRAM banks from the R3051 data bus to reduce the loading effect and to prevent any bus contentions between the R3051 and the DRAMs from occurring. Note that this also alleviates concerns about the relatively slow tri-state times associated with DRAM devices. The data buffers selected are actually bi-directional latching transceivers; the use of a latching transceiver greatly simplified the timing control of the DRAM accesses, as will be described later.

DRAM addresses are provided by multiplexing the latched R3051 address bus, using IDT FBT2827B memory drivers. This device type was chosen based on its ability to drive large capacitive loads, such as that found when driving 32 DRAMs. A single FBT output has sufficient drive to drive all four banks of the DRAM sub-system.

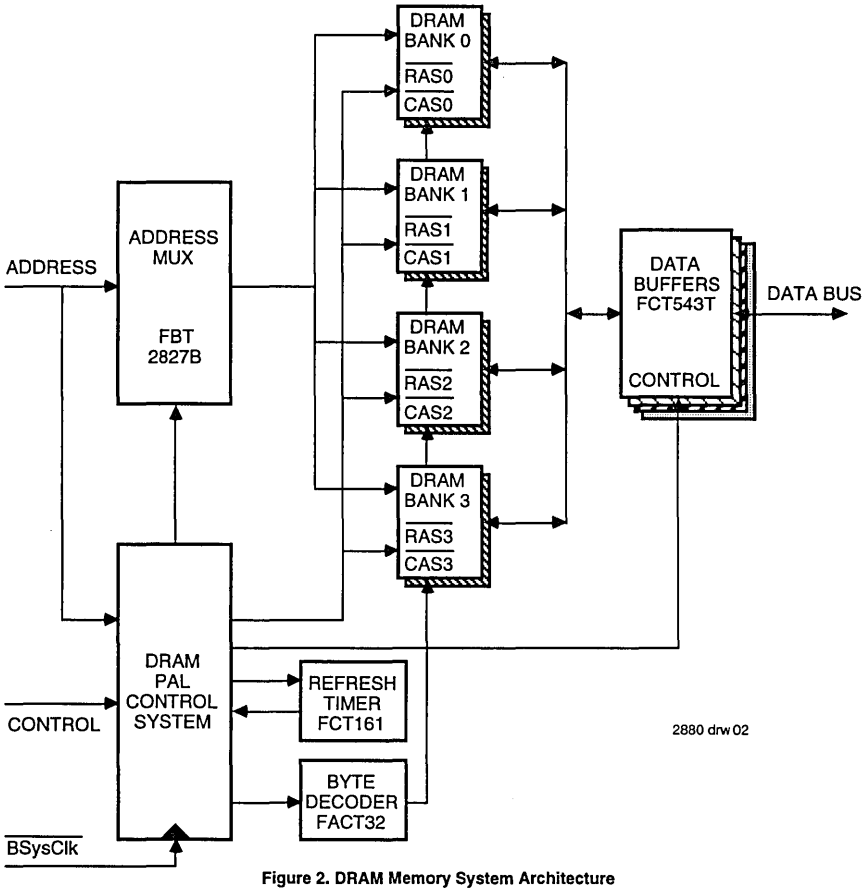


Figure 2. DRAM Memory System Architecture

A22	0	0	0	0	0	0	0	0	1	1	X
A21	0	0	1	0	0	0	1	1	X	X	X
A20	0	1	0	1	0	1	0	0	X	X	X
$\overline{WR}$	1	1	1	1	0	0	0	0	1	0	1
$\overline{RD}$	0	0	0	0	1	1	1	1	0	1	1
SELECTION	**READ BANK 0**	**READ BANK 1**	**READ BANK 2**	**READ BANK 3**	**WRITE BANK 0**	**WRITE BANK 1**	**WRITE BANK 2**	**WRITE BANK 3**	**READ OUTSIDE DRAM SPACE**	**WRITE OUTSIDE DRAM SPACE**	**NO ACCESS**

Table 1. DRAM Memory Space Decoding

In an R3051 system, it is possible to perform a 32-bit read access even when smaller data elements are requested. However, on writes, it is important to enable only those bytes which are actually being written by the CPU. The R3051 bus interface provides four individual byte enables to indicate which byte lanes are involved in a particular transfer. The DRAM sub-system uses a byte decoder (OR gate) to individually select from 1 to 4 bytes for write accesses. Each write byte enable is connected to those DRAMs which reside on that particular byte lane (across the multiple banks)

An 8-bit refresh timer requests the refreshing of the DRAMs every 9.6 µsec. Although this is more frequent than is actually required by the DRAMs, the use of this value simplified the control logic associated with page mode write. DRAMs require that $\overline{RAS}$ be maintained low no longer than 10µsec; by choosing a refresh value smaller than this maximum time, the system is assured that maximum $\overline{RAS}$ low time will not be violated.The operation of the DRAM memory system is synchronized by $\overline{BSysClk}$.

STATE MACHINE IMPLEMENTATION

A simple state machine is used to perform the major aspects of DRAM control. The state machine uses a simple four-bit counter (C(3:0)) to dictate the timing for the DRAM control and CPU response, and is sequenced using $\overline{BSysClk}$. There are nine major states to the state machine, as illustrated in figure 3; these states are dictated by the type of transfer requested and the state the DRAM control logic was left in by the prior transfer. Three PALs are required to implement the entire DRAM control logic.

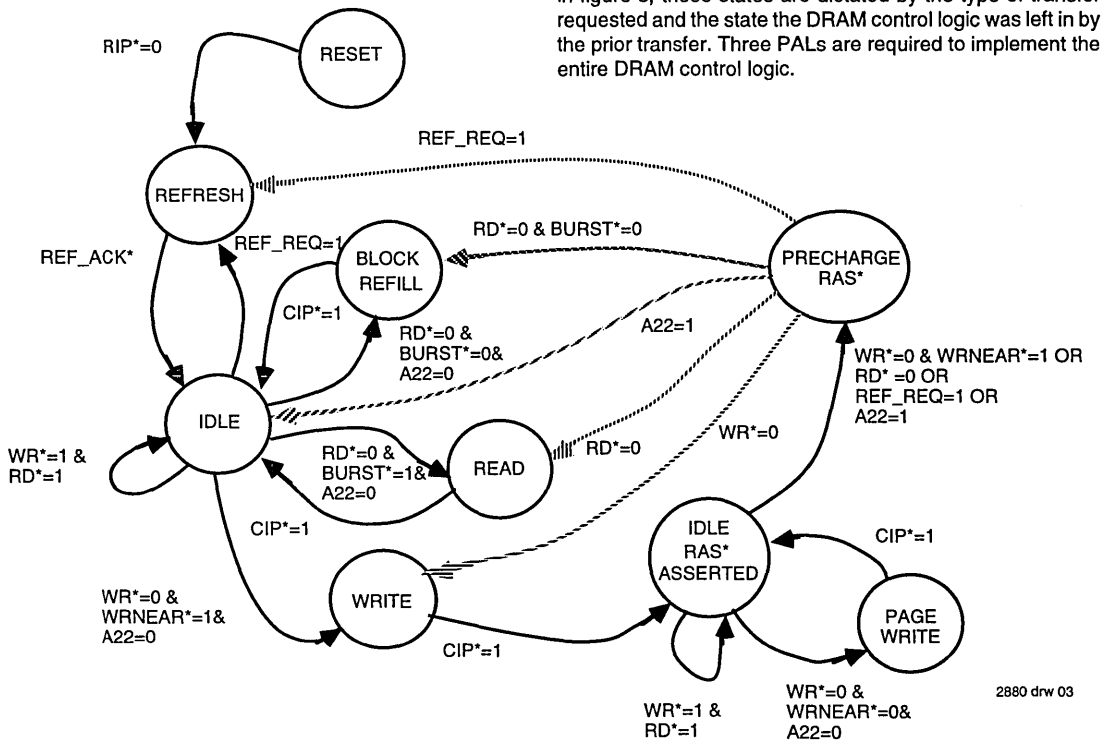


Figure 3. State Machine

The state machine uses the $\overline{\text{Reset}}$ pulse to reset its internal states and to synchronize its operation to the R3051. During the RESET state, it also performs one refresh cycle before entering the IDLE state.

In the IDLE state, the state machine arbitrates between a refresh cycle and a bus access. A DRAM bus access is started whenever $\overline{\text{Rd}}$ or $\overline{\text{Wr}}$ are asserted and A22 is low. A refresh request is detected using the REF_REQ (Refresh_Request) pulse from the refresh timer.

The state machine supports 4 types of bus accesses: "Block refill read", "Single read", "Single write" and "Page write", according to the types of transfers which the R3051 may request.

After a "Single write" or a "Page write" access, the machine enters the IDLE $\overline{\text{RAS}}$ ASSERTED state. This state is very much analogous to the IDLE state, except that the $\overline{\text{RAS}}$ control signal to the DRAMs remains asserted. This state allows subsequent "near" writes to be retired using page mode accesses, which are much quicker than standard accesses. When the IDLE $\overline{\text{RAS}}$ ASSERTED state must be exited (i.e. an action other than near write is requested) the $\overline{\text{RAS}}$ signal must be pre-charged prior to another DRAM transaction.

THE DRAM MEMORY SYSTEM IMPLEMENTATION DETAIL

The DRAM memory system consists of the control system, the address path and the data path as illustrated earlier in Figure 2.

PAL System

The state machine and control PAL system consists of 3 standard speed PALs: Pal 1 (PAL22V10-10), Pal 2 (PAL20R8-10) and Pal 3 (PAL16R8-10). Figure 4 illustrates the control system and the address path. The PAL equations are included in the appendix to this applications note.

Pal 1 is driven by $\overline{\text{SysClk}}$ directly. This allows the $\overline{\text{CIP}}$ line to detect transitions on the $\overline{\text{Rd}}$ and $\overline{\text{Wr}}$ signals from the R3051. Signals generated by Pal 1 include:

- 4 $\overline{\text{RAS}}$ signals (one per DRAM bank)
- The $\overline{\text{DRAM\_ACK}}$ and $\overline{\text{DRAM\_RDCEN}}$ response signals to the R3051 family CPU.
  These signals are used to provide termination response to the processor.
- The $\overline{\text{CIP}}$ ($\overline{\text{Cycle\_In\_Progress}}$) indicates to the rest of the control system that a bus access is being performed.
- The $\overline{\text{DRAM\_WN}}$ ($\overline{\text{DRAM\_WrNear}}$) signal indicates that the $\overline{\text{RAS}}$ signals are kept asserted after a "Single write" or a "Page write" access.

Pal 2 is also driven by $\overline{\text{SysClk}}$ directly. Pal 2 generates:

- 4 $\overline{\text{CAS}}$ signals (one per DRAM bank)
- DRAM_LE (DRAM_Latch_Enable), which latches the read data into the data buffers.
- The $\overline{\text{S}}$ ($\overline{\text{Select}}$) controls the memory drivers selection.
- The T/$\overline{\text{R}}$ (Transmit/$\overline{\text{Receive}}$) controls the data buffers during read accesses.
- The $\overline{\text{DRAM\_WR}}$ ($\overline{\text{DRAM\_Write}}$), used during write accesses.

Pal 3 uses the buffered $\overline{\text{CIP}}$ signal ($\overline{\text{BCIP}}$) which is delayed with respect to $\overline{\text{CIP}}$ by the buffer propagation delay. This is important to ensure the proper operation of Pal 3, which is driven by the buffered $\overline{\text{SysClk}}$ ($\overline{\text{BSysClk}}$). Pal 3 generates the master 4-bit counter. It also generates:

- The $\overline{\text{RIP}}$ ($\overline{\text{Reset\_In\_Progress}}$), which indicates that a reset cycle is being performed.
- The $\overline{\text{REF\_ACK}}$ ($\overline{\text{Refresh\_Acknowledge}}$) signals that a refresh cycle is being performed.
- The $\overline{\text{GATE\_COUNTER}}$ controls the operation of the counter when transitioning between bus accesses and refresh accesses.

Refresh Timer

The refresh timer consists of 2 "74FCT161" counters cascaded together as shown in Figure 4. The refresh timer issues a REF_REQ pulse every 9.6 µsec. The refresh timer is loaded with the value b00001111 after each refresh. It is incremented by one for every clock cycle. At value b11111111, it will issue the REF_REQ pulse. This amounts to a total count of 240 which at 25 Mhz reflects a 9.6 µsec refresh period.

The refresh period is set to be shorter than the maximum 15.5 µsec refresh period that most DRAM require. The refresh interval has been set to 9.6 µsec in order not to violate the $\overline{\text{RAS}}$ maximum pulse width of 10 µsec (t_{RAS} = 10 µsec max). In an IDLE $\overline{\text{RAS}}$ ASSERTED state, the $\overline{\text{RAS}}$ signals are left asserted while the $\overline{\text{CAS}}$ signals are de-asserted.

Byte Decoding

The byte decoding uses a "74FACT32" OR gate to OR the $\overline{\text{BE}}$ signals from the R3051 with the $\overline{\text{DRAM\_WR}}$ signal to produce the write-byte signals WB(3:0). The $\overline{\text{DRAM\_WR}}$ signal ensures that the $\overline{\text{WB}}$(3:0) are only asserted during DRAM write accesses and that the $\overline{\text{WB}}$(3:0) meet the "write command hold time" (t_{WCH} = 20 nsec) of the DRAMs. It also ensure that the $\overline{\text{WB}}$(3:0) are asserted before the $\overline{\text{CAS}}$ signals for "Early Write" accesses. Every $\overline{\text{WB}}$ signal enables one byte of the DRAM banks and of the data buffers during write accesses to allow for partial word write operations. The $\overline{\text{WB}}$(3:0) are always issued one clock cycle before the $\overline{\text{CAS}}$ signals are asserted, in order to meet the timing requirements for a DRAM "Early Write" cycle.

Address Path

The DRAM address path consists of 2 "74FBT2827B" memory drivers to multiplex the row and column address of the DRAMs. The "FBT2827" have a 25 Ω series resistance incorporated in the output buffers and are used to drive multiple memory banks with large capacitive loading. The $\overline{\text{S}}$ bit from Pal 2 selects between the row address and the column address that drive all the DRAM banks. Figure 4 illustrates the address path architecture. The address to the DRAMs is always set one clock cycle before the assertion of either the $\overline{\text{RAS}}$ or the $\overline{\text{CAS}}$ signals, in order to guarantee proper address set-up time to the DRAMs.

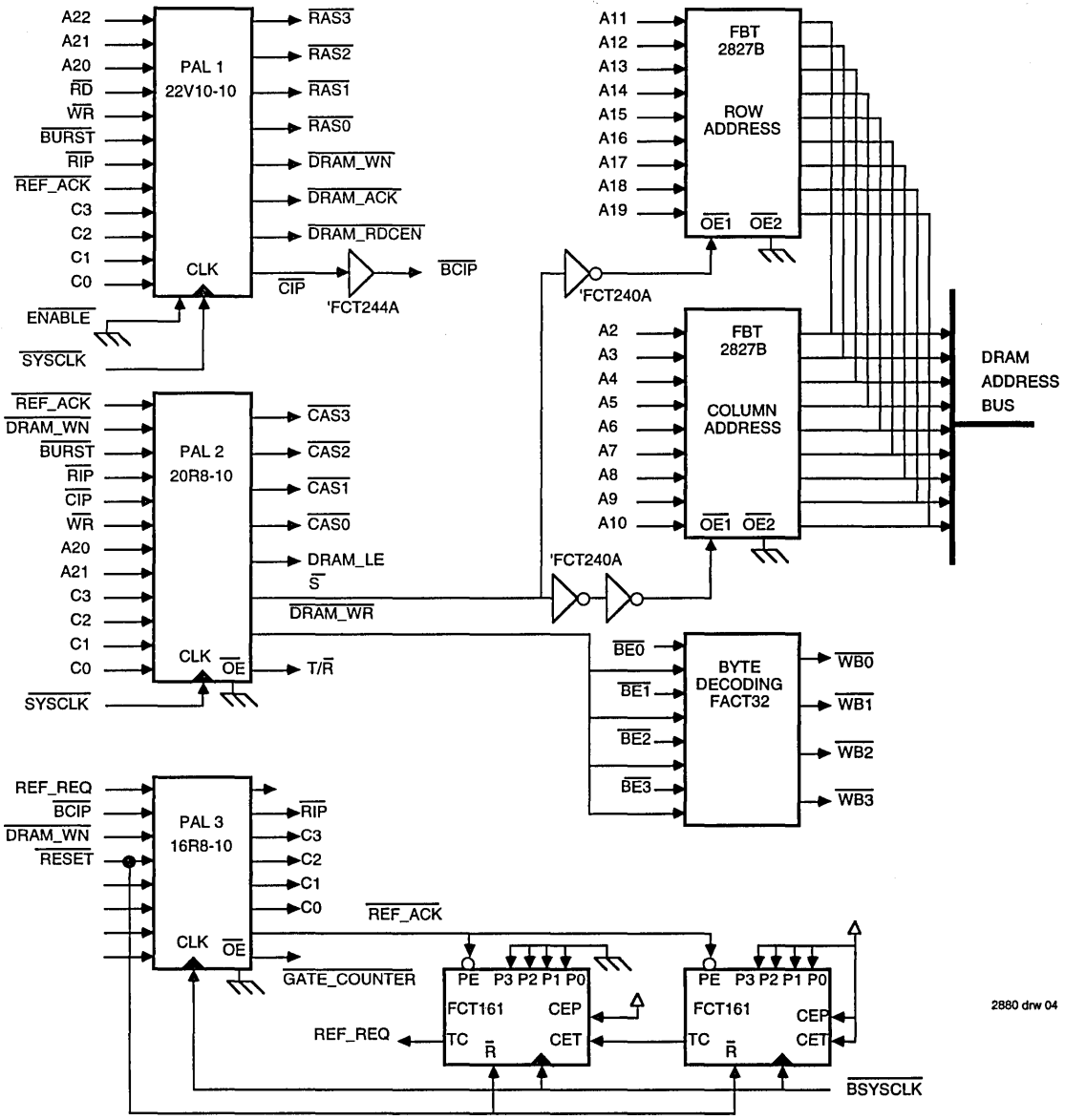


Figure 4. Control System and Address Path

Data Path

The data path consists of the DRAM banks and 4 74FCT543 latched transceivers. Figure 5 illustrates the architecture of the data path and of the data buffers. Latching transceivers are used to allow more access time to the DRAMs; the data is captured by the latches one-half cycle before they are needed by the CPU. During this half-cycle, the data propagates through the buffer; if traditional buffering transceivers had been used, the buffer propagation delay would have occurred at the expense of the DRAM access time.

Up to four banks of DRAMs are used, with each bank having its own set of $\overline{RAS}$ and $\overline{CAS}$ signals to minimize the loading impact of multiple DRAM devices. Address bits A21 and A20 determine the bank selection.

The latched transceivers serve three roles in the DRAM sub-system: they isolate the DRAMs from the A/D bus of the

R3051 to minimize loading; they latch the data from the DRAMs on reads to allow a better timing model; and they are used to prevent bus contention from occurring at the end of a read (as the processor begins another transaction). The R3051 is connected to the A bus of the transceivers, and the DRAM system is connected to the B bus.

In a processor write access, the R3051 drives both the address and the data. In this case the latches are left transparent to pass the processor data through directly to the DRAMs. Only those transceivers whose byte lanes are involved in the write are output enabled, since only those DRAMs will be written into. DRAMs not accessed in this write will output the current contents of their memory at that location, since the $\overline{OE}$ of the DRAMs is asserted. $\overline{DRAM\_WR}$ controls the $\overline{LEAB}$, leaving the latch transparent throughout the write. $\overline{WB}(3:0)$ controls the $\overline{OEAB}$ of the latches, thus enabling only those bytes that are written.

In a processor read access, the DRAM system drives the data bus. The DRAM system is synchronized to the rising edge of $\overline{BSysClk}$, and the R3051 samples the input data on the falling edge of $\overline{SysClk}$ before terminating the access. Thus, the DRAM control design, which drives the $\overline{RAS}$ and $\overline{CAS}$ signals on the rising edge of $\overline{SysClk}$, actually removes $\overline{CAS}$ one-half cycle before the data is sampled by the CPU. Thus, data output by the DRAMs is actually latched by the transceivers, and remains valid when the CPU samples the A/D bus one-half clock cycle later.

The DRAM_LE from the DRAM controller is connected to the $\overline{LEBA}$ pin, which latches the data into the transceivers. The $T/\overline{R}$ signal connected to the $\overline{CEBA}$ pin, which controls the direction of the bi-directional transceiver. The $\overline{DataEn}$ signal from the R3051 is connected directly to the $\overline{OEBA}$ pin to control the timing of the output enable onto the A/D bus. This ensures that the output buffers are tri-stated before the next R3051 access starts and prevents any bus contention.

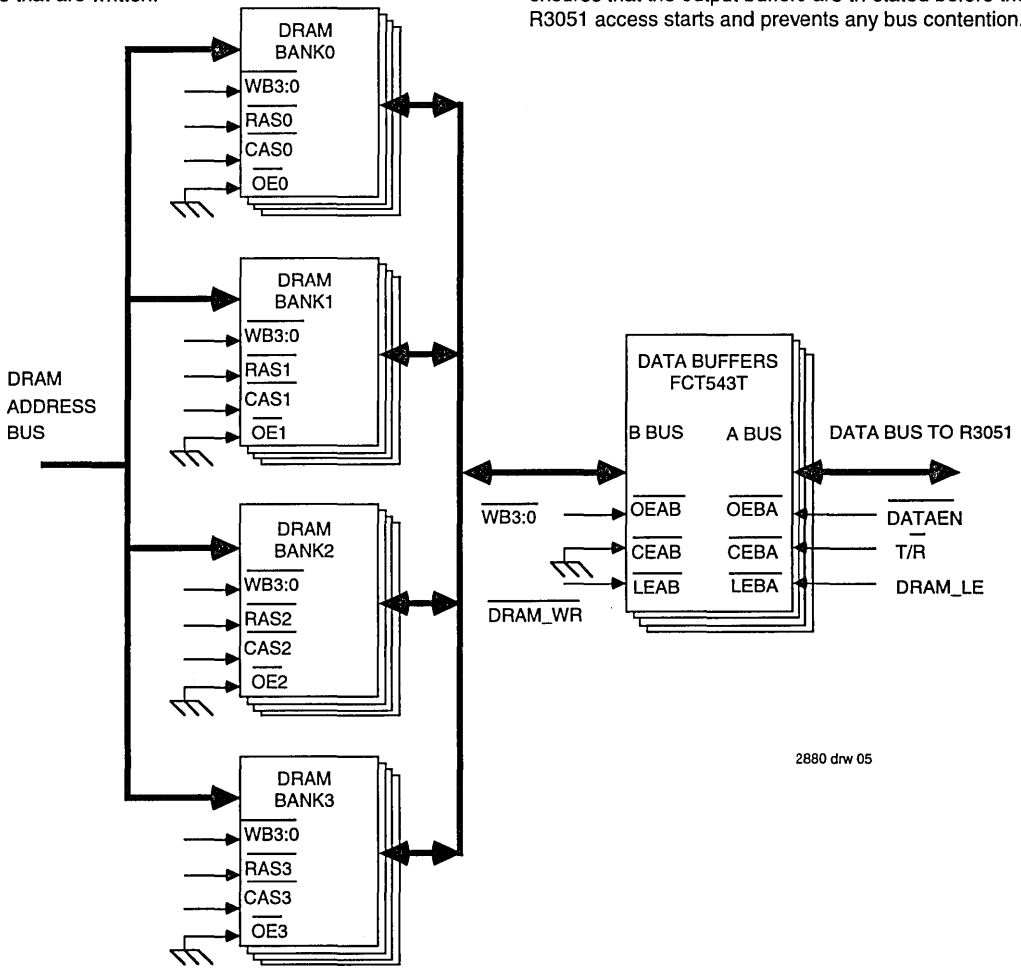


Figure 5. DRAM Banks and Data Buffers

THE DRAM MEMORY SYSTEM TIMING

The R3051 system interface allows this DRAM interface to be simply constructed. Features of the R3051 which are used in this DRAM system include:

- On-chip four-deep read and a four-deep write buffers. These buffers decouple the system interface speed from the speed of the execution engine on-chip.
- Single word reads and four-word refills. Block refills amortize the relatively long latency of DRAMs over multiple words, taking advantage of high-bandwidth capabilities (e.g. Page Mode) offered by DRAMs.
- The $\overline{\text{WrNear}}$ signal, which informs the external DRAM sub-system that two consecutive writes have the same upper 22 address bits (equivalent to a local page of 256 words), and can be written using a Page Mode access.

For the system running at 25 Mhz, the clock period is 40 nsec. DRAMs with 80 nsec of access time require 160 nsec (t_{RC} =160 nsec) to complete one read access (as per DRAM data sheet). A 5 clock cycles (200 nsec) read access time allows an acceptable margin for address decoding, control signal propagation, and bus interface.

For a 4 word block refill read, the initial latency (time to read the first word) is the same as for a single word read access (200 nsec). For the next 3 consecutive words, the DRAM memory system provides a word every 2 clock cycles (every 80 nsec). A block refill access can be completed in 11 clock cycles (440 nsec), which is an average of 110 nsec per word. Thus, block refill, with this simple scheme, provides a significant improvement in the average access time per word (over 2 clock cycles per word savings).

The state machine to manage write operations takes advantage of two features of the R3051:

- On a write cycle, the write data from the processor is held one full clock cycle after the clock edge where the processor samples its ACK input. Thus, the DRAM system can give an early acknowledge, and still rely on the CPU to continue driving data.
- The $\overline{\text{WrNear}}$ output from the CPU, which indicates that this write may be retired using a page mode write. This reduces the number of cycles required to perform write-intensive operations, such as building the program stack or flushing the write buffer.

The state machine for single word writes is optimized to allow subsequent near writes to be retired using page mode accesses. The DRAM memory system takes advantage of the $\overline{\text{WrNear}}$ signal from the R3051 by defaulting to the case that any single write to the DRAM system will be followed by another write with the same upper 22 address bits (within the local page of 256 words). Given this assumption, the $\overline{\text{RAS}}$ signals must be kept asserted after every write access to remain in the page mode of the DRAMs.

Thus, an initial single write can be performed in 4 clock cycles (160 nsec) since the $\overline{\text{RAS}}$ signals are not de-asserted and the $\overline{\text{RAS}}$ precharge time (t_{RP} = 70 nsec) will be deferred until the end of the page write mode. Note that this is faster than a single read; the state machine takes advantage of the fact that the processor will drive data a full clock cycle after acknowledge is given.

A consecutive write to the same DRAM page can be performed in 3 clock cycles (120 nsec) since the $\overline{\text{RAS}}$ signal is already asserted and doesn't need to be precharged. When this state is exited (when a write outside of page or a different type of access occurs) the $\overline{\text{RAS}}$ signal needs to be precharged for 2 clock cycles (80 nsec) before responding to the pending access.

Single Write Cycle/Page Write Cycle

Figure 6 illustrates the timing diagrams for a single write access followed by a page write. The R3051 initiates a single DRAM write access by the assertion of $\overline{\text{Wr}}$ and with A22 low. Since the state machine is in the IDLE state, $\overline{\text{RAS}}$ is deasserted and the ROW addresses are flowing through the address multiplexer. The $\overline{\text{CIP}}$ is issued on the next clock edge to inform the rest of the machine that the write is being processed, thus preventing the commitment of any other state (e.g. refresh). The appropriate $\overline{\text{RAS}}$ signal is issued on the same edge as the $\overline{\text{CIP}}$. The $\overline{\text{DRAM\_ACK}}$ is issued on the following edge and the $\overline{\text{CAS}}$ signal on the 4th edge to terminate the write access. At the end of the access, the $\overline{\text{CIP}}$ is removed while the $\overline{\text{RAS}}$ signal is kept asserted in anticipation of a consecutive write access within the same page. At the end of an initial write access, the $\overline{\text{DRAM\_WN}}$ signal remains asserted. This signal informs the rest of the state machine that the $\overline{\text{RAS}}$ signals are kept asserted.

Idle, $\overline{\text{RAS}}$ Asserted State

At the end of a write access the state machine enters this state where a $\overline{\text{RAS}}$ signal is kept asserted while the state machine awaits a subsequent transaction. If the next access is a local write ($\overline{\text{WrNear}}$ from the R3051 is asserted) the state machine enters the page write mode. If a different access type occurs (read, block refill, not local write) or a refresh is pending, the state machine exits this state.

Upon exiting this state, the machine precharges the $\overline{\text{RAS}}$ signal before responding to the pending access. For the ease of discussion, any access that requires the $\overline{\text{RAS}}$ signals to be precharged before the access is processed will be referred to as "delayed" access. If an access outside the DRAM space is detected ($\overline{\text{Wr}}$ or $\overline{\text{Rd}}$ asserted while A22=1) the $\overline{\text{RAS}}$ signals are immediately de-asserted and the machine goes into the IDLE state. This is an important condition; an intervening write to another memory location causes the R3051 to report subsequent writes as "near" to that other memory location, and thus the DRAM controller should not process these writes as near writes.

Page Write Cycle

A page write cycle is a write access to the DRAM following another write with the same upper 22 address bits. Figure 6 illustrates the timing diagram for a page write access. The R3051 initiates a page write cycle by the assertion of $\overline{\text{Wr}}$, $\overline{\text{WrNear}}$ and A22 = 0. On the following clock edge $\overline{\text{CIP}}$ and $\overline{\text{DRAM\_ACK}}$ are issued, and on the 3rd clock edge $\overline{\text{CAS}}$ is asserted. and the access is terminated ($\overline{\text{CIP}}$ is negated). The $\overline{\text{RAS}}$ and $\overline{\text{DRAM\_WN}}$ signals are kept asserted, allowing

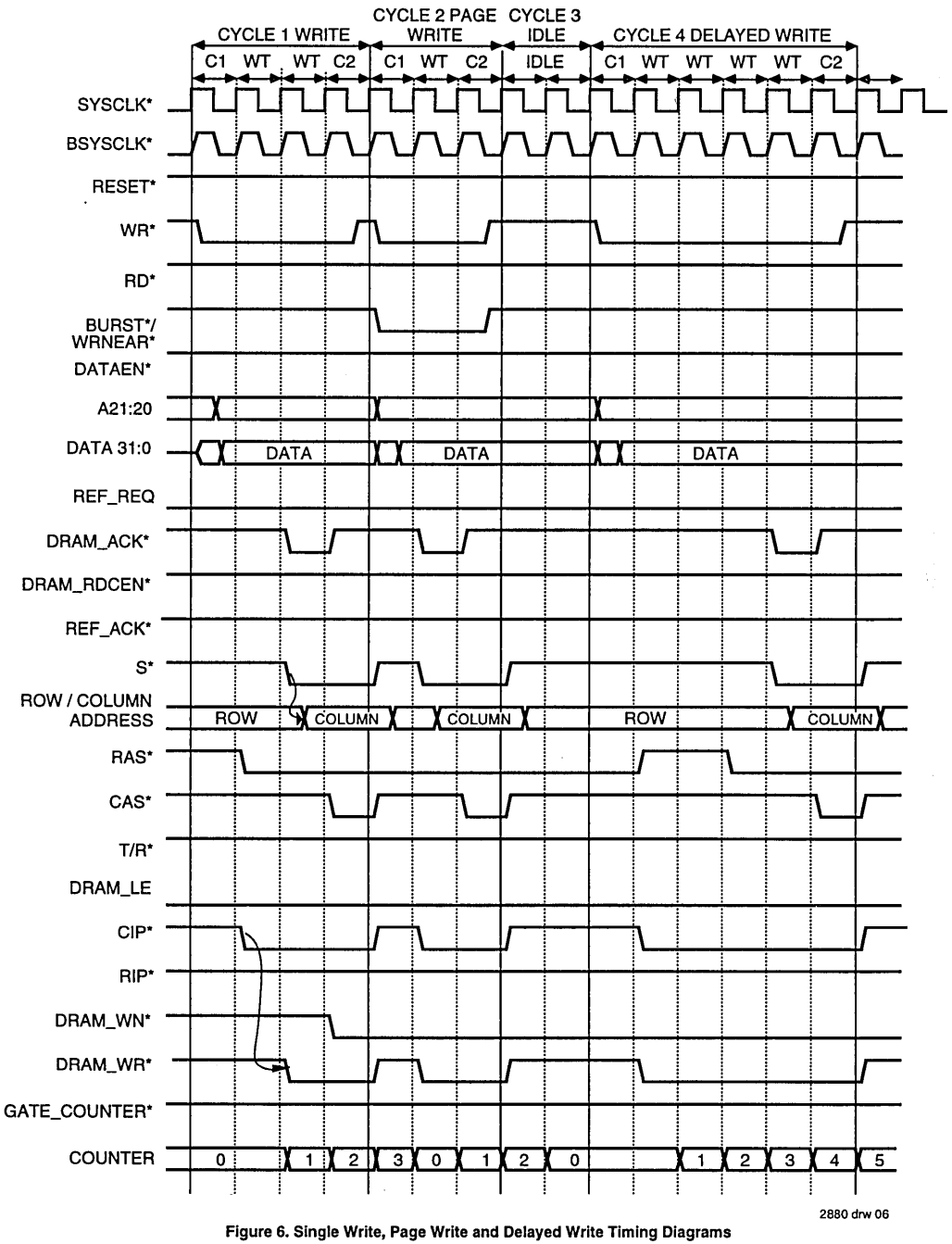


Figure 6. Single Write, Page Write and Delayed Write Timing Diagrams

subsequent page writes to be rapidly processed. The state machine exits this state into the IDLE RAS ASSERTED state to await subsequent page mode writes.

Delayed Write Cycle

The delayed write cycle has exactly the same sequence as a single write but is delayed by two clock cycles. A delayed write is a "non-near" write detected in the IDLE RAS ASSERTED state. Figure 6 illustrates the timing diagrams for a delayed write access.

The R3051 initiates a delayed write access by the assertion of Wr and A22 = 0 while RAS and DRAM_WN are asserted. On the next clock edge RAS is de-asserted while the DRAM_WN is kept asserted. The precharging of the RAS signal takes two clock cycles. The DRAM_WN signal is kept asserted to inform the state machine that the control signals for this access have to be delayed by two clock cycles. This is true for all the delayed accesses.

Single Read Cycle

A single read cycle is a read access to the DRAM following an IDLE state in which the RAS and the DRAM_WN are not asserted. Figure 7 illustrates the timing diagrams for a read access. The R3051 initiates a single read access by the assertion of Rd with A22 low while the state machine is IDLE and all RAS outputs are de-asserted. The CIP is issued on the next clock edge to inform the rest of the machine that a cycle is ongoing, thus preventing the commitment of any other state. The appropriate RAS signal is issued on the same edge as the CIP. Two clock cycles later, the CAS, DRAM_RDCEN and the DRAM_ACK are issued to terminate the cycle.

For a read access both the DRAM_ACK and the DRAM_RDCEN are required to end the cycle. The processor will not actually sample RdCEn until one-clock after the clock edge used to generate DRAM_RDCEN, and thus will not sample the data until one and one-half clock cycles after the edge used to generate DRAM_RDCEN. From the timing diagrams it is clear that the CAS and the RAS signals are removed half a clock cycle before the falling edge of the clock when the R3051 samples the data. DRAM_LE latches the DRAM data into the transceivers and holds it for one clock cycle. At the end of the access the CIP is removed.

Delayed Read Cycle

The timings of a delayed read are exactly the same as for a single read but shifted by two clock cycles to accommodate RAS pre-charge time. A delayed read cycle is a read access to the DRAM following an IDLE RAS ASSERTED state in which the RAS and the DRAM_WN are still asserted. Figure 8 illustrates the timing diagrams for a delayed read access. Once a read access is detected, the RAS signal is de-asserted while the DRAM_WN is kept asserted. The RAS signal is precharged for two clock cycles. At the end of a delayed read, the DRAM_WN and the CIP are removed and the machine enters the IDLE state.

Block Refill Cycle

A block refill cycle is a 4 word read access to the DRAM following an IDLE state. Figure 7 illustrates the timing diagrams for 4-word block refill access. The R3051 indicates a block refill read access by the assertion of Rd and Burst with A22 low. The DRAM control sub-system handles block refill accesses using the Throttled Block Refill mode of the R3051. In a throttled read, RdCEn is used to control the data rate of memory back to the CPU. The Ack input is not provided back to the processor until the transfer has sufficiently progressed such that the last word of the transfer is clocked into the on-chip read buffer before the processor core requires it.

In the block refill access the first word read takes the same time as a single read while the 3 subsequent words are read into the read buffer at the rate of 1 word every two clock cycles. The DRAM_RDCEN is issued with every word being read to cause the R3051 to latch the data into the read buffer. The DRAM_ACK is issued between the second and the third word read. This ensures that for 4 subsequent falling edges of SysClk the read buffer can provide data to the R3000A core at the rate of a word every clock cycle.

Block refill uses the page mode characteristics of the DRAM to obtain subsequent words at a high data rate. In this access, the RAS signal is kept asserted while the CAS signal is toggled 4 times to produce 4 data words. Every word from the DRAM system is latched into the transceivers as for a single read operation, using the DRAM_LE to clock the latched transceivers. At the end of the access RAS and CIP are de-asserted, and the state machine returns to the IDLE state.

In the block refill access, address lines Addr(3:2) from the R3051 act as a two-bit counter to provide the address of 4 consecutive words. These two lines are incremented on the falling edge of SysClk. This timing could prove critical at high-frequencies: this is only half a clock margin (20 nsec) before the CAS signals are asserted, in which address set-up time to CAS must be provided. These two lines are part of the address path and are driving large capacitive loads. Two minimize additional delay due to loading, two sets or more of memory address drivers could then be used to minimize the effect of the capacitive loads and to ensure proper operation.

Delayed Block Refill Cycle

A delayed block refill cycle is a block refill access to the DRAM following an IDLE RAS ASSERTED state in which the RAS and the DRAM_WN are asserted. Figure 9 illustrates the timing diagrams for a delayed block refill access. A delayed block refill is exactly the same as a block refill with the exception that the access is shifted by two clock cycles to accommodate RAS precharge requirements. The DRAM_WN signals to the machine that the access has a delayed timing. At the end of the access, the DRAM_WN and the CIP are de-asserted.

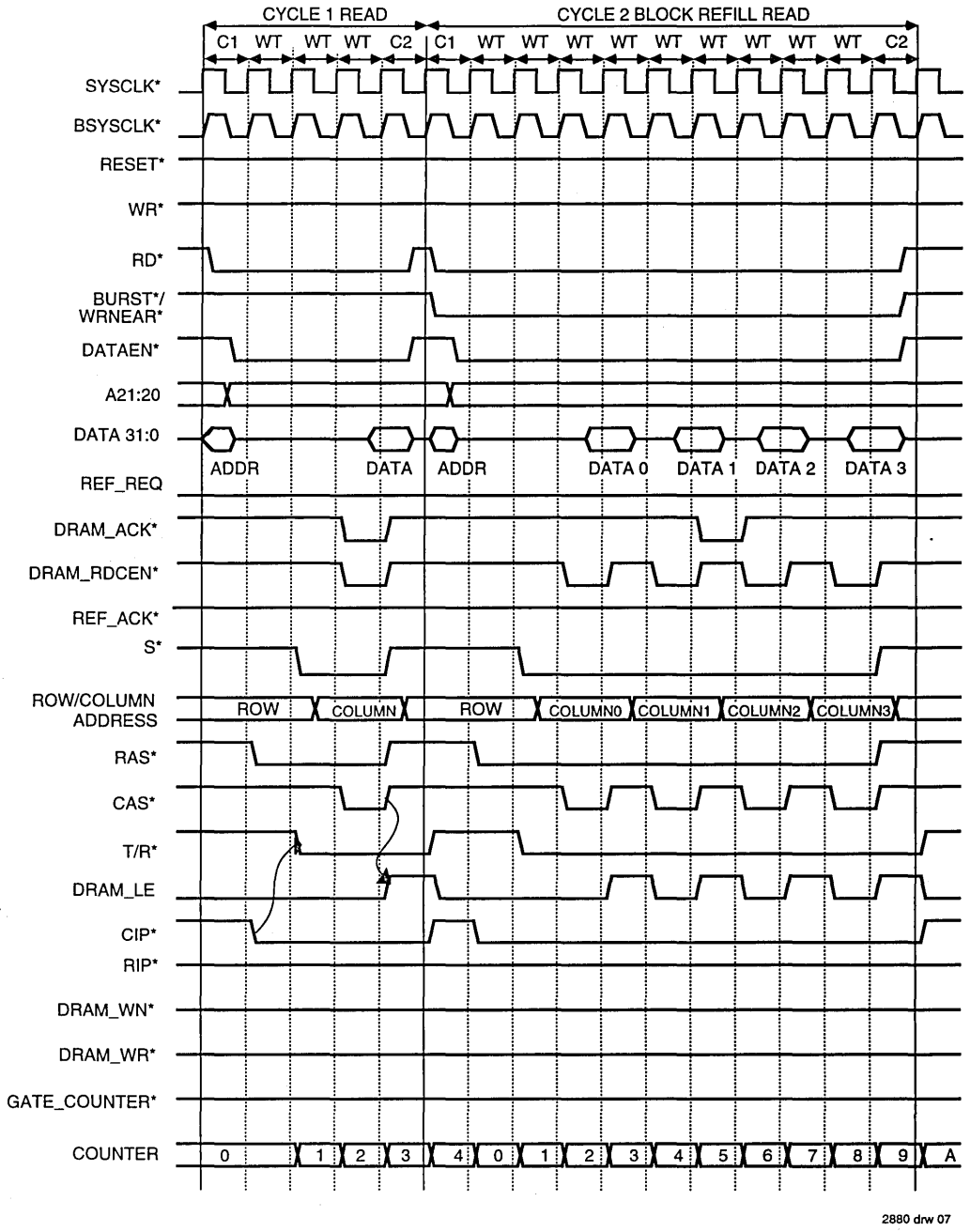


Figure 7. Single Read and Block Refill Read Timing Diagrams

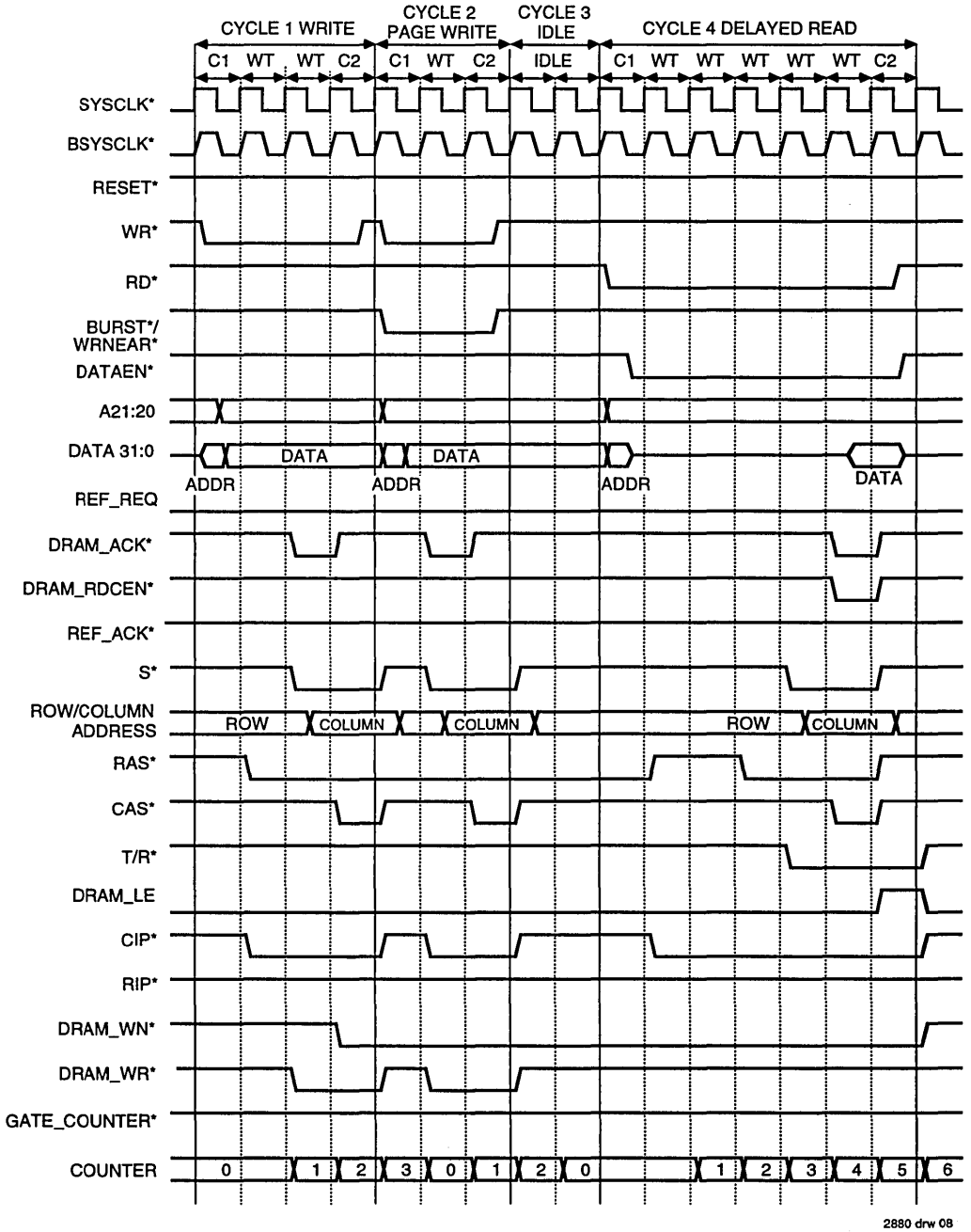


Figure 8. Delayed Single Read Timing Diagrams

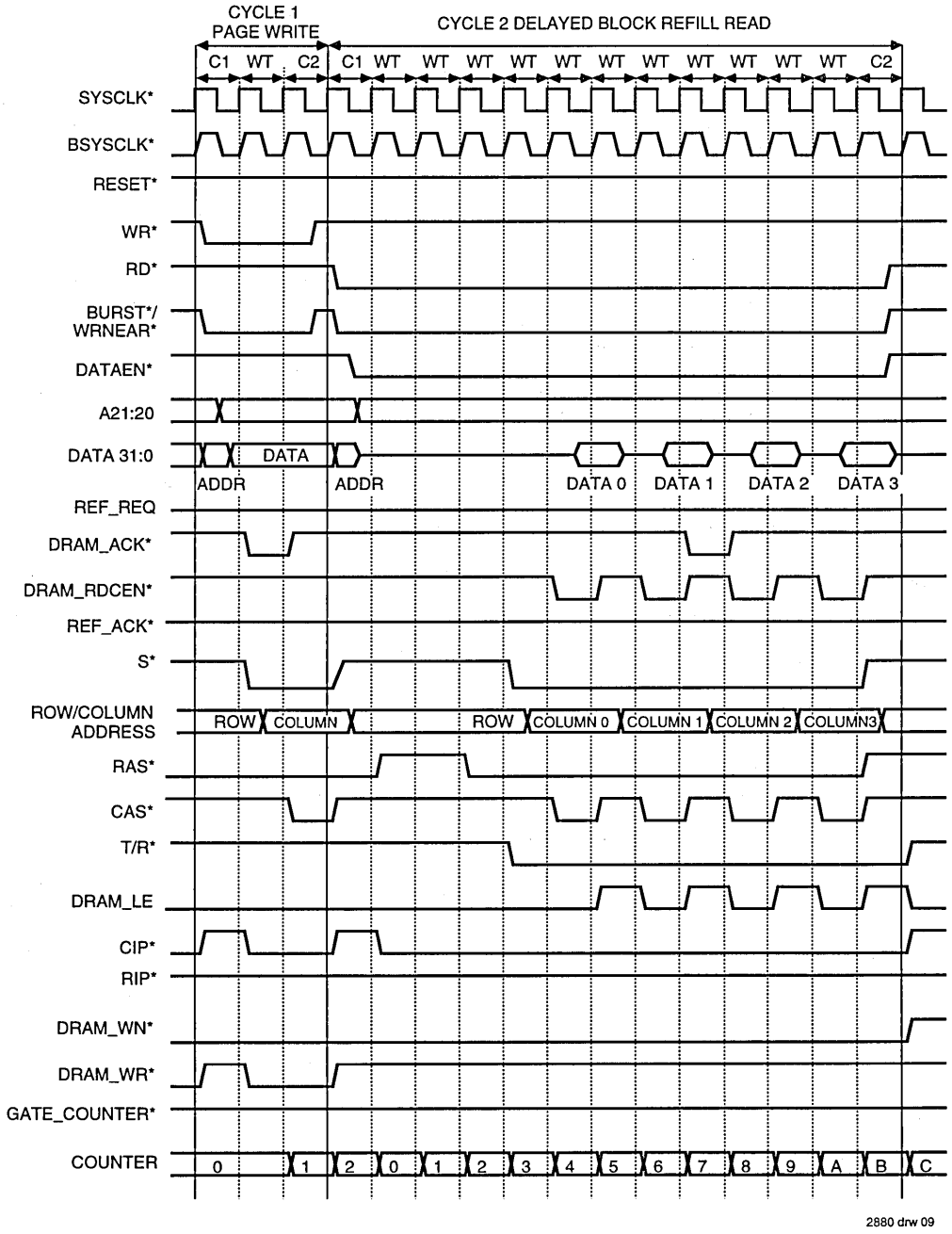


Figure 9. Delayed Block Refill Read Timing Diagrams

Refresh Cycle

A refresh cycle is initiated every time a REF_REQ pulse is detected. The state machine responds immediately by asserting the $\overline{\text{REF\_ACK}}$ signal on the following clock edge. This disables the refresh timer until the refresh access is completed. Figure 10 illustrates the timing diagrams for a refresh arbitration and the actual refresh access.

If a REF_REQ occurs during an access or at the same time as an access, the refresh is delayed until the access is terminated (signaled by $\overline{\text{CIP}}$ de-asserted). Asserting $\overline{\text{REF\_ACK}}$ at the detection of REF_REQ ensures that the following access will be a refresh access and prevents the commitment of any other state. Delaying a refresh request until the end of a bus access doesn't affect the DRAM operation, since the refresh period selected is much less than the maximum refresh period of a DRAM row. The refresh period is every 9.6 µsec and the longest access is the delayed block refill with 14 clock cycles (until $\overline{\text{CIP}}$ is removed) which is 0.56 µsec. Thus, the refresh will be serviced at a maximum of 10.16 µsec, which is substantially below the maximum 15.5 µsec refresh requirement of the DRAMs. By the same reasoning, if the granted access is a delayed access, the $\overline{\text{RAS}}$ signal will be precharged prior to the 10 µsec $\overline{\text{RAS}}$ pulse width maximum requirements. If a Page Mode Write is granted, it will be retired in 3 cycles, or .12µsec, and thus $\overline{\text{RAS}}$ will be precharged for the refresh no longer than 9.72µsec after it was asserted.

The refresh access is a $\overline{\text{CAS}}$-before-$\overline{\text{RAS}}$ refresh in which all four $\overline{\text{CAS}}$ and $\overline{\text{RAS}}$ signals are issued. The $\overline{\text{CAS}}$ signal is issued 1 clock cycle before the $\overline{\text{RAS}}$ signal. A refresh access takes 10 clock cycles. This time is long enough to allow the $\overline{\text{RAS}}$ signals to be precharged if needed (delayed refresh). A delayed refresh has then the same timing as a refresh access.

Figure 11 shows the timing diagrams for the delayed refresh cycles. $\overline{\text{GATE\_COUNTER}}$ controls the operation of the 4-bit counter when transitioning between bus accesses and refresh accesses. It is mainly used in the arbitration phase when a bus access and refresh access are requested at the same time.

Reset Cycle

A reset cycle is initiated by the assertion of the $\overline{\text{Reset}}$ signal. This is a hardware reset and is used to initialize the PALs to the correct IDLE state. The $\overline{\text{RIP}}$ signal is asserted on the following clock edge to inform the machine that a reset cycle is in progress. After the $\overline{\text{Reset}}$ signal is de-asserted, the $\overline{\text{RIP}}$ stays asserted and one refresh access is initiated. At the end of this refresh access, the $\overline{\text{RIP}}$ is removed and the state machine enters the IDLE state. Figure 12 illustrates the timing diagrams of the reset operation.

Most DRAMs require at least 8 $\overline{\text{CAS}}$ before $\overline{\text{RAS}}$ refresh accesses prior to a regular access, to insure proper initialization. The actual state machine provides only one refresh access. It is the responsibility of the software to ensure that no DRAM access is made prior to the elapsing of 8 refresh periods from the refresh timer. This can typically be insured by normal operation of the boot PROM; however, software could "spin-lock" for a pre-determined number of loops to insure that sufficient time has elapsed.

Idle State

The IDLE state is the state in which the machine is not performing any bus access or a refresh access but is constantly monitoring the bus for any access request. All the signals are de-asserted and the 4-bit counter operation is halted.

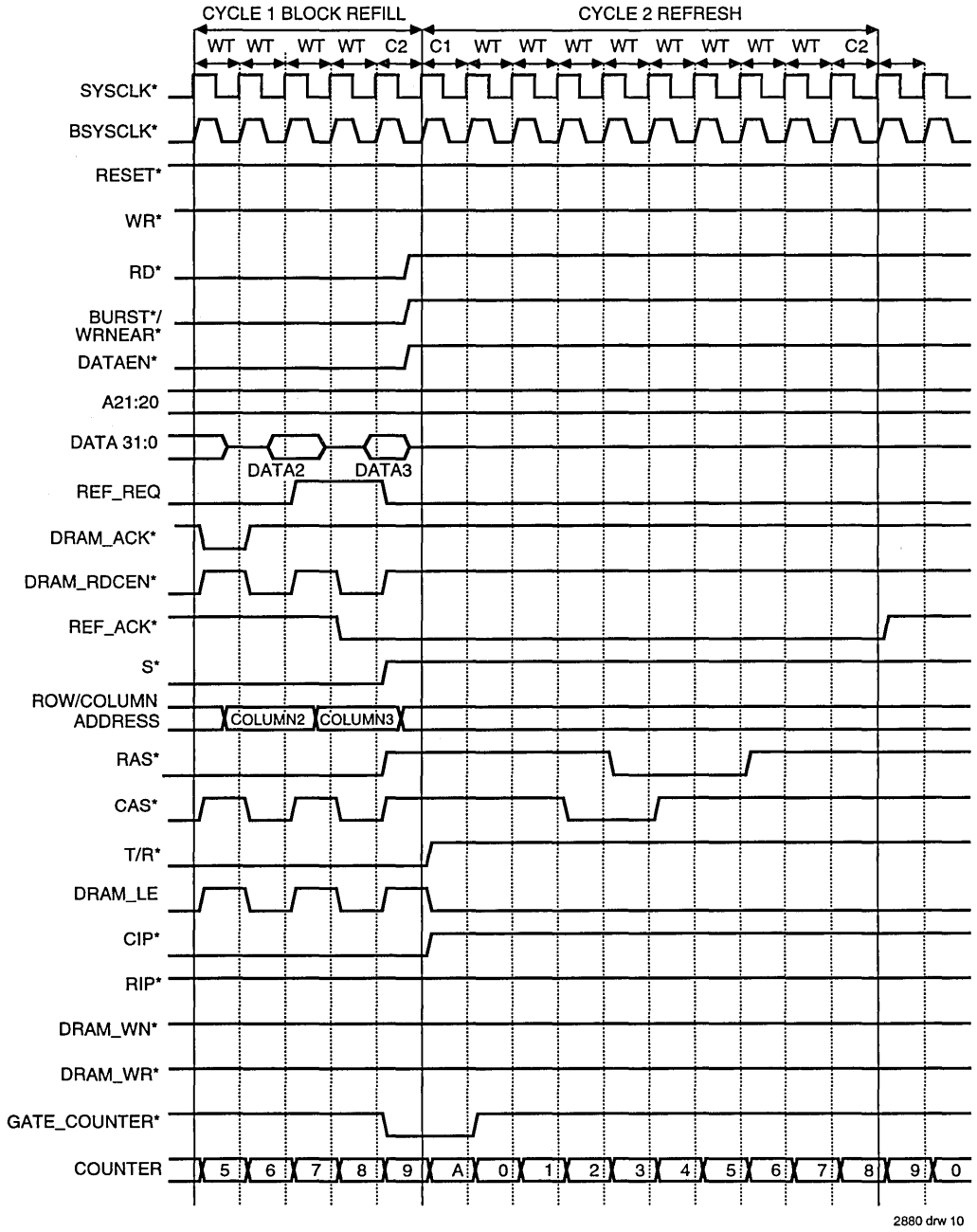


Figure 10. Refresh Arbitration and Refresh Timing Diagrams

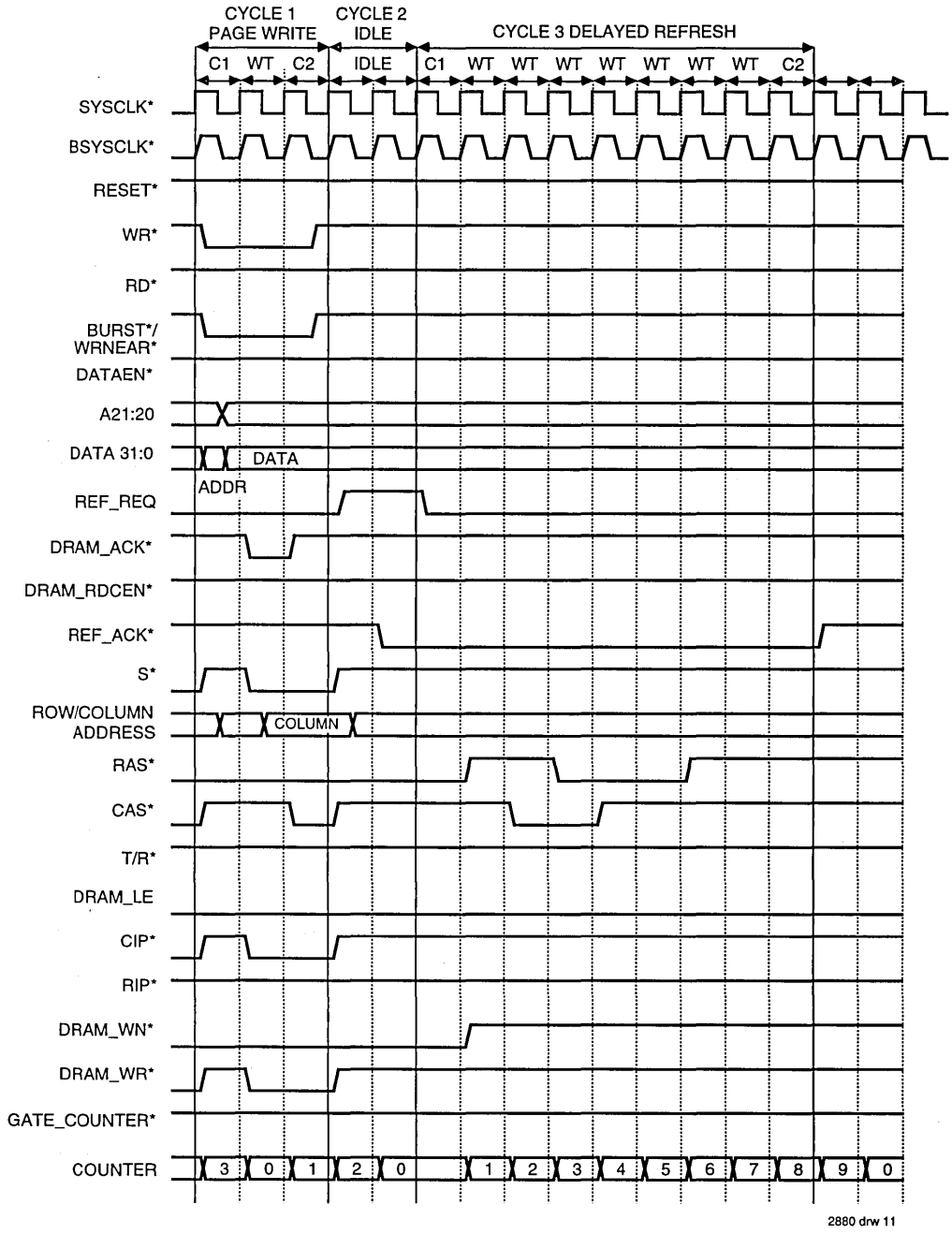


Figure 11. Delayed Refresh Timing Diagrams

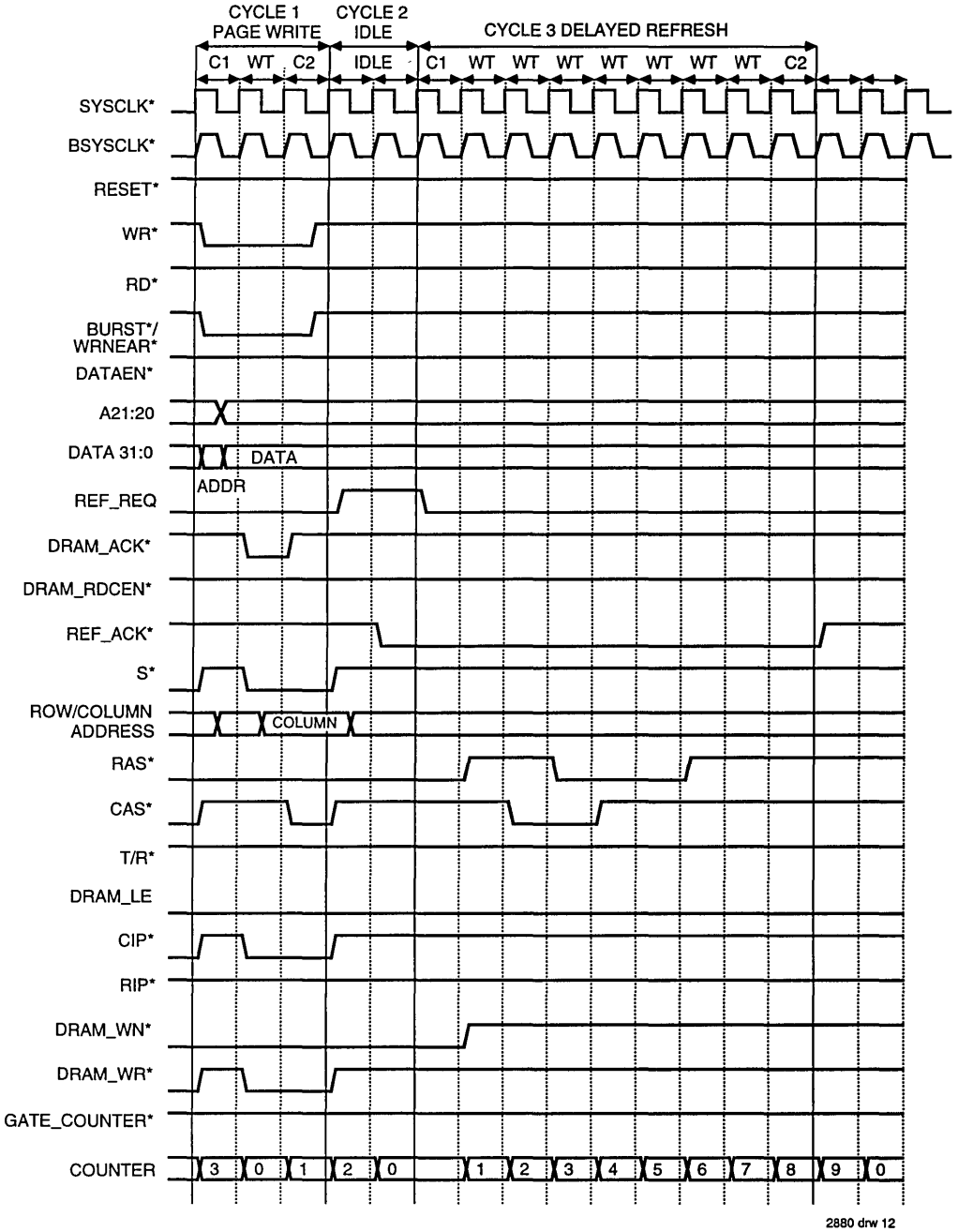


Figure 12. Reset Timing Diagrams

CRITICAL TIMING CALCULATIONS

The following is a timing analysis of some of the critical paths in the DRAM system.

DRAM Data for a Read or block Refill access

As illustrated in all the timing diagrams, the $\overline{CAS}$ signal is asserted for only 1 clock cycle for a read or a write access. For a write access there is no critical timing since the DRAM latches the data in at the $\overline{CAS}$ leading edge, and the processor insures sufficient data hold time by holding data for one cycle after $\overline{ACK}$ is detected.

For a read or a block refill access the DRAMs provide the data to the R3051 and the maximum delays must be considered. Figure 13 illustrates the detailed timing for a portion of a block refill access which is also true for a read access. The R3051 uses the $\overline{SysClk}$ for its reference with a period Tclk of 40 nsec. The $\overline{CAS}$ and the DRAM_LE signals are delayed with respect to $\overline{SysClk}$ by the Pal 2 propagation delay T1. The data is available from the DRAM after T2 (t_{cac} = 25 nsec max). The critical path requires that the DRAM data be available and meet the setup time of the transceivers before the DRAM_LE is asserted. The timing calculation for this data path is as follows:

Tclk	= 40.0 nsec	
- T1 max	= 8.0	
	= 32.0	
- T2 max	= 25.0	
	= 7.0	
- T setup	= 3.0	FCT543T data setup time.
	= 4.0	

The available margin is 4.0 nsec. Some 80 nsec DRAMs have T2 (t_{cac} = 20 nsec) which could offer more margin.

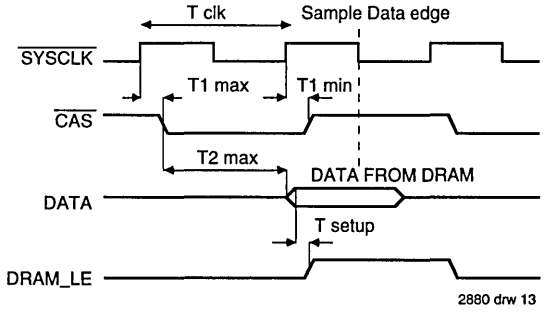


Figure 13. Read or Block Refill Access

Transceivers Turn Off time

For a read or a block refill access, the DRAMs provide the data to the R3051 through the latched transceivers. As illustrated in Figure 7, the R3051 reads the data from the bus half a clock cycle before it starts a new access in which it can drive address on the bus. This information is explained in detail in the R3051 User Manual.

The critical path requires that the transceivers be tri-stated before the R3051 starts driving the bus in the next clock cycle. The $\overline{DataEn}$ signal directly from the R3051 enables the B to A output buffers of the transceivers (FCT543T). The $\overline{DataEn}$ is delayed by T3 from the falling edge of $\overline{SysClk}$ at which the R3051 samples the data (as per R3051 data sheet). The transceivers disable the output buffers within T4. Figure 14 illustrates the timing for this path.

Tclk / 2	= 20.0 nsec
- T3 max	= 6.0
	= 14.0
- T4 max	= 9.0
T margin	= 5.0 nsec

This margin of 5 nsec is long enough to accommodate for any $\overline{SysClk}$ skews.

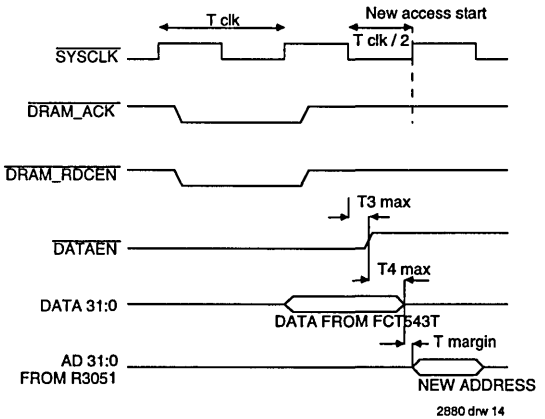


Figure 14. Termination of a Read or Block Refill Access

$\overline{\text{DRAM\_ACK}}$ and $\overline{\text{DRAM\_RDCEN}}$ Timings

The $\overline{\text{DRAM\_ACK}}$ and the $\overline{\text{DRAM\_RDCEN}}$ are issued for one clock cycle only as illustrated in the timing diagrams. They are removed by the clock edge which the R3051 uses to sample them. The R3051 requires that these two signals be held constant for a minimum of 4 nsec after the clock edge. These two signals are usually combined with similar signals from other memory sub-systems (e.g. EPROM) to form one set that is routed to the R3051. This extra delay, plus the Pal 1 minimum propagation delay are long enough to meet the R3051 required hold time.

PERFORMANCE

The performance of the different types of R3051 bus accesses to the DRAM memory is usually measured by the number of clock cycles it takes to send the $\overline{\text{Ack}}$ back to the R3051. This time is computed from the beginning of the external access. The performance of the DRAM system can be summarized as follows:

- single read: 4 clock cycles.
- block refill: 7 clock cycles.
- first write: 3 clock cycles.
- page write: 2 clock cycles.

The above numbers (with the exception of page write) will be increased by 2 in the case of delayed accesses.

Thus, relatively high memory performance is obtained with minimal external logic parts count, and low-cost commodity DRAM. More aggressive designs could utilize faster DRAMs, and techniques such as memory interleaving, to achieve still higher levels of performance.

CONCLUSION

The R3051 RISController family bus interface was designed to allow memory systems of differing complexity and performance to be implemented. Even a relatively simple DRAM system, as the one described here, offers very high performance. With simple modifications, this approach is applicable to higher frequencies (33 and 40 Mhz) and to interleaved memory systems yielding even higher performance.

```
{
      TITLE:              PAL1
      PURPOSE:            RAS
      AUTHOR:             BOB NAPAA, IDT INC.
      DATE:               4/5/91
}

MODULE PAL1;
TITLE PAL1;
TYPE AMD 22V10;

INPUTS;

      SYSCLKB                   NODE[PIN1];
      ENABLEB                   NODE[PIN2];
      RDB                       NODE[PIN3];
      WRB                       NODE[PIN4];
      BURSTB                    NODE[PIN5];
      RIPB                      NODE[PIN6];
      REFACKB                   NODE[PIN7];
      A22                       NODE[PIN8];
      A21                       NODE[PIN9];
      A20                       NODE[PIN10];
      C3                        NODE[PIN11];
      C2                        NODE[PIN13];
      C1                        NODE[PIN14];
      C0                        NODE[PIN15];

      {FEED BACK PINS}
      CIPB                      NODE[PIN16];
      RAS3B                     NODE[PIN17];
      RAS2B                     NODE[PIN18];
      RAS1B                     NODE[PIN19];
      RAS0B                     NODE[PIN20];
      DRAMWNB                   NODE[PIN21];
      DRAMACKB                  NODE[PIN22];
      DRAMRDCENB                NODE[PIN23];


OUTPUTS;
      CIPB                      NODE[PIN16]  ATTR[RL];
      RAS3B                     NODE[PIN17]  ATTR[RL];
      RAS2B                     NODE[PIN18]  ATTR[RL];
      RAS1B                     NODE[PIN19]  ATTR[RL];
      RAS0B                     NODE[PIN20]  ATTR[RL];
      DRAMWNB                   NODE[PIN21]  ATTR[RL];
      DRAMACKB                  NODE[PIN22]  ATTR[RL];
      DRAMRDCENB                NODE[PIN23]  ATTR[RL];

      {OUTPUT ENABLES}
      CIPBEN                    NODE[PIN16EN];
      RAS3BEN                   NODE[PIN17EN];
      RAS2BEN                   NODE[PIN18EN];
      RAS1BEN                   NODE[PIN19EN];
      RAS0BEN                   NODE[PIN20EN];
      DRAMWNBEN                 NODE[PIN21EN];
      DRAMACKBNODE[PIN22EN];
      DRAMRDCENBEN              NODE[PIN23EN];
```

```
TERMS;

        RAS3BEN          =     ENABLEB;
        RAS3B  NOT       :=    RAS3B AND REFACKB AND RIPB AND DRAMWNB AND !RDB AND
                               !A22 AND A21 AND A20 {read/block refill}
                         OR    !RAS3B AND !CIPB AND !RDB AND DRAMACKB AND DRAMRDCENB
                               {keep for read/delayed read}
                         OR    RAS3B AND !CIPB AND RIPB AND !DRAMWNB AND !RDB AND
                               !A22 AND A21 AND A20 AND !C3 AND !C2 AND !C1 AND C0
                               {delayed read/delayed block refill}
                         OR    !RAS3B AND !CIPB AND !RDB AND !BURSTB AND !C3 {keep block refill}
                         OR    !RAS3B AND !CIPB AND !RDB AND !BURSTB AND !DRAMWNB AND
                               !C1 {keep delayed block refill}
                         OR    RAS3B AND REFACKB AND RIPB AND DRAMWNB AND !WRB AND
                               !A22 AND A21 AND A20 {write}
                         OR    RAS3B AND REFACKB AND RIPB AND !DRAMWNB AND !WRB AND
                               !A22 AND A21 AND A20 AND !C3 AND !C2 AND !C1 AND C0
                               {delayed write}
                         OR    !RAS3B AND !WRB AND !CIPB {keep for write}
                         OR    !RAS3B AND !DRAMWNB AND REFACKB AND RIPB AND RDB AND
                               WRB AND BURSTB {no access pending}
                         OR    !RAS3B AND !DRAMWNB AND REFACKB AND RIPB AND !WRB AND
                               !BURSTB AND !A22 AND A21 AND A20 {keep for page write}
                         OR    !REFACKB AND CIPB AND !RAS3B AND !DRAMWNB AND C0
                               {remove in refresh}
                         OR    RAS3B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND !C2
                               AND C1 AND C0 {issue for refresh}
                         OR    !RAS3B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                               AND !C1 AND !C0 {keep for refresh}
                         OR    !RAS3B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                               AND !C1 AND C0; {keep for refresh}

        RAS2BEN          =     ENABLEB;
        RAS2B  NOT       :=    RAS2B AND REFACKB AND RIPB AND DRAMWNB AND !RDB AND
                               !A22 AND A21 AND !A20 {read/block refill}
                         OR    !RAS2B AND !CIPB AND !RDB AND DRAMACKB AND DRAMRDCENB
                               {keep for read/delayed read}
                         OR    RAS2B AND !CIPB  AND RIPB AND !DRAMWNB AND !RDB AND
                               !A22 AND A21 AND !A20 AND !C3 AND !C2 AND !C1 AND C0
                               {delayed read/delayed block refill}
                         OR    !RAS2B AND !CIPB AND !RDB AND !BURSTB AND !C3 {keep block refill}
                         OR    !RAS2B AND !CIPB AND !RDB AND !BURSTB AND !DRAMWNB AND
                               !C1{keep delayed block refill}
                         OR    RAS2B AND REFACKB AND RIPB AND DRAMWNB AND !WRB AND
                               !A22 AND A21 AND !A20 {write}
                         OR    RAS2B AND REFACKB AND RIPB AND !DRAMWNB AND !WRB AND
                               !A22 AND A21 AND !A20 AND !C3 AND !C2 AND !C1 AND C0
                               {delayed write}
                         OR    !RAS2B AND !WRB AND !CIPB {keep for write}
                         OR    !RAS2B AND !DRAMWNB AND REFACKB AND RIPB AND RDB AND
                               WRB AND BURSTB {no access pending}
                         OR    !RAS2B AND !DRAMWNB AND REFACKB AND RIPB AND !WRB AND
                               !BURSTB AND !A22 AND A21 AND !A20 {keep for page write}
                         OR    !REFACKB AND CIPB AND !RAS2B AND !DRAMWNB AND C0
                               {remove in refresh}
                         OR    RAS2B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND !C2
                               AND C1 AND C0 {issue for refresh}
                         OR    !RAS2B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                               AND !C1 AND !C0 {keep for refresh}
                         OR    !RAS2B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                               AND !C1 AND C0; {keep for refresh}
```

```
RAS1BEN          =     ENABLEB;
RAS1B NOT        :=    RAS1B AND REFACKB AND RIPB AND DRAMWNB AND !RDB AND
                       !A22 AND !A21 AND A20 {read/block refill}
                 OR    !RAS1B AND !CIPB AND !RDB AND DRAMACKB AND DRAMRDCENB
                       {keep for read/delayed read}
                 OR    RAS1B AND !CIPB AND RIPB AND !DRAMWNB AND !RDB AND
                       !A22 AND !A21 AND A20 AND !C3 AND !C2 AND !C1 AND C0
                       {delayed read/delayed block refill}
                 OR    !RAS1B AND !CIPB AND !RDB AND !BURSTB AND !C3 {keep block refill}
                 OR    !RAS1B AND !CIPB AND !RDB AND !BURSTB AND !DRAMWNB AND
                       !C1{keep delayed block refill}
                 OR    RAS1B AND REFACKB AND RIPB AND DRAMWNB AND !WRB AND
                       !A22 AND !A21 AND A20 {write}
                 OR    RAS1B AND REFACKB AND RIPB AND !DRAMWNB AND !WRB AND
                       !A22 AND !A21 AND A20 AND !C3 AND !C2 AND !C1 AND C0
                         {delayed write}
                 OR    !RAS1B AND !WRB AND !CIPB {keep for write}
                 OR    !RAS1B AND !DRAMWNB AND REFACKB AND RIPB AND RDB AND
                       WRB AND BURSTB {no access pending}
                 OR    !RAS1B AND !DRAMWNB AND REFACKB AND RIPB AND !WRB AND
                       !BURSTB AND !A22 AND !A21 AND A20 {keep for page write}
                 OR    !REFACKB AND CIPB AND !RAS1B AND !DRAMWNB AND C0
                       {remove in refresh}
                 OR    RAS1B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND !C2
                       AND C1 AND C0 {issue for refresh}
                 OR    !RAS1B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                       AND !C1 AND !C0 {keep for refresh}
                 OR    !RAS1B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                       AND !C1 AND C0; {keep for refresh}

RAS0BEN          =     ENABLEB;
RAS0B NOT        :=    RAS0B AND REFACKB AND RIPB AND DRAMWNB AND !RDB AND
                       !A22 AND !A21 AND !A20 {read/block refill}
                 OR    !RAS0B AND !CIPB AND !RDB AND DRAMACKB AND DRAMRDCENB
                       {keep for read/delayed read}
                 OR    RAS0B AND !CIPB AND RIPB AND !DRAMWNB AND !RDB AND
                       !A22 AND !A21 AND !A20 AND !C3 AND !C2 AND !C1 AND C0
                       {delayed read/delayed block refill}
                 OR    !RAS0B AND !CIPB AND !RDB AND !BURSTB AND !C3 {keep block refill}
                 OR    !RAS0B AND !CIPB AND !RDB AND !BURSTB AND !DRAMWNB AND
                       !C1 {keep delayed block refill}
                 OR    RAS0B AND REFACKB AND RIPB AND DRAMWNB AND !WRB AND
                       !A22 AND !A21 AND !A20 {write}
                 OR    RAS0B AND REFACKB AND RIPB AND !DRAMWNB AND !WRB AND
                       !A22 AND !A21 AND !A20 AND !C3 AND !C2 AND !C1 AND C0
                       {delayed write}
                 OR    !RAS0B AND !WRB AND !CIPB {keep for write}
                 OR    !RAS0B AND !DRAMWNB AND REFACKB AND RIPB AND RDB AND
                       WRB AND BURSTB {no access pending}
                 OR    !RAS0B AND !DRAMWNB AND REFACKB AND RIPB AND !WRB AND
                       !BURSTB AND !A22 AND !A21 AND !A20 {keep for page write}
                 OR    !REFACKB AND CIPB AND !RAS0B AND !DRAMWNB AND C0
                       {remove in refresh}
                 OR    RAS0B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND !C2
                       AND C1 AND C0 {issue for refresh}
                 OR    !RAS0B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                       AND !C1 AND !C0 {keep for refresh}
                 OR    !RAS0B AND !REFACKB AND CIPB AND DRAMWNB AND !C3 AND C2
                       AND !C1 AND C0; {keep for refresh}
```

```
        DRAMWNBEN           =     ENABLEB;
        DRAMWNB  NOT        :=    DRAMWNB AND !CIPB AND RIPB AND !WRB  AND !C3  AND !C2 AND
                                  !C1 AND C0 {write}
                            OR    !DRAMWNB AND !REFACKB AND CIPB AND RIPB AND !C3 AND !C2
                                  AND !C1 AND !C0{remove at refresh}
                            OR    !DRAMWNB AND RIPB AND !RAS3B  {keep asserted if any RAS}
                            OR    !DRAMWNB AND RIPB AND !RAS2B
                            OR    !DRAMWNB AND RIPB AND !RAS1B
                            OR    !DRAMWNB AND RIPB AND !RAS0B
                            OR    !DRAMWNB AND RIPB AND !RDB AND !CIPB {keep for read}
                            OR    !DRAMWNB AND RIPB AND !WRB AND !CIPB;   {keep for write}

        DRAMACKBEN          =     ENABLEB;
        DRAMACKB  NOT       :=    !CIPB AND !RDB AND DRAMWNB AND BURSTB AND !C3 AND !C2 AND
                                  !C1 AND C0 {read}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND BURSTB AND !C3 AND !C2
                                  AND C1 AND C0 {delayed read}
                            OR    !CIPB AND !RDB AND DRAMWNB AND !BURSTB AND !C3 AND C2 AND
                                  !C1 AND !C0 {block refill}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND !BURSTB AND !C3 AND C2
                                  AND C1 AND !C0 {delayed block refill}
                            OR    !CIPB AND !WRB AND DRAMWNB  AND !C3 AND !C2  AND !C1 AND !C0
                                  {write}
                            OR    !CIPB AND !WRB AND !DRAMWNB AND BURSTB AND !C3 AND !C2
                                  AND C1 AND !C0 {delayed write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND REFACKB AND RIPB AND
                                  CIPB AND !A22 AND !RAS3B {page write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND REFACKB AND RIPB AND
                                  CIPB AND !A22 AND !RAS2B {page write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND REFACKB AND RIPB AND
                                  CIPB AND !A22 AND !RAS1B {page write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND REFACKB AND RIPB AND
                                  CIPB AND !A22 AND !RAS0B ;{page write}

        DRAMRDCENBEN        =     ENABLEB;
        DRAMRDCENB  NOT     :=    !CIPB AND !RDB AND DRAMWNB AND BURSTB AND !C3 AND !C2 AND
                                  !C1 AND C0 {read}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND BURSTB AND !C3 AND !C2
                                  AND C1 AND C0 {delayed  read}
                            OR    !CIPB AND !RDB AND DRAMWNB AND !BURSTB AND !C3 AND C0
                                  {block  refill}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND !BURSTB AND !C3 AND !C2
                                  AND C1 AND C0 {delayed block refill}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND !BURSTB AND !C3 AND C2
                                  AND C0 {delayed block refill}
                            OR    !CIPB AND !RDB AND !DRAMWNB AND !BURSTB AND C3 AND !C2
                                  AND !C1 AND C0;   {delayed block refill}

        CIPBEN              =     ENABLEB;
        CIPB  NOT           :=    CIPB AND REFACKB AND RIPB AND !RDB AND !A22 {read}
                            OR    CIPB AND REFACKB AND RIPB AND !WRB AND !A22 {write}
                            OR    !CIPB AND !RDB {keep for read}
                            OR    !CIPB AND !WRB ;{keep for write}

END;
END  PAL1.
```

```
{
        TITLE:              PAL2
        PURPOSE:            CAS
        AUTHOR:             BOB NAPAA, IDT INC.
        DATE:               4/5/91
}

MODULE  PAL2;
TITLE  PAL2;
TYPE MMI  20R8;

INPUTS;

        {SYSCLKB                    NODE[PIN1];  }
        REFACKB                     NODE[PIN2];
        DRAMWNB                     NODE[PIN3];
        BURSTB                      NODE[PIN4];
        RIPB                        NODE[PIN5];
        CIPB                        NODE[PIN6];
        WRB                         NODE[PIN7];
        A21                         NODE[PIN8];
        A20                         NODE[PIN9];
        C3                          NODE[PIN10];
        C2                          NODE[PIN11];
        {OUTENABLEB                 NODE[PIN13];  }
        C1                          NODE[PIN14];
        C0                          NODE[PIN23];

        {FEED  BACK  PINS}
        CAS3B                       NODE[PIN22];
        CAS2B                       NODE[PIN21];
        CAS1B                       NODE[PIN20];
        CAS0B                       NODE[PIN19];
        DRAMLE                      NODE[PIN18];
        DRAMWRB                     NODE[PIN17];
        SB                          NODE[PIN16];
        TRB                         NODE[PIN15];

OUTPUTS;

        CAS3B                       NODE[PIN22];
        CAS2B                       NODE[PIN21];
        CAS1B                       NODE[PIN20];
        CAS0B                       NODE[PIN19];
        DRAMLE                      NODE[PIN18];
        DRAMWRB                     NODE[PIN17];
        SB                          NODE[PIN16];
        TRB                         NODE[PIN15];

TABLE;

        CAS3B  NOT          :=      CAS3B AND RIPB AND !CIPB  AND DRAMWNB AND (A21 AND A20
                                    AND !C3 AND !C2 AND !C1 AND C0)   {read or write}
                            OR      CAS3B AND RIPB AND !CIPB  AND !DRAMWNB AND (A21 AND A20
                                    AND !C3 AND !C2 AND C1 AND C0)   {delayed read/write}
                            OR      CAS3B AND RIPB AND !CIPB AND !BURSTB AND DRAMWNB AND
                                    WRB AND !SB AND (A21 AND A20 AND !C3 AND C0)   {block refill}
                            OR      CAS3B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                                    WRB AND !SB AND (A21 AND A20 AND C0)   {delayed block refill}
                            OR      CAS3B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                                    !WRB AND !SB AND (A21 AND A20 AND !C3 AND !C2 AND !C1 AND
                                    !C0)   {page write}
```

```
                OR    CIPB AND DRAMWNB AND !REFACKB AND CAS3B AND (!C3 AND !C2
                      AND C1 AND !C0)   {refresh}
                OR    CIPB AND DRAMWNB AND !REFACKB AND !CAS3B AND (!C3 AND !C2
                      AND C1 AND C0); {refresh}

CAS2B  NOT      :=    CAS2B AND RIPB AND !CIPB  AND DRAMWNB AND (A21 AND !A20
                      AND !C3 AND !C2 AND !C1 AND C0)   {read or write}
                OR    CAS2B AND RIPB AND !CIPB  AND !DRAMWNB AND (A21 AND !A20
                      AND !C3 AND !C2 AND C1 AND C0)   {delayed read/write}
                OR    CAS2B AND RIPB AND !CIPB AND !BURSTB AND DRAMWNB AND
                      WRB AND !SB AND (A21 AND !A20 AND !C3 AND C0)   {block refill}
                OR    CAS2B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      WRB AND !SB  AND (A21 AND !A20 AND C0)   {delayed block refill}
                OR    CAS2B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      !WRB AND !SB AND (A21 AND !A20 AND !C3 AND !C2 AND !C1 AND
                      !C0)   {page write}
                OR    CIPB AND DRAMWNB AND !REFACKB AND CAS2B AND
                      (!C3 AND !C2 AND C1 AND !C0) {refresh}
                OR    CIPB AND DRAMWNB AND !REFACKB AND !CAS2B AND
                      (!C3 AND !C2 AND C1 AND C0); {refresh}

CAS1B  NOT      :=    CAS1B AND RIPB AND !CIPB AND DRAMWNB AND (!A21 AND A20
                      AND !C3 AND !C2 AND!C1 AND C0)    {read or write}
                OR    CAS1B AND RIPB AND !CIPB  AND !DRAMWNB AND (!A21 AND A20
                      AND !C3 AND !C2 AND C1 AND C0)    {delayed read/write}
                OR    CAS1B AND RIPB AND !CIPB AND !BURSTB AND DRAMWNB AND
                      WRB AND !SB AND (!A21 AND A20 AND !C3 AND C0)    {block refill}
                OR    CAS1B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      WRB AND !SB  AND (!A21 AND A20 AND C0)   {delayed block refill}
                OR    CAS1B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      !WRB AND !SB AND (!A21 AND A20 AND !C3 AND !C2 AND !C1 AND
                      !C0)   {page write}
                OR    CIPB AND DRAMWNB AND !REFACKB AND CAS1B AND (!C3 AND !C2
                      AND C1 AND !C0) {refresh}
                OR    CIPB AND DRAMWNB AND !REFACKB AND !CAS1B AND (!C3 AND !C2
                      AND C1 AND C0); {refresh}

CAS0B  NOT      :=    CAS0B AND RIPB AND !CIPB AND DRAMWNB AND (!A21 AND !A20
                      AND !C3 AND !C2 AND !C1 AND C0)   AND CAS0B {read or write}
                OR    CAS0B AND RIPB AND !CIPB  AND !DRAMWNB AND (!A21 AND !A20
                      AND !C3 AND !C2 AND C1 AND C0)   AND CAS0B {delayed read/write}
                OR    CAS0B AND RIPB AND !CIPB AND !BURSTB AND DRAMWNB AND
                      WRB AND  !SB AND (!A21 AND !A20 AND !C3 AND C0)    {block refill}
                OR    CAS0B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      WRB AND !SB  AND (!A21 AND !A20 AND C0)   {delayed block refill}
                OR    CAS0B AND RIPB AND !CIPB AND !BURSTB AND !DRAMWNB AND
                      !WRB AND !SB AND (!A21 AND !A20 AND !C3 AND !C2 AND !C1 AND
                      !C0)    {page write}
                OR    CIPB AND DRAMWNB AND !REFACKB AND CAS0B AND (!C3 AND !C2
                      AND C1 AND !C0) {refresh}
                OR    CIPB AND DRAMWNB AND !REFACKB AND !CAS0B AND (!C3 AND !C2
                      AND C1 AND C0); {refresh}

DRAMLE  NOT     :=    TRB AND CAS3B AND CAS2B AND CAS1B AND !CAS0B {issue after}
                OR    TRB AND !CAS3B AND CAS2B AND CAS1B AND CAS0B {any CAS if}
                OR    TRB AND CAS3B AND !CAS2B AND CAS1B AND CAS0B {read cycle}
                OR    TRB AND CAS3B AND CAS2B AND !CAS1B AND CAS0B
                OR    CAS3B AND CAS2B AND CAS1B AND CAS0B;
```

```
      DRAMWRB  NOT          :=    !CIPB AND RIPB AND !WRB AND DRAMWRB {issue for write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND DRAMWRB AND RIPB
                                  AND REFACKB {issue for page write}
                            OR    !CIPB AND !DRAMWRB AND CAS3B AND CAS2B AND CAS1B
                                  AND CAS0B AND RIPB; {keep until end of write}

      SB NOT                :=    SB AND !CIPB AND DRAMWNB AND (!C3 AND !C2 AND !C1
                                  AND !C0) {read/write/block refill}
                            OR    !SB AND !CIPB AND !BURSTB AND WRB AND !C3 {keep for block refill}
                            OR    SB AND !CIPB AND !DRAMWNB AND (!C3 AND !C2 AND C1
                                  AND !C0) {delayed read/write/block refill}
                            OR    !SB AND !CIPB AND !DRAMWNB AND !BURSTB AND WRB AND
                                  !C1 {delayed block refill}
                            OR    !SB AND !CIPB AND BURSTB AND WRB AND C0 AND CAS3B AND
                                  CAS2B AND CAS1B AND CAS0B {read and delayed read}
                            OR    !SB AND !CIPB AND !WRB  AND CAS3B AND CAS2B AND CAS1B AND
                                  CAS0B {keep for write}
                            OR    !WRB AND !BURSTB AND !DRAMWNB AND SB AND REFACKB; {page write}

      TRB NOT               :=    TRB AND !CIPB AND WRB AND DRAMWNB AND (!C3 AND !C2
                                  AND !C1 AND !C0) {read/block refill}
                            OR    TRB AND !CIPB AND WRB AND !DRAMWNB AND SB AND (!C3
                                  AND !C2 AND C1 AND !C0) {delayed read/block refill}
                            OR    !TRB AND !CIPB AND !SB; {keep asserted for read/block refill}

END;
END  PAL2.
```

```
{
      TITLE:              PAL3
      PURPOSE:            COUNTER
      AUTHOR:             BOB NAPAA,  IDT  INC.
      DATE:               4/5/91
}

MODULE  PAL3;
TITLE  PAL3;
TYPE  MMI  16R8;

INPUTS;

      {BSYSCLKB                 NODE[PIN1];  }
      RESETB                    NODE[PIN2];
      REFREQ                    NODE[PIN3];
      BCIPB                     NODE[PIN4];
      DRAMWNB                   NODE[PIN5];
      {OUTENABLEB               NODE[PIN11];  }

      {FEED  BACK  PINS}
      RIPB                      NODE[PIN18];
      C3                        NODE[PIN17];
      C2                        NODE[PIN16];
      C1                        NODE[PIN15];
      C0                        NODE[PIN14];
      REFACKB                   NODE[PIN13];
      GATECOUNTERB              NODE[PIN12];

OUTPUTS;

      RIPB                      NODE[PIN18];
      C3                        NODE[PIN17];
      C2                        NODE[PIN16];
      C1                        NODE[PIN15];
      C0                        NODE[PIN14];
      REFACKB                   NODE[PIN13];
      GATECOUNTERB              NODE[PIN12];

TABLE;

      RIPB  NOT                 :=    !RESETB       {reset}
                                OR    !RIPB  AND  !RESETB    {keep  for  reset}
                                OR    !RIPB  AND  REFACKB    {keep  for  refresh}
                                OR    !RIPB  AND  !REFACKB  AND  !C3;  {keep  until  end  of  refresh}

      C3  NOT                   :=    !GATECOUNTERB  AND  !BCIPB  AND  REFACKB
                                OR    !GATECOUNTERB  AND  BCIPB
                                OR    GATECOUNTERB  AND  BCIPB  AND  REFACKB
                                OR    !C3  AND  !C2
                                OR    !C3  AND  C2  AND  !C1
                                OR    !C3  AND  C2  AND  C1  AND  !C0
                                OR    C3  AND  C2  AND  C1  AND  C0;

      C2  NOT                   :=    !GATECOUNTERB  AND  !BCIPB  AND  REFACKB
                                OR    !GATECOUNTERB  AND  BCIPB
                                OR    GATECOUNTERB  AND  BCIPB  AND  REFACKB
                                OR    !C2  AND  !C1
                                OR    !C2  AND  C1  AND  !C0
                                OR    C2  AND  C1  AND  C0;
```

```
C1 NOT                    :=    !GATECOUNTERB AND !BCIPB AND REFACKB
                          OR    !GATECOUNTERB AND BCIPB
                          OR    GATECOUNTERB AND BCIPB AND REFACKB
                          OR    !C1 AND !C0
                          OR    C1 AND C0;

C0 NOT                    :=    !GATECOUNTERB AND !BCIPB AND REFACKB
                          OR    !GATECOUNTERB AND BCIPB
                          OR    GATECOUNTERB AND BCIPB AND REFACKB
                          OR    C0;

REFACKB NOT          :=   REFACKB AND REFREQ AND RESETB     {for refreq}
                          OR    !REFACKB AND !BCIPB AND RESETB    {as long as cipb low}
                          OR    !REFACKB AND !C3 AND RESETB AND GATECOUNTERB
                     {keep asserted}
                          OR    REFACKB AND RESETB AND !RIPB {reset}
                          OR    !REFACKB AND !GATECOUNTERB; {keep for reset}

GATECOUNTERB NOT := GATECOUNTERB AND !REFACKB   AND !BCIPB AND RIPB
                                    {issue for both refack and cipb}
                          OR    !GATECOUNTERB AND !BCIPB AND RIPB
                                      {keep as long as cipb}
                          OR    !GATECOUNTERB AND !REFACKB AND RIPB AND C3
                          OR    !GATECOUNTERB AND !REFACKB AND RIPB AND C2
                          OR    !GATECOUNTERB AND !REFACKB AND RIPB AND C1
                          OR    !GATECOUNTERB AND !REFACKB AND RIPB AND C0;

END;
END PAL3.
```

Integrated Device Technology, Inc.

IDT79R3051™ MAIN MEMORY AND SYSTEM I/O INTERFACING

APPLICATION NOTE AN-92

By Andrew Ng

INTRODUCTION

The IDT79R3051™ RISController™ family provides a simple, flexible external bus interface to directly support main memory and system I/O resources. The bus interface is straightforward in that it uses a single, multiplexed 32-bit address and data bus and a small number of supporting control signals. The bus interface is adaptable in that it can handle different types and speeds of memory including DRAM, SRAM, and EPROM and different kinds of I/O resources. Thus the simple, flexible R3051 bus interface allows designers to make optimal trade-offs between system speed and cost issues.

MAIN MEMORY DESIGN

The R3051 normally accesses its internal instruction and data cache memories as in Figure 1, while using external main memory as a secondary source of memory as in Figure 5. Since the R3051 contains its own internal instruction and data caches, the complexity of the cache timing and interfacing is kept on-chip, which allows the external interface to be dedicated to main memory and system I/O interfacing. The system interface is decoupled from cache memory by the use of an internal 4-deep read buffer and an internal 4-deep write buffer. The instruction and data cache allow the R3051 to access 1 instruction and 1 data word on each clock cycle. On reads, when a cache miss or an uncachable reference occurs, the R3051 begins an external read cycle which buffers 1 word on non-burst reads and 4 words at a time on burst reads from system I/O and main memory. On writes, the R3051 maintains a write-through cache update policy which simultaneously updates both the data cache and main memory. With the use of its 4-deep write buffer, the R3051 can continue to execute instructions from its instruction cache while the main memory retires up to 4 words from the write buffer.

Read and Write Cycle Protocols

The simple read interface allows a wide range of memories and I/O to be used with the R3051, from slow I/O peripherals to high speed burst accessed DRAM and SRAM. As shown in Figure 2 and 3, the read interface supports both single datum accesses and 4-word burst accesses simply by providing a Burst output signal and by providing dedicated LSB address line outputs Addr(3:2) which are used as a word counter. System I/O or main memory is only required to acknowledge each of the 4 words with the $\overline{RdCEn}$ input which is used as a read clock enable to latch each word into the 4-deep read buffer. Read interfacing also has the option of using the $\overline{Ack}$

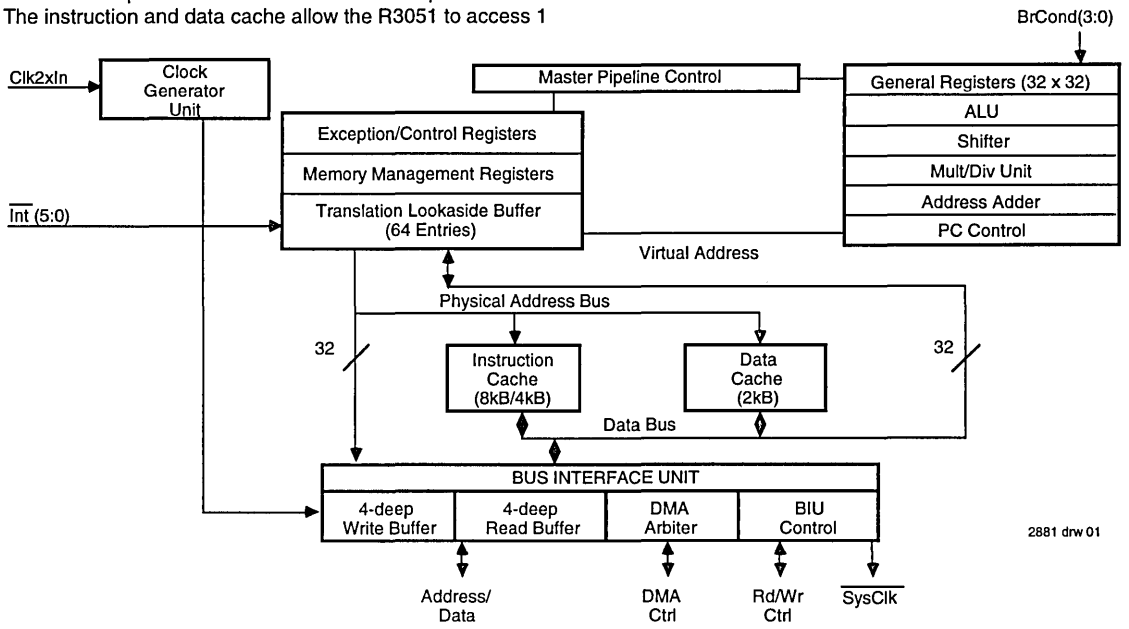


Figure 1. R3051 RISController Internal Architecture

acknowledge input signal to optimally control when the R3051 core restarts its pipeline on burst read cycles.

The simple write interface allows a wide range of memories and I/O to be used with the R3051 by buffering writes from the R3051 core which are done at cache speeds. This allows main memory and I/O to retire write cycles at their own rate of speed by returning $\overline{Ack}$, to acknowledge that the word has been received as shown in Figure 4.

Basic System Functional Blocks

The following sections will describe the functional blocks that are typical of R3051 main memory and system I/O interfacing. As shown in Figure 5 these blocks include:

- Address De-multiplexing
- Address Decoding and Chip Selection
- Data Transceivers
- Wait-State Controller and Interface Handshaking
- Read/Write Enables and Strobes

The discussion concentrates on the general interface blocks involved when using the following modules:

- SRAM Interfacing
- DRAM Interfacing
- EPROM Interfacing
- I/O Interfacing
- DMA Interfacing

Specific information on using the different memory and I/O types is presented in detail in other application notes.

ADDRESS DE-MULTIPLEXER AND DECODER

The R3051 uses a multiplexed A/D(31:0) bus to output its address and to send and receive data. Thus main memory must de-multiplex the address by using the R3051's Address Latch Enable control signal, ALE, before decoding the address to select chip enables.

Latching A/D(31:0)

Transparent latches such as the IDT54/74FCT373 and the IDT54/74FCT841 pass inputs straight through to the outputs when their Latch Enable input is high. When their Latch Enable input is low, the data in the latches are held constant. The R3051 provides the ALE output for direct connection to the transparent latches' Latch Enable pins. Transparent latches are typically used to allow address decoding to take place when ALE is high and the address begins to become valid, instead of waiting until the latch closes.

The Address Latch Enable, ALE, is designed to clock the address into a transparent latch such as the FCT373. ALE is

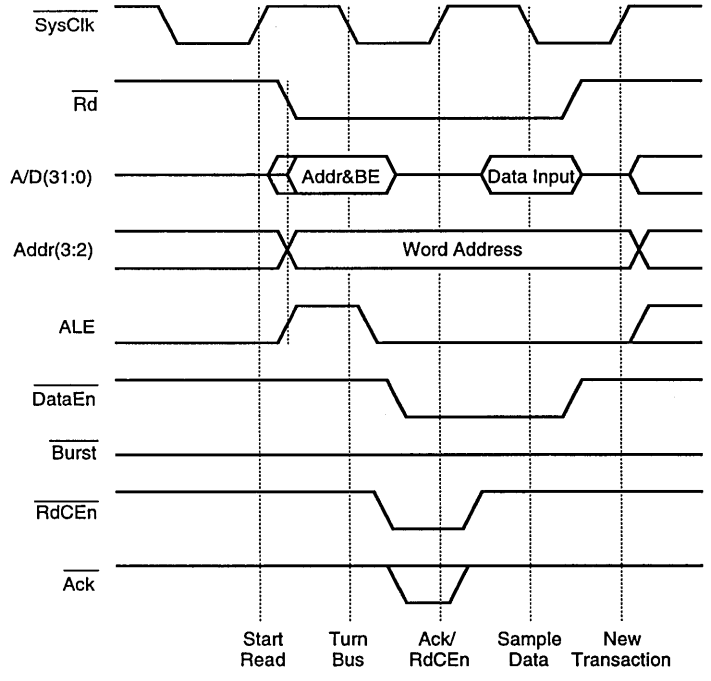


Figure 2. R3051 Single Word Read

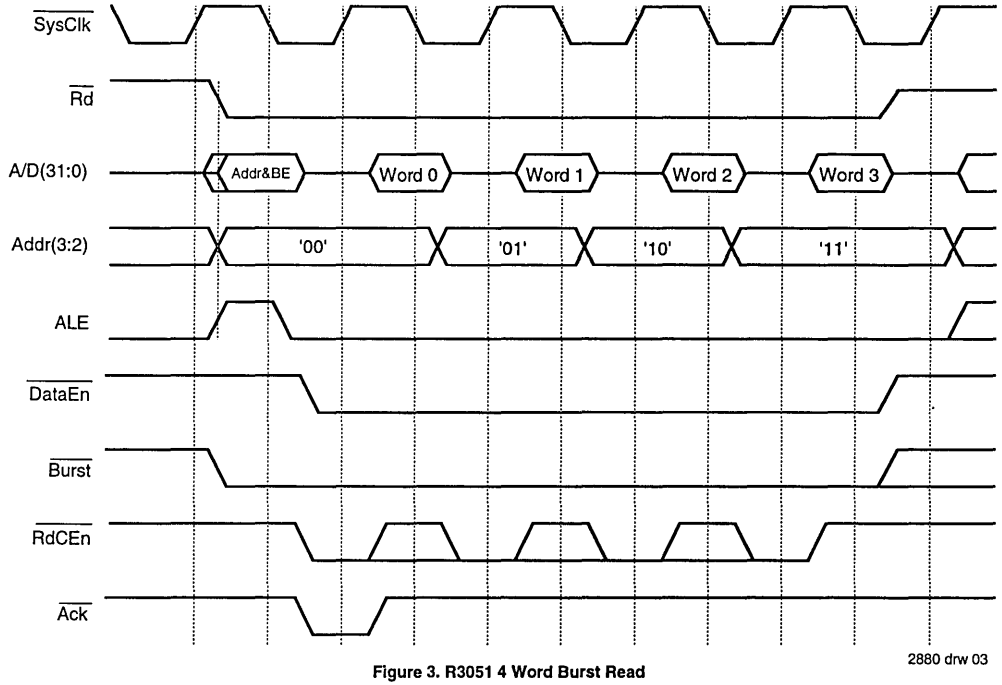


Figure 3. R3051 4 Word Burst Read

also designed to meet the address hold time of latches. As with all high speed processors, ALE should be considered a critical signal. Thus Printed Circuit Board routing should minimize ALE's trace length and crosstalk susceptibility.

Decoding A(31:0)

Address decoding, which selects between the various memory and I/O banks in the system, can be done with IDT54/74FCT138/139 decoders as shown in Figure 6.

The time for the main memory chip selects to become valid in such a scheme is:

$$t_{Decode} = \max(t_{3051ALEProp} + t_{373LEtoO}, t_{3051AddrProp} + t_{373DtoO}) + t_{138AtoO} + t_{Cap}$$

Systems that require the chip selects to not have decoding glitches while the address drives to a valid value can register the decoder outputs by using $\overline{SysClk}$ as the clock and a $\overline{CycleStart}$ signal as the clock enable. The $\overline{CycleStart}$ signal is derived from the $\overline{Rd}$ and $\overline{Wr}$ control lines so that it asserts at the beginning of every memory cycle.

Decoding Byte Enables with Chip Selects

During the address phase, the R3051 uses the lower 4 bits of the multiplexed A/D(31:0) bus to output $\overline{BE}(3:0)$. Byte enables are used to determine which bytes of each word are being read or written to support partial word accesses. Because $\overline{BE}(3:0)$ are used throughout the memory cycle, they are latched by ALE along with the other A/D bits.

In general, it is permissible to process all reads as 32-bit reads—the processor will only take the data it requested from the bus. However, in write operations, the system must insure that only the specified bytes are written. Thus, the byte enable outputs are used to control this.

There are two ways in which the byte enables may be used:

- Gate the byte enables with the memory chip selects. Thus, only those bytes of memories which will be written are selected. A single write enable can then be presented to all banks of that memory subsystem. This solution requires that each memory sub-system further decode the chip-selects, and thus one decoder per memory sub-system is required.
- Gate the byte enables with the memory chips read/write enables/strobes. Thus, although all of the devices in that bank of memory are "selected", only those bytes to be written are enabled for the writes. This is a common strategy in DRAM sub-systems. Note that the individual byte strobes may be broadcast to all memory systems, and the address decoder will insure that only one sub-system is "Selected". Thus, a single decoder for byte enables can serve the entire memory system.

If the memories being used are 1-bit to 8-bits wide, gating the byte enables with the chip selects can be done. Because the byte enables are predetermined within the R3051 by using the LSB address bits, the endianness of the system, and the

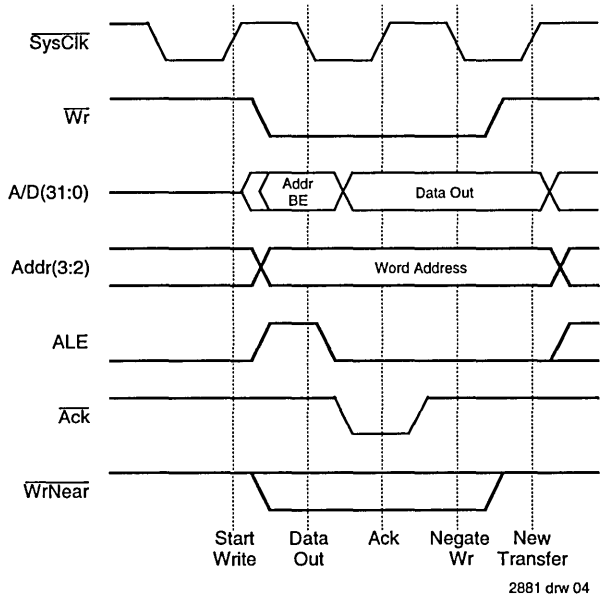


Figure 4. R3051 Single Word Write

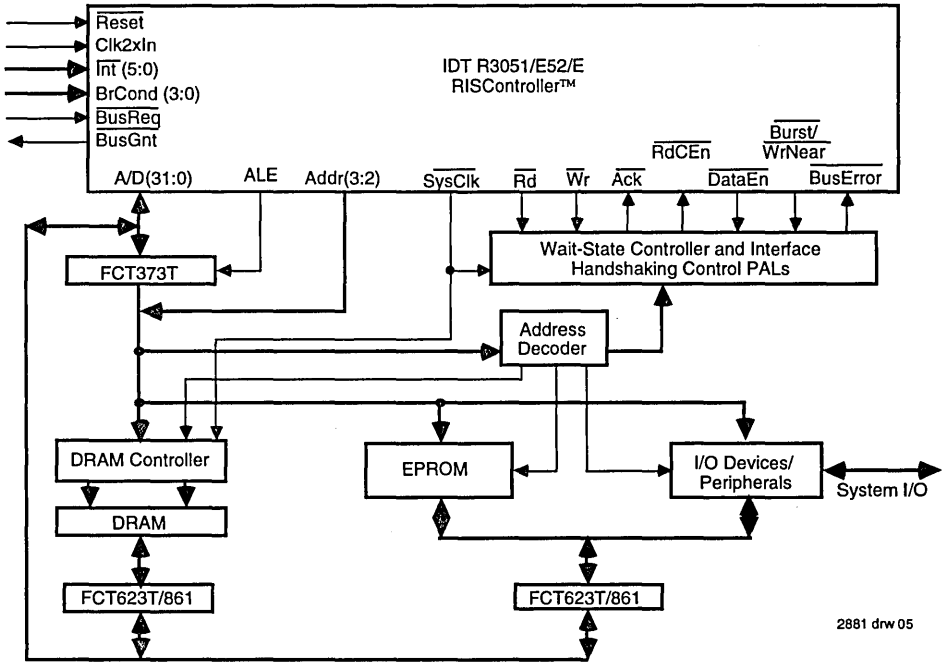


Figure 5. R3051 with Main Memory

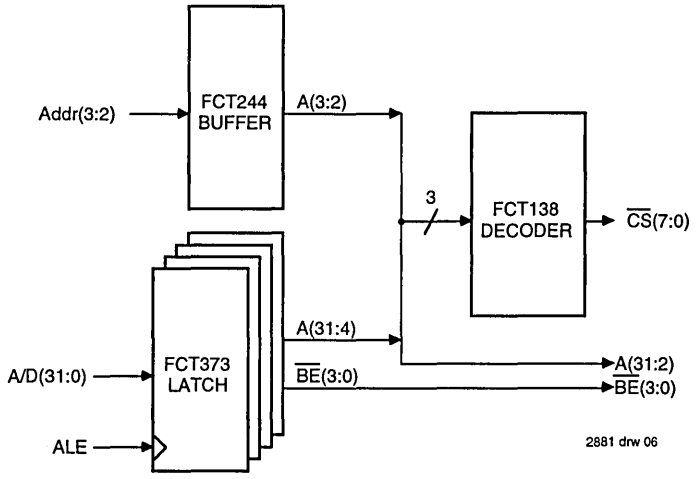


Figure 6. Address De-multiplexer and Decoder

type of load or store instruction, the byte enables have the same timing as the rest of the A/D lines during the address phase when ALE is asserted. This allows a memory decoder to have individual chip selects for each byte of each bank with no timing penalty. An example is shown in Figure 7.

As gating the byte enables with the chip selects usually takes more output pins than gating the byte enables with the read and write enables, the latter is usually preferred. The use of byte enables with read/write enables will be discussed in the read/write enable/strobe section.

Using Addr(3:2)

Since the lower 4 A/D bits are used for byte enables during the R3051's address phase, the R3051 provides the information for addressing words through its Addr(3:2) output pins. The R3051 uses 4 bytes per word and pre-decodes the byte enables instead of providing the 2 LSB address lines. Addr(3:2) are driven throughout external bus cycles and do not require latching. During non-burst read cycles and all write cycles, Addr(3:2) contains the instruction cache miss address. The advantage of dedicating output pins for Addr(3:2) is that during burst read cycles, Addr(3:2) are incremented from 0 to 3 by the R3051 $\overline{RdCEn}$ protocol so that the system memory system does not have to provide a counter for this function.

Since each memory chip requires Addr(3:2), large memory systems that use Addr(3:2) extensively may want to use buffers. A common strategy may be to provide a buffered version of Addr(3:2) to non-time critical areas of memory (e.g. the boot prom), or to areas which do not perform burst accesses (I/O devices), and directly use the outputs of the R3051 in time-critical areas such as the DRAM control.

The crossover point where buffering is appropriate can be determined by determining if the delay through an IDT54/74FCT244 buffer and the capacitive derating from all the Addr(3:2) inputs driven by the buffer (Addr(3:2) can be buffered for separate branches of memory banks) would be less than the delay from the capacitive derating from all the Addr(3:2) inputs driven directly from the R3051. In addition, the crossover doesn't occur until Addr(3:2) is delayed past when rest of the A(31:4) lines reach their inputs.

$$t_{3051Addr(3:2)} + t_{244} + t_{244Cap} \leq \max(t_{3051Addr(3:2)} + t_{3051Cap}, t_{A(31:4)})$$

where:

$$t_{244Cap} = (\text{sum}(C_{Input/Output}) + C_{244} + t_{Trace} - 50)/33 \text{ pf/nsec}$$

$$t_{3051Cap} = (\text{sum}(C_{Input/Output}) + C_{3051} + t_{Trace} - 25)/25 \text{ pf/nsec}$$

Using Diag(1:0)

Some systems may need to know whether a read cycle is cachable or uncachable and whether a cachable read cycle is an instruction or a data fetch. In Figure 8, this information is provided by latching the diagnostic pins, Diag(1:0) with the same latch controls as the address lines. These signals are useful for:

- Decoding whether a reference to the lowest half GB of physical memory is from kseg0 or kseg1.
- Tracing processor execution by knowing which address caused the I-Cache miss.

DATA TRANSCEIVERS

The R3051 uses a multiplexed A/D(31:0) bus to output its address and to send and receive data. Thus main memory must drive or receive data after the R3051 has tri-stated its address. Further, to support high-performance memory systems, the R3051 family is capable of initiating a new bus transaction one-half clock cycle after data is sampled for a read operation.

Determining if Data Transceivers are needed

Multiplexed CPU busses often use data transceivers to separate the memory system from the processor bus. Read cycles require the memory system to stop driving data on the A/D bus before the processor drives the next memory cycle's address. Slow memories with relatively long output disable times cannot meet this limitation without data transceivers. However, some memories, such as the IDT71B256 BiCEMOS™ 32Kx8 Static RAM, have very short access time and output disable time which makes it possible to consider attaching memory device data I/O pins directly to the multiplexed A/D(31:0) bus. Alternatively, in low frequency systems, the amount of time provided by the R3051 may be sufficient for the memory devices attached to the bus.

The key parameter is the memory output disable time, TOZ, which has to be less than 1/2 clock to disable before the next memory's address is driven. In addition the address and data driven from the R3051 is delayed because of the extra capacitance of the memory data I/O pins.

$$tOZ \leq tSysClk/2 - tDisableControl + \min(t3051Addr)$$

Data Transceivers also serve to isolate memory banks from each other. In systems with varying speeds of memory, transceiver banks can be used to separate chips with relatively long output disable times from those with relatively quick output disable times. Thus in many systems, fast scratch-pad SRAMs may have their own set of transceivers, while slower EPROMs and I/O peripherals might have a separate set of transceivers.

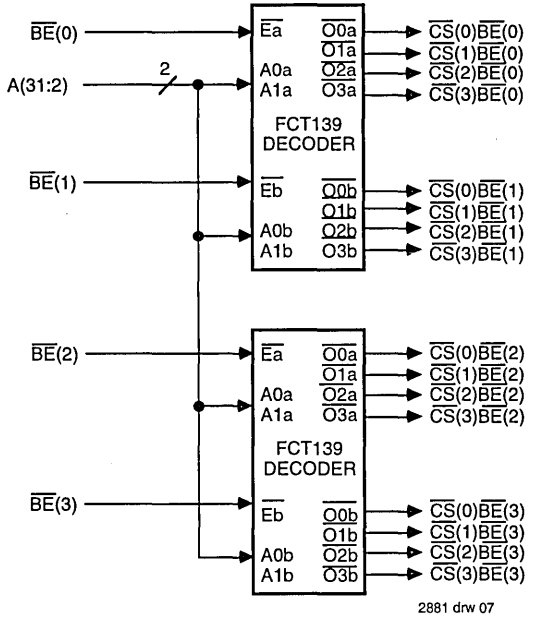


Figure 7. Gating Byte Enables with Chip Selects

Using IDT54/74FCT861's and IDT54/74FCT245's for Data Transceivers

Most systems will use slower memories and thus require data buffering through a transceiver interface. There are two basic families of transceiver interfaces:

1: IDT54/74FCT861 with separate enable pins for each direction
2: IDT54/74FCT245 with a direction pin and an enable pin

Using IDT54/74FCT861's for Data Transceivers

The 10-bit transceiver FCT861 approach functionally combines two 10-bit tri-statable FCT827 buffers internally. The 8-bit FCT623T transceiver is similar to the FCT861 except that one of its output enables is active high. On read cycles, if there is only one transceiver bank, then DataEn can be used directly to control the read direction output enable. Otherwise, combinational logic such as an FCT157/257 multiplexer can be used to combine DataEn with the chip selects of the bank whose transceivers need to be enabled (see Figure 16 for a similar common input OR gate circuit). Alternatively, some transceivers, such as the 9-bit IDT54/74FCT863 and the 8-bit IDT54/74FCT543 have two logically AND'ed output enables for each direction so that DataEn and the bank chip select can be hooked up directly to the transceiver. State machines using an inverted SysClk can also use a Rd derived signal to synchronously assert and de-assert the read direction output enable.

The write direction output enable can use a signal derived from Wr which asserts at the beginning of the cycle and waits until after the data has been strobed into the memory or I/O device before de-asserting to provide sufficient data setup and hold time. For systems with 1 wait-state or more, the derived write direction enable signal should ideally assert after the A/D bus finishes driving its address phase to reduce switching noise.

The transceiver control's critical timing path is the transition from a read cycle to a write cycle. After a read cycle, slower memory chips take a relatively long time to disable from the data bus. If the next memory cycle is a write, the transceivers will drive data onto the same bus. Such systems can use the second memory cycle's wait-states to delay the assertion of the transceiver's write direction output enable until the first memory cycle's memory has fully disabled. The cutoff for

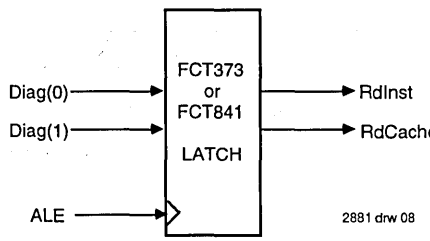


Figure 8. Latching Diag(1:0)

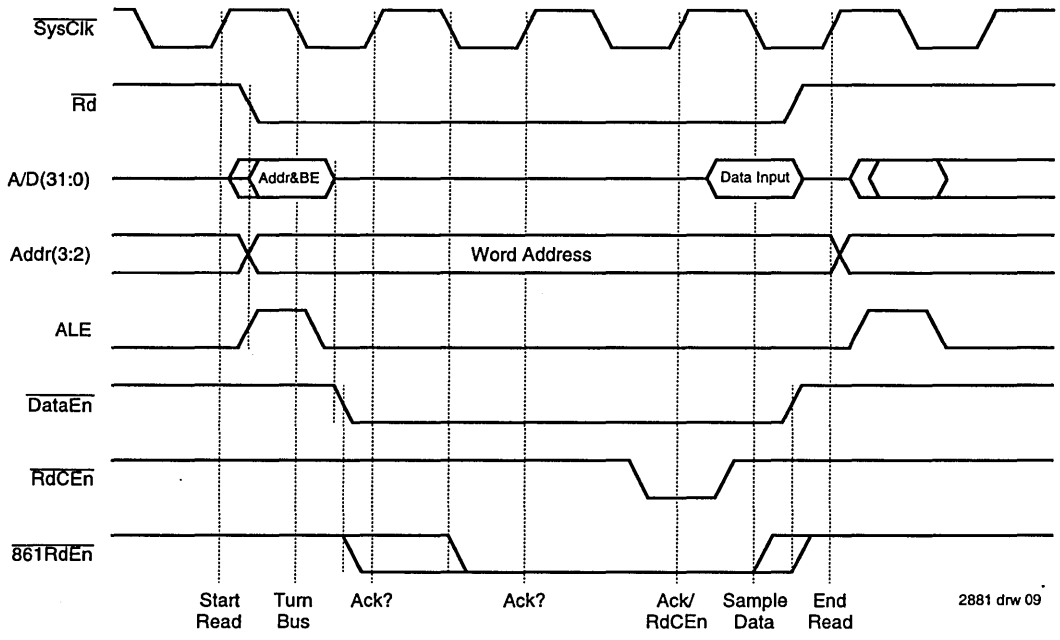


Figure 9a. Timing Diagram of FCT861 Read Direction Enable

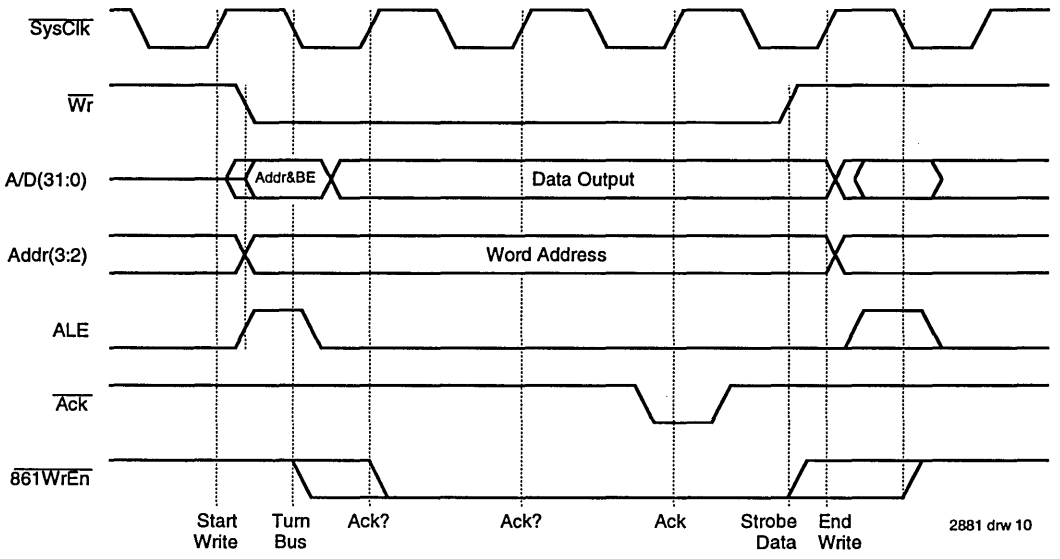


Figure 9b. Timing Diagram of FCT861 Write Direction Enable

determining if the memory output disable time is small enough to require no wait-states is:

tSysClk >= tDisableControl + tMemReadDisable - tWriteData

Systems that use memory chips without an output enable pin (i.e., a read is implied for every chip select with no write enable) require special transceiver interfacing in order to support partial word writes. During partial word writes, where only some of the bytes are selected for writing, bytes which are not being written may actually output onto their byte lanes, and thus conflict with the transceiver write direction outputs. In such memory sub-systems, there are two options: only chip select those devices actually being written into; or, only enable those transceivers whose byte lanes are used in this write transfer. Either of these solutions will insure that no bus conflict occurs.

Using IDT54/74FCT245's for Data Transceivers

The 8-bit FCT245 transceiver approach ideally requires that the direction control only be changed when the outputs are disabled to prevent bus contention. Although such systems are easy to design, this general discussion uses the following assumptions:

1: Either a SysClk or SysClk based state machine is used.
2: The memories require at least 1 wait-state.

The output enable of an FCT245 needs to be determined by finding the start and end of the memory cycle, which can be determined by logically AND'ing $\overline{Rd}$ and $\overline{Wr}$. The assertion of the output enable can be easily delayed to occur well after the transfer, depending on the number of wait-states in the memory controller. That is, the transceiver only needs to be enabled in time to allow the data to propagate through to the CPU as the read data response is finally returned to the processor. In read cycles, the output may be disabled using the same clock edge as is used by the CPU to negate $\overline{Rd}$. On write transactions, the transceiver must be enabled until the data set-up and hold time requirements of the memory being written are met, which may extend until the next falling edge of $\overline{SysClk}$ (note for the R3051, the processor guarantees that valid data will remain for one-half clock cycle after the negation of $\overline{Wr}$).

The T/$\overline{R}$ direction pin of the FCT245 should be asserted before the output enable asserts, which can be achieved by using a $\overline{Rd}$ or $\overline{Wr}$ derived signal. The direction should be held until the next clock edge after $\overline{Rd}$ or $\overline{Wr}$ de-asserts; that is, until after the output enable is de-asserted..

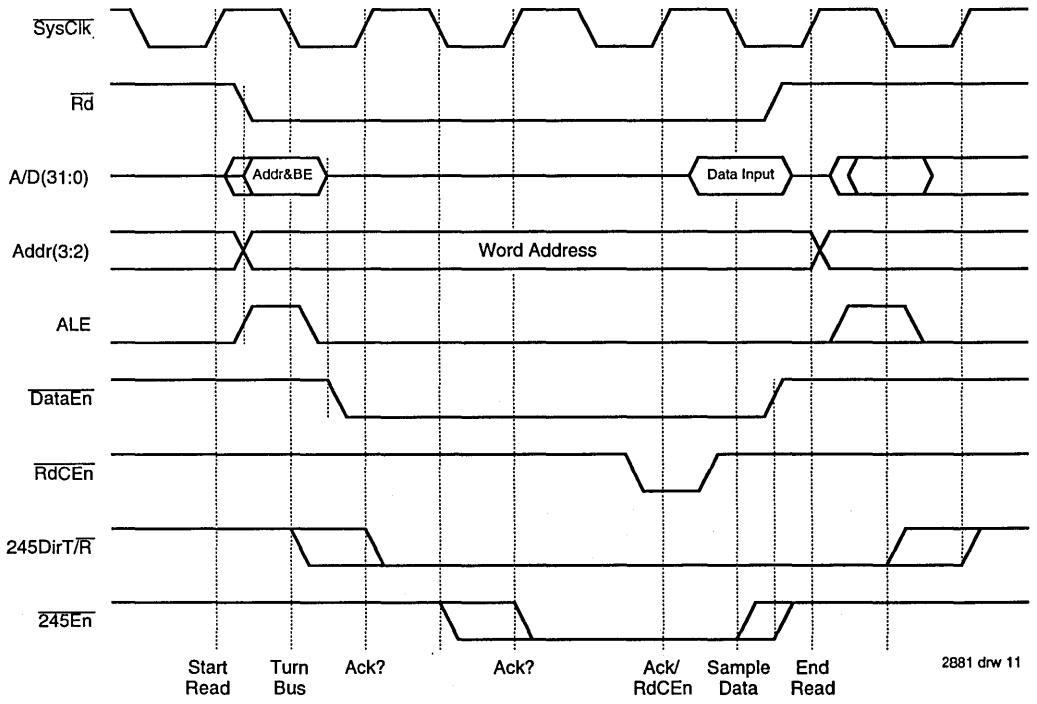


Figure 10a. Timing Diagram of FCT245 Enable and T/$\overline{R}$ Direction Controls for a Read

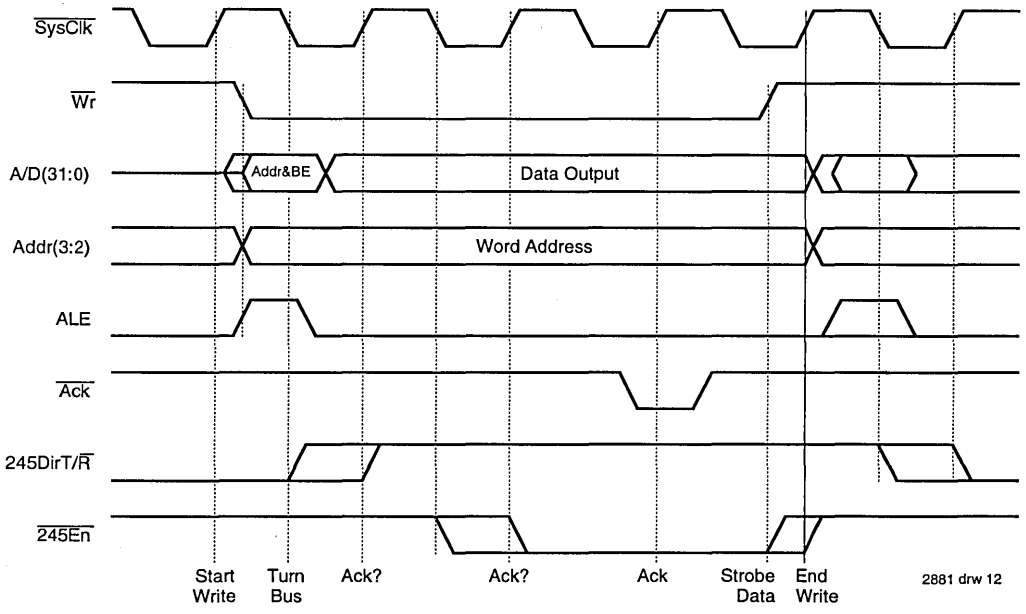


Figure 10b. Timing Diagram of FCT245 Enable and T/R Direction Controls for a Write

Systems that use memories without a dedicated output enable pin require separate byte output enables in the data path, as discussed above.

PULL-DOWN/UP RESISTORS ON R3051 OUTPUTS

The R3051 tri-states its outputs under three conditions:

1: If no external read or write memory cycles are being executed, the A/D bus will tri-state. Control signal outputs will be driven to negated states.
2: If a DMA bus grant is given, all bus interface outputs will tri-state.
3: If the Tri-State reset mode has been invoked, all outputs except SysClk will be tri-stated.

The following paragraphs detail which outputs are affected when the R3051 is in a tri-stated condition.

Pull-down/up Resistors on the A/D Bus

The R3051 tri-states the A/D bus when it finishes a write (or read) cycle and there is not another pending memory cycle that it needs to execute. This situation occurs when the R3051 is getting instructions from its internal instruction cache and it executes a sequence without store instructions. Since the A/D bus can be tri-stated for these periods, it is desirable for the input pins of the address latches and data transceivers to maintain the A/D bus with defined, valid logic values by using pull-up/pull-down resistors. The use of pull-up or pull-down resistors also has the benefit of easing Automatic Test Equipment programming on board-level and in-circuit tests.

Pull-down/up Resistors on Control Lines for DMA

The R3051 has an on-chip Direct Memory Access (DMA) arbiter that allows outside processors and controllers to take control of the external memory systems, and perform transactions. It does this by indicating a request to the R3051, which then tri-states its bus interface to allow it to be driven by the external agent.

During DMA, the R3051 will execute instructions from its internal caches until it has a cache miss, makes an uncacheable reference, or its write buffer becomes full.

An external agent requests bus mastership by asserting the R3051 BusReq input. If BusReq is asserted by the DMA device, the R3051 tri-states its outputs and asserts BusGnt to signal to the DMA device so that it can begin to drive its own memory cycles. During DMA, the R3051 tri-states all outputs except SysClk and BusGnt. During the time that the R3051 and the DMA controller transfer control back and forth, neither one drives the control line outputs (to avoid bus conflicts). In order to properly transfer control, the R3051 control outputs should be kept in their de-asserted state. If the transfer time is relatively short, the system designer may choose to rely on bus capacitance to hold these signals in their negated positions. Alternatively, a more conservative strategy is to hold the bus in a negated position with pull-down or pull-up resistors. Thus Rd, Wr, Burst/WrNear, and DataEn should use pull-up resistors and ALE should use a pull-down resistor.

Pull-down/up Resistors on Control Lines for Tri-State

The R3051 has a reset mode vector which allows the chip to tri-state all its outputs, except SysClk. This mode is attained by asserting Tri-State via SInt(1) while Reset is asserted. In addition to the control lines above, BusGnt is tri-stated. Thus for Automatic Test Equipment programming on board-level and in-circuit testing, a pull-up resistor for BusGnt can be used.

WAIT-STATE CONTROLLER LOGIC

Wait-states are used to extend the number of clocks within a memory transfer to provide sufficient memory access and data setup time for the particular type of memory being accessed. Such control can be provided with a wait-state controller state machine. In general, a wait-state machine has four steps:

1: Detect the beginning of a memory cycle
2: Determine the type of cycle:
   a: Which chip select (address decode)
   b: Read or write
   c: Single word or burst, write near or non-page write
3: Count out cycles until memory is ready and assert R3051 handshaking signals
4: Acknowledge the end the cycle

Thus, the basic control strategy is to use a counter which is held at zero until a cycle is started, and which then increments every clock cycle until the transfer is completed. This master counter then provides the reference by which control outputs to the memory, data path, and CPU are provided.

R3051's use of both Clock Edges

The R3051 uses both edges of the clock to assert and de-assert its control signals. This is to ameliorate the fixup time between memory cycles, which for most processors, takes 1 full clock cycle. The R3051 is able to do the fixup in 1/2 clock cycle. This would seem to complicate the design of state machines which must latch these signals synchronously to one edge or the other. However, as will be shown in the following sections, a traditional state machine that follows a small number of simple design rules can still use a single edge clock.

The R3051 uses an input clock, Clk2xIn, that runs at twice the frequency of the processor. The R3051 provides an output clock, SysClk, that runs at the same frequency as the processor and can be used to clock external state machines. The polarity of SysClk was chosen intentionally so that either an unbuffered SysClk or an inverted version of SysClk, (referred to here as SysClk) can be used. Because all the R3051 control outputs have very short propagation delays (less than 1/2 clock), a state machine can use either edge of SysClk.

In developing the set of constraints brought on by the use of both the rising and falling clock edges, some observations can be made:

1: All clockable control line outputs, except DataEn assert off the rising edge of SysClk.
2: All clockable control line outputs de-assert off the falling edge of SysClk.
3: All control line inputs required by the R3051 are sampled on the rising edge of SysClk.

Observations 1 and 2 can be specifically applied to two of the primary control signals, Rd and Wr.

1: Rd and Wr both assert off the rising edge of SysClk.
2: Rd and Wr both de-assert off the falling edge of SysClk.

The similarity of edge assertions for Rd and Wr can be used to simplify the wait-state controller.

Detecting the Beginning of a Memory Cycle

State machines looking for the beginning of a memory cycle can look for one of two things:

1: Rd or Wr asserting
2: ALE asserting

In general, state machines have to choose between using SysClk and SysClk. State machines such as those implemented in ASICs can use both clock edges, however, to simplify the discussion it will be assumed that only one or the other clocks is being used. If SysClk is used, certain registers must use SysClk directly from the processor to provide sufficient hold time from the processor. Only a negative edge clocked register can synchronously clock ALE under worst case timing, since ALE is only high surrounding the falling SysClk edge which requires a negative edge triggered flip-flop. SysClk cannot be used because its inverter delay will put it past when ALE could fall.

Machines which use SysClk (the inverted SysClk) will have a delay from inverting SysClk. All state machines can use Rd and Wr to determine the beginning of a cycle. SysClk machines are able to do this easily with wide margins on setup and hold times to its registers. SysClk machines must use SysClk directly from the processor and use registers with 0 hold time and also have a guaranteed minimum clock to output delay to meet the R3051's input hold time.

Determining the type of Memory Cycle

The type of memory cycle usually depends on the following variables:

1: Type of memory
2: Read or write cycle
3: Burst or non-burst, write near or non-page write

These three variables are usually logically AND'ed together to form equations for determining the number of wait-states before asserting RdCEn, Ack, or BusError as well as any transceiver controls. The chip selects from the memory decoder can be used to determine the type of memory to count

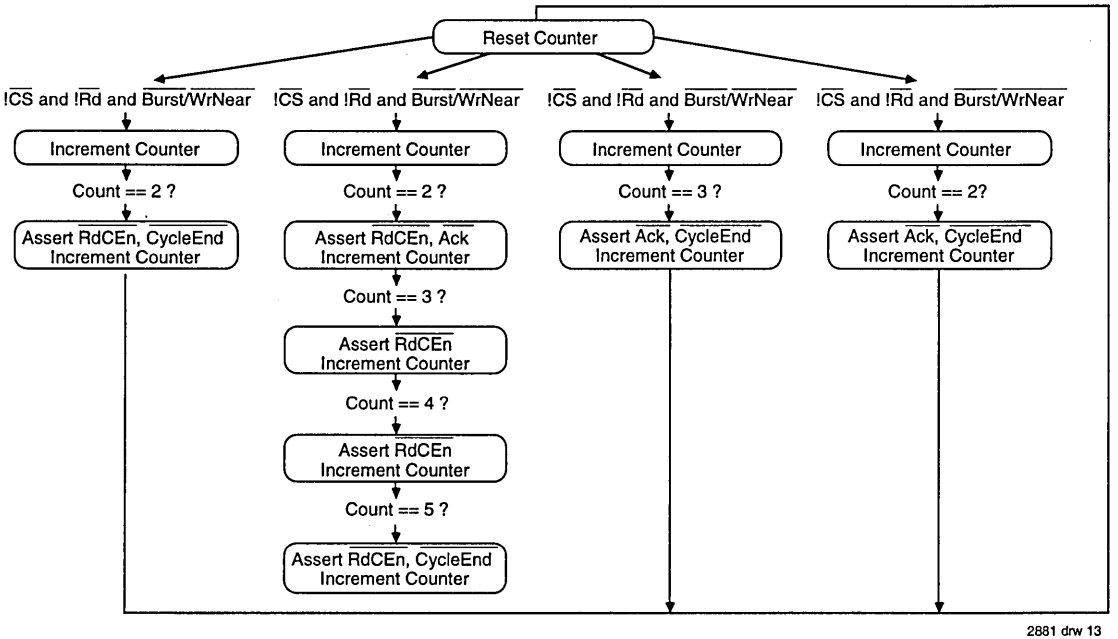


Figure 11. State Diagram of an Example Wait-State Controller for a Single Memory Type

the correct number of wait-states. By using the R3051's $\overline{Rd}$ and $\overline{Wr}$ lines, the transceiver controls can be defined. On read cycles, the R3051's $\overline{Burst/WrNear}$ line determines if 1 word or 4 words are to be returned. On write cycles, $\overline{Burst/WrNear}$ determines if a consecutive write is on the same 256 word page as its predecessor. An example of a state transition diagram that uses the read/write and burst/non-burst variables for one memory type is shown in Figure 11. Each memory type in the system also has a state diagram.

Further variables that affect the type of memory cycle are implied by the mode initialization vector which is supplied during processor reset initialization. The variables determine whether the data byte ordering is Big or Little Endian and whether data cache miss refills are handled one word at a time or as 4 word block refill reads. BigEndian and DBRefill are set by multiplexing the interrupt lines on the de-assertion of reset, an example of which is shown in Figure 12.

The mode vector of the R3051 was chosen to allow it to be supplied by just using pull-up resistors on the appropriate interrupt inputs. For example, the multiplexer shown in Figure 12 could be eliminated, and the pull-up resistors tied directly to the SInt(2:0) pins.

Note that to maintain compatibility with future versions of the R3051 family, $\overline{Int}$(5:3) should be high when $\overline{Reset}$ is de-asserted. This also can be performed using pull-up resistors.

Memory Interface Handshaking

The R3051 uses two inputs, $\overline{RdCEn}$ and $\overline{Ack}$,to indicate that the memory system is ready to receive or return data. On read cycles, $\overline{RdCEn}$ is sampled on the rising edge of $\overline{SysClk}$ by the R3051 so that it can enable its internal read buffer clock on the next falling edge of $\overline{SysClk}$. Thus on single word reads, a single $\overline{RdCEn}$ is asserted as the memory becomes ready as shown in Figures 2 and 11. On 4 word burst reads, $\overline{RdCEn}$ is asserted for each of the 4 words. Thus on burst reads, the wait-state controller can optionally "throttle" each word into the R3051 by delaying the return of each word by a varying number of clocks. $\overline{RdCEn}$ can be generated by gating the memory type and the count:

```
RdCEn not := Reset and CycleEnd and BusError and (
                (!RamCS and !Rd
                    and ( (Counter == 02H)
                        or(!Burst/WrNear and (Counter==03H))
                        or(!Burst/WrNear and (Counter==04H))
                        or(!Burst/WrNear and (Counter==05H))
                    )
                )
);
```

The acknowledge input, $\overline{Ack}$, has two uses. On burst reads, $\overline{Ack}$ can be used to optimize the processor execution engine restart. On writes, $\overline{Ack}$ is used to signal the end of the cycle, as will be explained later. The R3051 throttles burst reads into its internal read buffer at the rate of the memory system; however, it reads data from the read buffer on every clock cycle. Therefore, the R3051 will either wait until the 4th $\overline{RdCEn}$ has occurred to begin reading the internal read buffer, or until the memory system signals $\overline{Ack}$ to the processor. Asserting $\overline{Ack}$ on a burst read cycle causes the R3051 to start reading words from the read buffer in the next cycle; thus, the memory

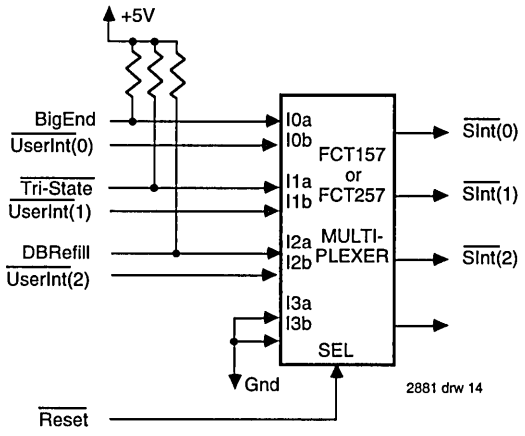


Figure 12. Reset Vector Circuit

system times the assertion of Ack so that the 4th word can be presented by the memory system just before it is read from the read buffer. Thus for optimal speed burst reads, Ack should be asserted 3 clocks before the last RdCEn occurs, as shown in Figure 3.

On write cycles, Ack is sampled on the rising edge of SysClk by the R3051 so that the cycle ends on the next falling edge of SysClk as shown in Figure 4. Ack is used by the wait-state controller on write cycles to acknowledge that data is being strobed into memory. Ack can be generated by gating the memory type and count.

Note that in writes, the WrNear output from the processor may also affect the write timing. For example, when writing to Page Mode DRAMs, it will be possible to retire near writes faster than non-near writes.

An example of generating Ack from gating the memory type and count is:

```
Ack not := Reset and CycleEnd and BusError and (
            (!RamCS and !Wr
                and ( ( Burst/WrNear and (Counter == 03H))
                    or (!Burst/WrNear and (Counter == 02H))
                )
            )
        or  (!RamCS and !Rd
                and ( (!Burst/WrNear and (Counter == 02H))
            )

);
```

Stopping the Counting

Four common ways to end the memory cycle and stop the counter include:

1: Use a SysClk state machine and look for the de-asserting edge of Rd or Wr

2: Use a SysClk state machine and gate the type of cycle into the counter to reset it independently of the de-asserting edge of Rd and Wr (predict the end of the cycle)

3: Use registers with asynchronous resets and gate Rd and Wr into the reset

4: Interlock a SysClk register looking for the asserting edge of Rd or Wr with a SysClk register looking for the de-asserting edge of Rd or Wr

In method 1, the SysClk registering of Rd or Wr is straightforward. However, if the counting is based on SysClk, the state machine will not be able to bring Ack or RdCEn low during the first possible clock cycle that they are sampled for by the R3051. This is, because the state machine will not detect the assertion of Rd or Wr in time. This implies that a SysClk based state machine will have a minimum of one or more wait-states.

In method 2, SysClk based state machines must determine when to stop counting independent of the de-assertion of Rd or Wr. In general they cannot use Rd or Wr to terminate the cycle because Rd or Wr may de-assert within the buffered (inverter delayed) SysClk register's setup or hold time. Thus SysClk based state machines should use its counter to determine when the cycle will end, e.g., with CycleEnd. CycleEnd or a similar signal uses the chip selects and a counter to determine the end of the memory cycle, without using the de-asserting edges of Rd and Wr. Logic equations for CycleEnd and the LSB of an N-bit binary up counter look like:

```
CycleEnd not := Reset and CycleEnd and (
                    (!RamCS and (Counter == 02H) and !Rd and Burst)
                    (!RamCS and (Counter == 05H) and !Rd and !Burst)
                    (!RamCS and (Counter == 03H) and !Wr and Burst)
                    (!RamCS and (Counter == 02H) and !Wr and !Burst)
                    ({Bus Error Timeout} (Counter == 0FH))
);

Counter(0) := Reset and CycleEnd and BusError and (!Rd or !Wr)
                                    and (Counter(0) xor 1)
;
```

A Timing Diagram of CycleEnd showing how CycleEnd asserting at the end of the memory cycle will reset the wait-state counter independently of Rd and Wr is shown in Figure 13.

Counters using CycleEnd use the type of cycle to determine when the wait-state counter should stop and reset independent of the de-asserting edge of Rd or Wr.

Wait-state machines implemented in ASICs can consider using method 4 which involves interlocking SysClk and SysClk based registers as shown in Figure 15. ASICs can also selectively combine two independent SysClk and SysClk state machines to avoid 1/2 cycle interlock timing constraints.

Bus Errors

Bus errors can be handled by timing out with the wait-state controller counter as it is about to overflow. For all types of memory cycles, the R3051 de-asserts its control edges, e.g.,

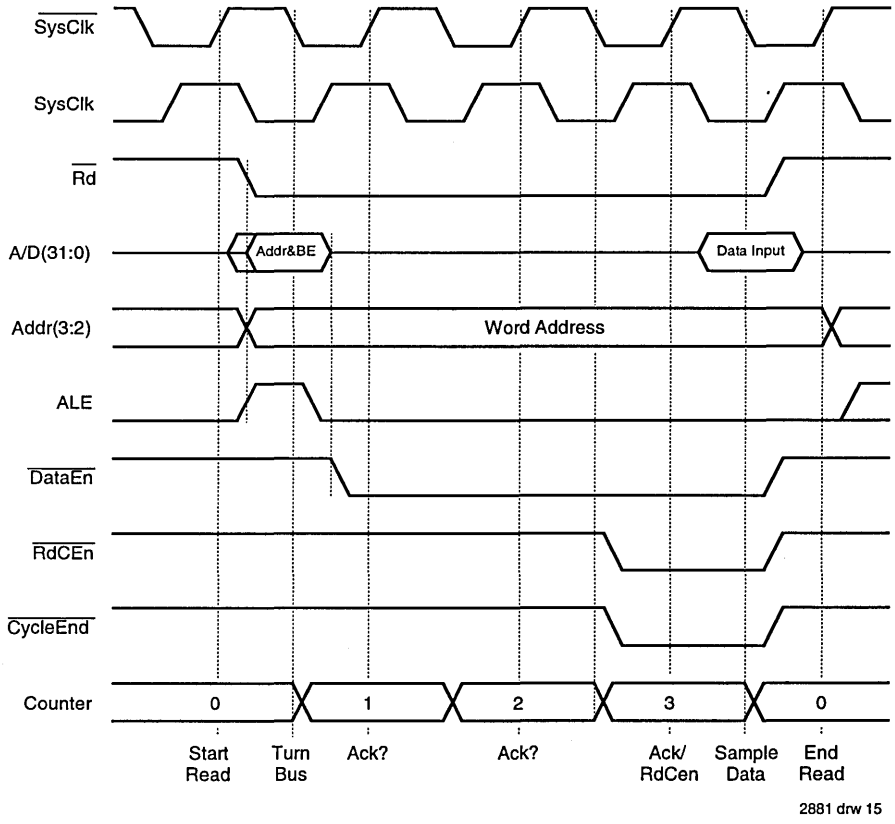


Figure 13. Timing Diagram of CycleEnd

Rd or Wr, on the clock following the assertion of BusError. SysClk based state machines can look for the de-asserting edge of Rd or Wr in order to reset the wait-state machine's counter. In SysClk based state machines, BusError can directly reset the wait-state machine's counter or the overflow count can be used to assert CycleEnd which will then reset the counter.

Bus errors signal an exception to the R3051 only if it is a read cycle. If exceptions need to be noted for write or DMA cycles, BusError should be gated into an interrupt line. The interrupt must be held until the R3051 can acknowledge it, since the R3051 re-registers its interrupt inputs on each clock cycle in which it is executing instructions in its run or fixup state.

READ ENABLES AND WRITE ENABLES

Memories and I/O devices have a combination of chip selects, read enables, and write enables to drive data out of the device and to strobe data into the device. Because the exact timing and functions of the selects, enables, and strobes differ for DRAM, SRAM, and I/O, this section discusses read and write enables and their relationship to the byte enables.

Read Enables

In general, a memory or I/O device has an output enable pin to enable its data outputs on a read cycle. Typical designs will address all 8-bit and 16-bit I/O devices using 32-bit word addressed, (i.e., use Addr(3:2) as their LSBs). Even though the R3051 produces byte enables on read cycles, it is rare to require use of the byte enables for reads as the R3051 will internally mask the bytes not being used. The output enable for the device can be derived from Rd or from DataEn.

If more than one memory device uses a single transceiver, it may be necessary to generate device Output Enables using a delayed version of DataEn. If one of the memory or I/O devices has a long output disable to tri-state time, then extra time must be allowed for that device to tri-state before another device is enabled. An equation determining if the read enables should be delayed on a back to back read cycle is:

tSysClk >= tDisableControl + tOldMemoryDisable - tNewMemoryData + tCap

The output enable control should be asserted at least until the clock cycle that Rd and DataEn de-assert to provide sufficient data hold time to the R3051.

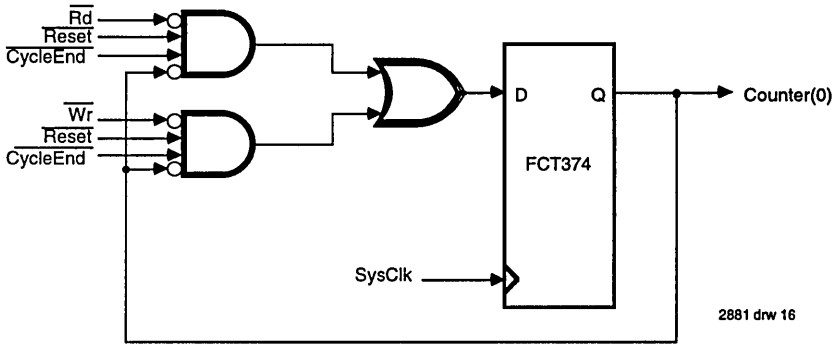


Figure 14. Using CycleEnd in a SysClk Based Counter

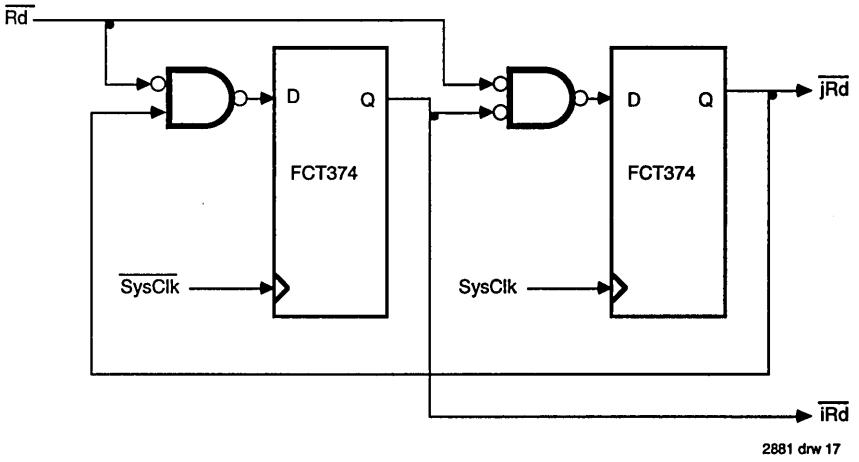


Figure 15. Using Interlocked Registers

Gating Write Enables and Byte Enables

Memory and I/O devices have a write enable pin or a similar protocol to strobe data into the device. A special case occurs for partial word stores, where only the pertinent bytes of a word have their byte enables asserted. Partial word stores occur when a store byte, store half-word, or store tri-byte instruction is executed. Because of the efficiency and optimization capabilities of modern compilers, such as the MIPS™ and IDT Compilers for the R3000™ family, the hardware must always assume that the software will make use of the partial word store instructions. Thus the write enables (or as shown earlier the chip selects) of each byte of a word must be gated with their respective byte enables. Gating the byte enables into the write enables can be done with an FCT157/257 multiplexer by configuring it as a set of four OR gates with a common input term as shown in Figure 16. The write enable signal can be derived from $\overline{Wr}$.

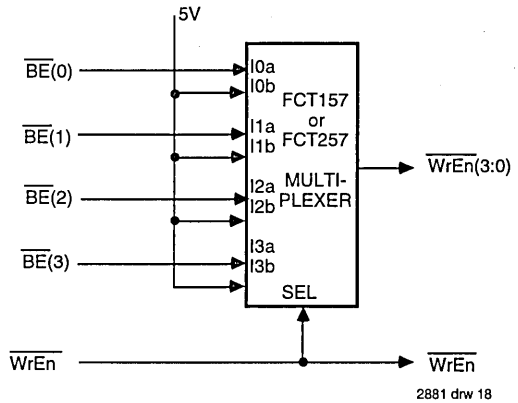


Figure 16. Gating Byte Enables into the Write Enables

SUMMARY

The main memory interface of the R3051 is conventional and simple. Basic blocks include address de-multiplexing, address decoding, data transceivers, wait-state controller, as well as the memory and I/O modules themselves. The R3051's uses both edges of the clock for control signals to reduce inter-cycle latency. Thus conventional wait-state controller algorithms can be used if the following guidelines are followed:

1: In $\overline{SysClk}$ based wait-state controllers, the input clock should be unbuffered from the processor's $\overline{SysClk}$ output. $\overline{SysClk}$ controllers will have a minimum of 1 or more wait-states. $\overline{SysClk}$ registers require small hold time and a minimum clock to output propagation delay to meet the R3051 input hold time.

2: In SysClk (inverted version of processor $\overline{SysClk}$ output) based wait-state controllers, the master reference counter must be reset independently of the de-asserting edges of $\overline{Rd}$ or $\overline{Wr}$. This can be done by gating the memory type and cycle type into a $\overline{CycleEnd}$ output which deterministically resets the counter.

The R3051's integration of an instruction cache, a data cache, read buffers, and write buffers allows simple main memory interfacing which can be implemented using a small amount of external logic. Thus the R3051 reduces the cost and board size of RISC processing, while maintaining very high throughput.

Integrated Device Technology, Inc.

INTERFACING THE R3051™ TO THE SONIC™

APPLICATION NOTE AN-95

by Danh Le Ngoc (Integrated Device Technology, Inc.) and Paul Cheng & Bill Harmon (National Semiconductor)

OVERVIEW

The IDT R3051™ family is a series of high-performance 32-bit microprocessors featuring a high-level integration and high-performance. The R3051 family integrates the MIPS R3000A™ RISC CPU, along with 8KB of instruction cache and 2KB of data cache. The R3051 family uses a simple time-multiplexed 32-bit address and data bus to provide a low cost system interface (and to minimize the cost of ASIC devices designed to interface with the processor). In order to minimize the impact of a time-multiplexed bus, the R3051 family incorporates a 4-deep read buffer and 4-deep write buffer into the interface, allowing relatively slow memory systems to be mated to a high-speed processor. The R3051 family is able to offer 35 MIPS of integer performance at 40MHz without requiring external SRAM or caches.

The R3051 family is designed to bring the high-performance inherent in the MIPS RISC architecture into low cost simplified embedded applications such as laser printers, X-Window terminals and network bridges and routers. Figure 1 illustrates the simplified block diagram of the R3051-based X-Window terminal.

The focus of this application note to describe the interface between the R3051 and National Semiconductor's System Oriented Network Interface Controller(SONIC).

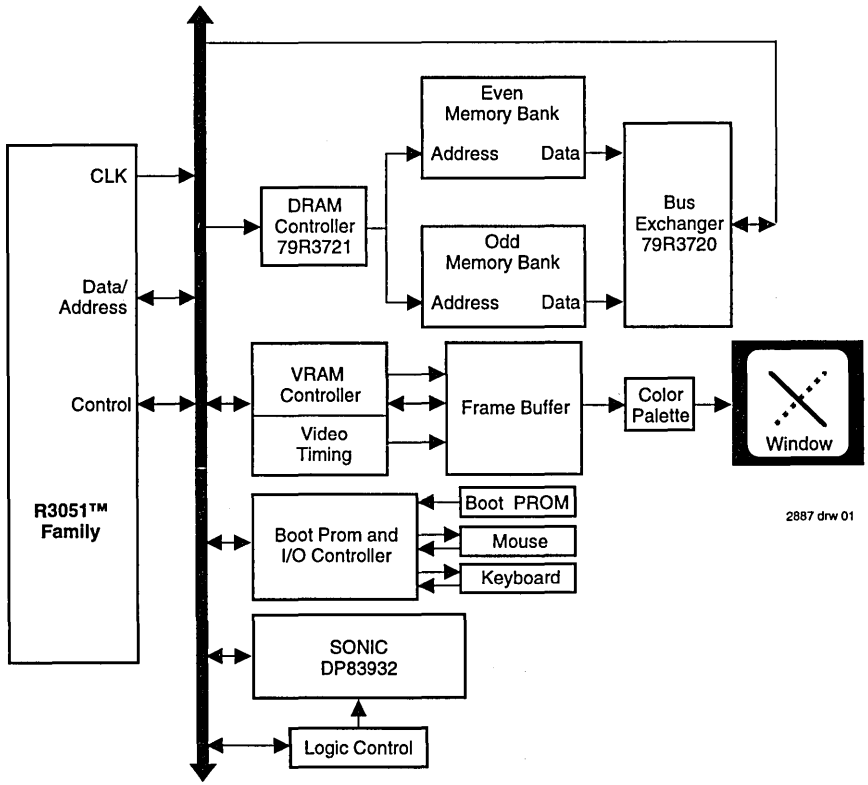


Figure 1. X-Window Terminal

SONIC is a Trademark of National Semiconductor
RISController is a Trademark of Integrated Device Technology

The SONIC™ is National Semiconductor's System Oriented Network Interface Controller (DP83932). This Ethernet controller is intended to provide a high performance 32 or 16-bit Ethernet connection for systems that require efficient, high throughput, low power network connectivity. The SONIC can be employed in an R3051-based system, in order to tightly couple the system's CPU and main memory to the network. Figure 2 depicts this interface.

The SONIC is ideally suited to embedded processing applications such as X-Terminals, due to its unique feature set. The SONIC completely supports all the required specifications set forth in the IEEE 802.3 standard, including the Media Access Control (MAC) requirements contained in the IEEE 802.3 layer management specification. Additionally, SONIC's high performance DMA channels allow it to use a very small percentage of the bus bandwidth, while its efficient linked list buffer management scheme limits the number of descriptor and data fetches required. It is also important to note that the SONIC utilizes internal content addressable memory (CAM) to provide a 100% perfect address filter for both multicast and physical address packets. This alleviates the need to waste bus bandwidth, memory space, and CPU time on unwanted packets. Finally, the SONIC contains an integrated Manchester encoder/decoder, which is required in all Ethernet applications. This provides a savings in board space, as well as improved reliability.

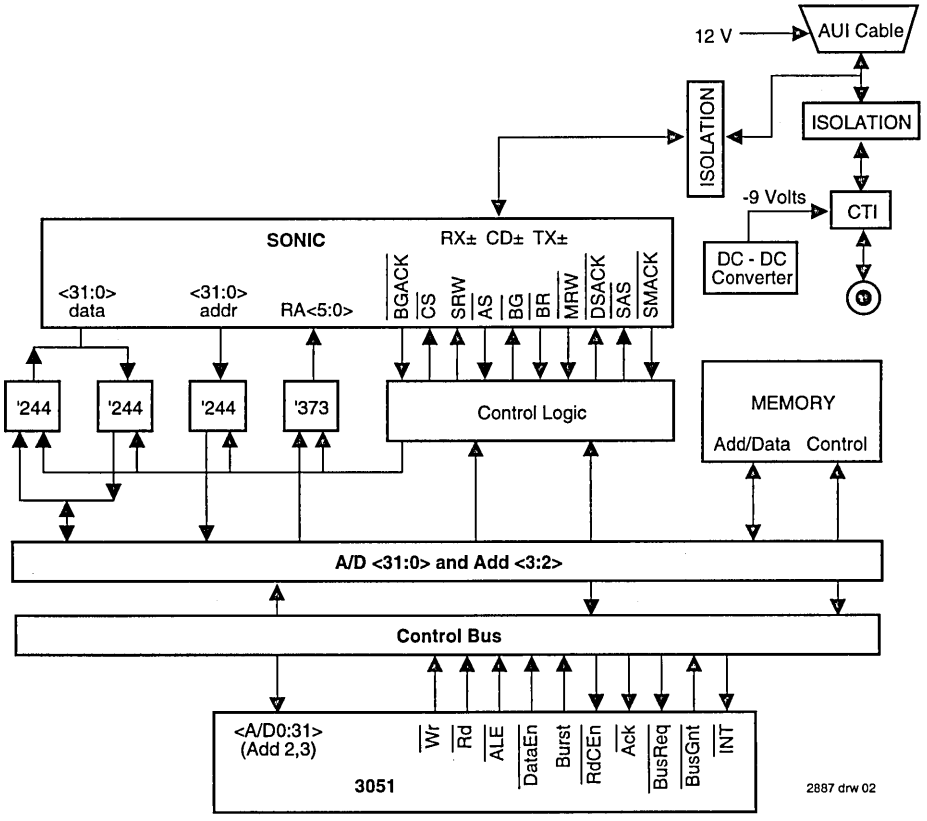


Figure 2. SONIC Interface to the R3051

FUNCTIONAL OVERVIEW

System Interface

The R3051 has a multiplexed 32-bit address and data bus. Since the SONIC's address and data buses are demultiplexed, it is necessary to employ a set of external latches to connect the SONIC to the processor's address and data buses. In many applications, these latches may also be used to demultiplex the R3051 bus to other parts of the system memory and I/O.

In order to allow the R3051 to have access to the SONIC's internal registers, as well as allow the SONIC to gain control of the system bus and perform DMA operations, the SONIC is interfaced to the system bus as both a slave and a master. As a slave, the SONIC appears as a block of 256 bytes, consisting of sixty-four 32 bit words. The SONIC can be mapped into any location of memory and will typically provide for a 7 cycle register access. In R3051 applications, the SONIC will typically be mapped into the processor kseg1, which is an unmapped, uncached address space typically used for processor I/O resources.

As a master, the SONIC will arbitrate with the R3051 for ownership of the bus and proceed to operate as a 32-bit DMA engine between the network and the system memory. While operating on the bus, the SONIC is capable of performing 32-bit/3 cycle DMA operations. It is important to note that the ability to place the SONIC on the same bus as the R3051 and the system memory is critical: this eliminates the need for the Ethernet controller to have a local buffer, which the CPU must spend time and bandwidth to transfer to main memory. The ability of the SONIC to place data directly in main memory and communicate with the CPU through linked list descriptors, as well as register accesses, makes the SONIC/R3051 interface CPU and bandwidth efficient.

Network Interface

With respect to the physical layer design, both AUI drop cable Ethernet and thin wire Ethernet are supported. The block diagram in Figure 2 contains a 15 pin AUI drop cable connector for standard drop cable Ethernet implementations, as well as a thin wire Ethernet connection via the National Semiconductor coaxial transceiver interface (CTI, DP8392). Either of these network connections can be chosen through the use of a single jumper between the 5 volt supply and the 5 volt to –9 volt DC-to-DC converter. In either case, the AUI signals (RX±, TX±, and CD±) are sent back to the SONIC. These signals are interfaced to the ENDEC portion of the SONIC, which provides for communication between the AUI interface and the non-return to zero (NRZ) signals (RXD,TXD, and COL) of the Media Access Control (MAC) module of the SONIC. It should be noted that the integrated ENDEC module of the SONIC alleviates the need for an external Ethernet Manchester encoder/decoder, such as National's CMOS Serial Network Interface (CMOS SNI, DP83910).

ARCHITECTURE AND DESIGN

Bus Interface

The SONIC's bus interface can be externally configured to operate in one of two modes. If the SONIC's BMODE pin is tied to ground, the SONIC will operate on the bus exactly like an 80386 microprocessor. If the SONIC's BMODE pin is tied to 5 volts, the SONIC will operate on the bus exactly like a 68030 microprocessor. In this design, the most appropriate mode of operation was achieved by connecting BMODE to 5 volts.

The bus interface, as depicted in Figure 3, consists of 2 parts. There is an address bus interface and a data bus interface. Since the R3051's address and data buses are multiplexed, it is necessary to utilize a set of '244 buffers and '373 latches to multiplex the SONIC busses onto the CPU bus. The '244 buffers are required to tri-state the SONIC's address lines from the system bus during the data portion of master transfers, while the '373 is required to latch the register addresses being sent to the SONIC during slave operations. The output enable signal of the '244 is asserted when the SONIC is the master of the bus and both the SONIC's address strobe ($\overline{AS}$) is asserted and the master logic's address latch enable (ALE) signal is asserted. The '373 should latch the address when the R3051 is the bus master and it asserts its ALE signal.

The data bus interface requires the use of 2 sets of '244 buffers. The first set of buffers (Buffer 1) prevent the SONIC from placing data onto the system's multiplexed address and data bus prematurely. In the slave mode of operation, the output buffer is enabled once the address output drivers are tri-stated. This is signaled by the assertion of the DataEn signal. In the case of a master operation, the buffers are enabled once the address buffers external to the SONIC are tri-stated, which takes place upon the deassertion of the ALE signal.

The second set of buffers is enabled when the SONIC's registers are being written by the R3051 and data is being presented on the multiplexed system address/data bus, or when the SONIC is reading system memory and the memory is placing data on the multiplexed address / data bus. The assertion of the DataEn signal by the system signals that data is now able to be placed on the bus. The actual logic representation for the bus interface can be found in the bus interface logic segment of the Control Logic section of this application note.

Slave Operation

The timing diagram for a slave access of the SONIC is shown in Figure 4. The falling edge of the R3051's ALE signal latches the output of an address decoder and the address lines being passed to the register address lines of the SONIC. If the address decode selects the SONIC, a signal called "AdrDec" will be asserted. The logic for generating this signal

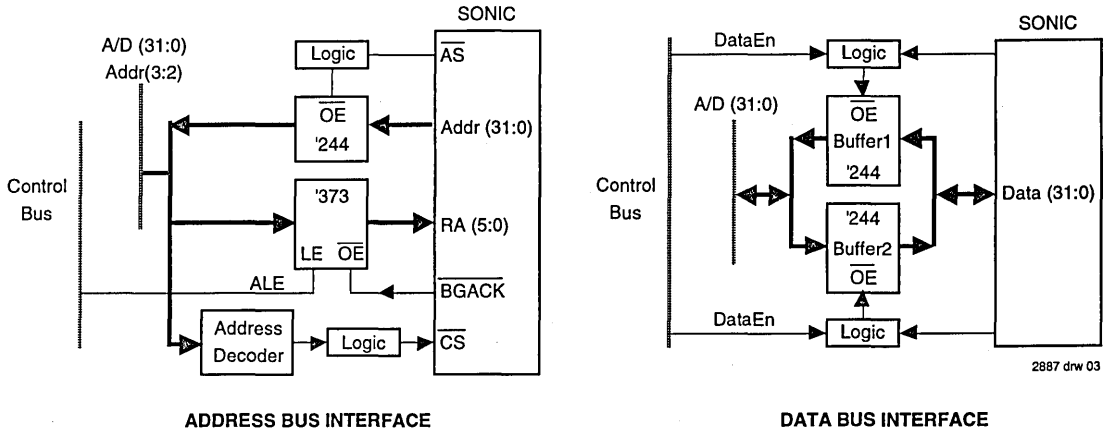


Figure 3. Address and Data Bus Interface

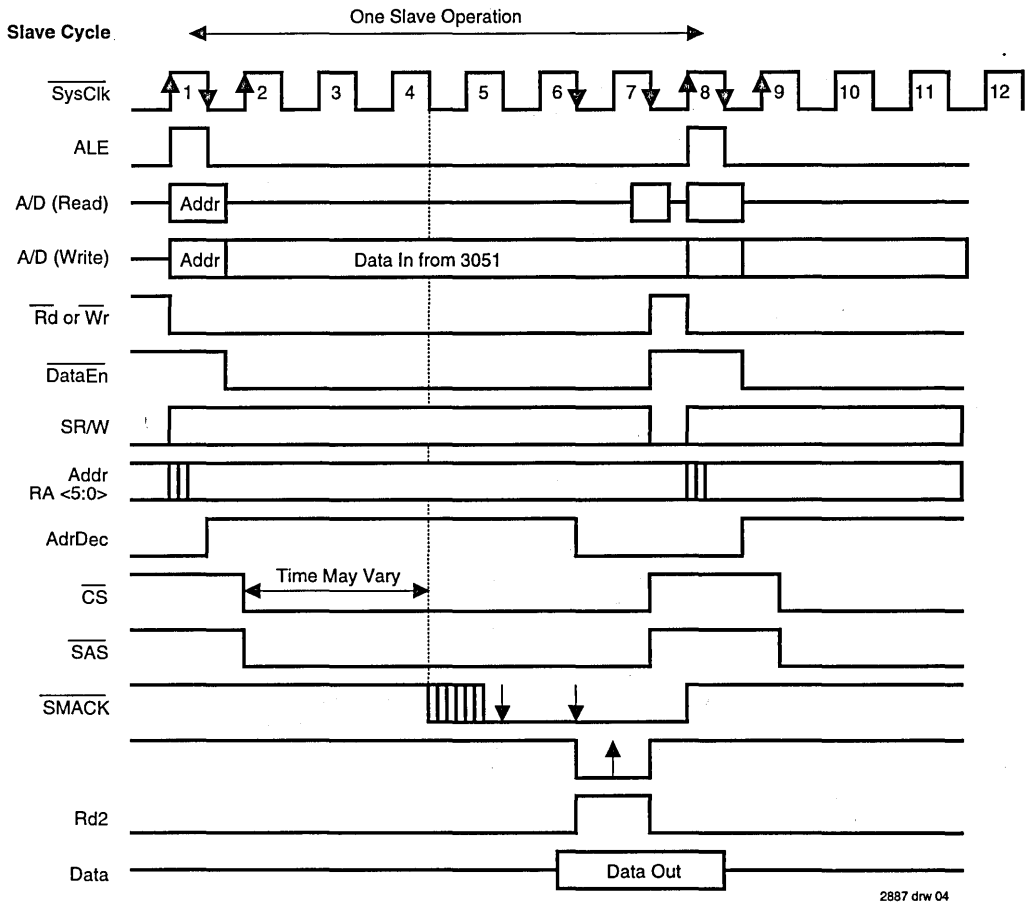


Figure 4. Slave Access Timing Diagram

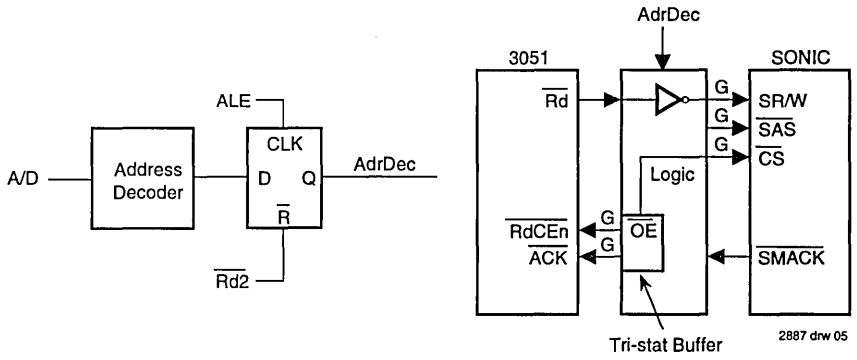


Figure 5. Slave Interface Block Diagram

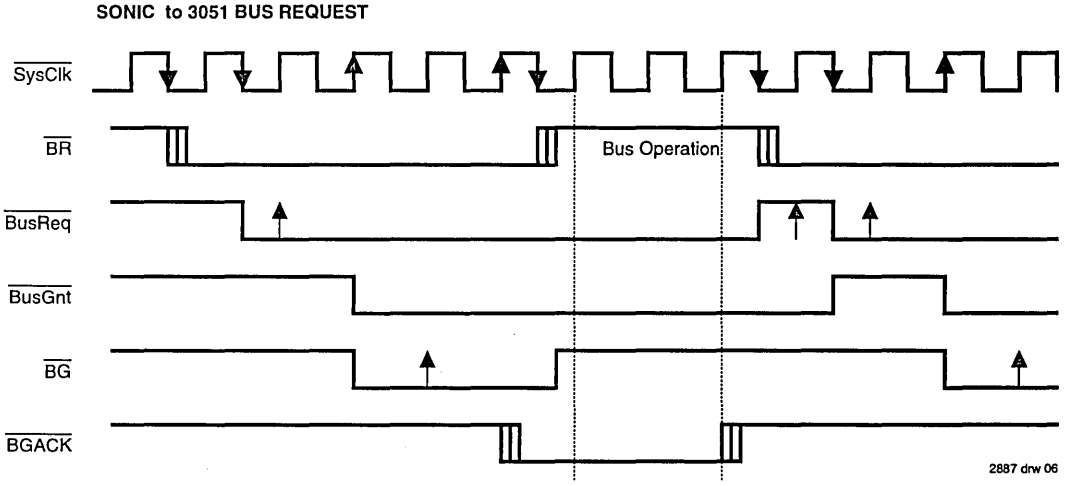


Figure 6. Bus Request Timing Diagram

is shown in Figure 5. The value of this signal is passed to the chip select (CS) and slave address strobe (SAS) signals of the SONIC on the rising edge of the bus clock. The acknowledge signals back to the R3051 (ACK for a write and RdCEn for a read) are asserted 2 clocks after the SONIC generates its slave acknowledge signal (SMACK). These signals remain asserted to the R3051 for a clock cycle, after which they are removed. The ACK and RdCEn signals inform the R3051 that the data has been latched or is valid, respectively. The deassertion of these signals results in the deassertion of CS and SAS to the SONIC. The logic for implementing this part of the design can be found in the slave logic segment of the Control Logic section.

Master Operation

The first step in designing the master interface is implementing the bus request logic. The timing diagram for this is shown in Figure 6. The bus request (BR) signal of the SONIC is passed to the R3051's bus request (BusReq) on the falling edge of the bus clock. The SONIC then waits for the bus grant (BusGnt) from the R3051, which is passed directly to the SONIC's bus grant (BG) signal. The assertion of BG causes the SONIC to assert bus grant acknowledge (BGACK) and begin its master DMA operations. It is important to note that the assertion of BGACK causes the SONIC to deassert BR, which would cause the bus request logic to deassert BG to the SONIC. Thus, the BusReq signal to the R3051 should be the logical "OR" of the SONIC BR and BGAck outputs. A block diagram of the bus request logic appears in Figure 7, while the actual illustration of the logic is found in the bus request logic segment of the Control Logic section.

Once the SONIC has gained control of the bus, it will begin to perform master DMA operations, as illustrated in the Figure 9 timing diagram. Ideally, if the memory is fast enough, the SONIC will be able to perform 3 cycle DMA. At 25 MHz, less than 3.75% of the bus' bandwidth will be consumed by the network interface.

There are two very important points to note. First, the R3051's ACK signal is basically equivalent to the SONIC's DSACK signals, but the SONIC's DSACK signals require that

the memory system provide a total of 8 ns hold time from the rising edge of the clock, while the R3051 requires only 4 ns. Second, the ALE signal generated from the SONIC's control signals will be deasserted 3 ns later than the R3051's would be. However, this should not be a significant factor, since the address set-up and hold time provided to the memory system's latches is consistent with the R3051's specification.

When interfacing to the multiplexed bus, it is necessary for the master logic to generate an ALE signal for the system bus. The ALE signal is asserted on the rising edge of the second cycle in the SONIC's memory access. It is necessary to assert the ALE in this cycle, in order to guarantee that the latch will be provided with an adequate amount of set-up time for the address. The ALE signal is then removed on the falling edge of the same clock cycle. The deassertion of ALE triggers the assertion of $\overline{\text{DataEn}}$ on a read operation, in order to inform the memory that the bus' address drivers are tri-stated and data can now be driven. The $\overline{\text{DataEn}}$ signal is actually arrived at by delaying the the ALE signal through a buffer or PAL, since the ALE signal is also responsible for disabling the output buffers of the address drivers.

The final piece of interface logic is used to make the SONIC's read and write (MR/$\overline{\text{W}}$) strobe compatible with the R3051's read ($\overline{\text{Rd}}$) and write ($\overline{\text{Wr}}$) signals. The SONIC's read/write signal is passed to the appropriate read or write strobe

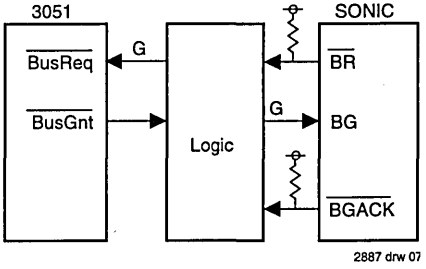


Figure 7. Bus Request Interface Block Diagram

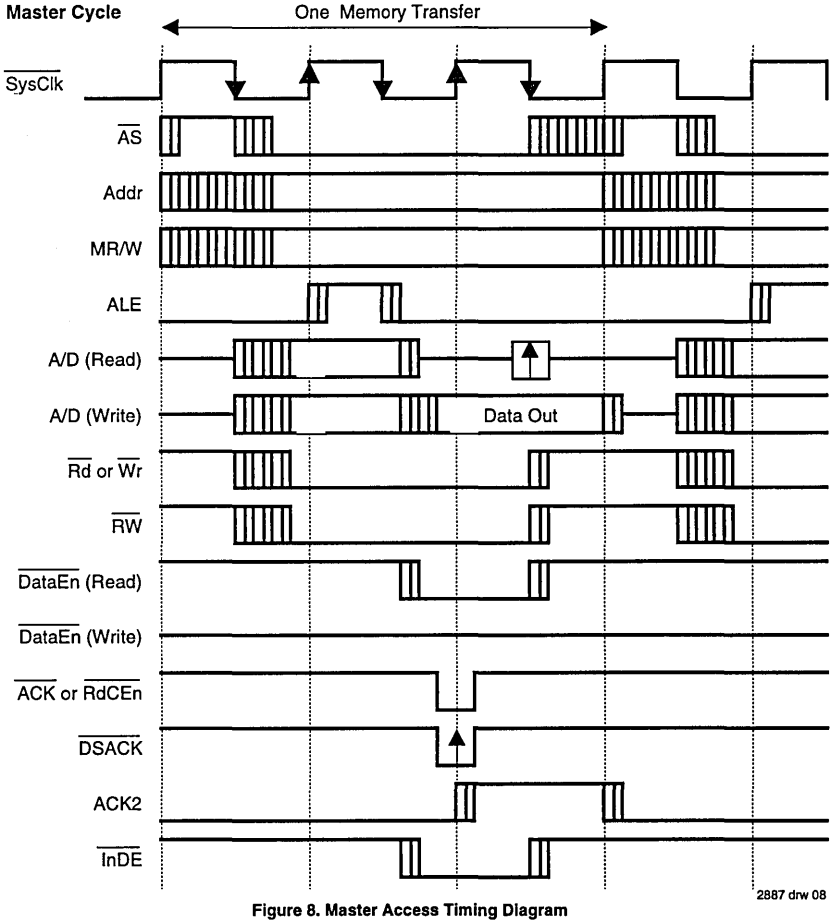


Figure 8. Master Access Timing Diagram

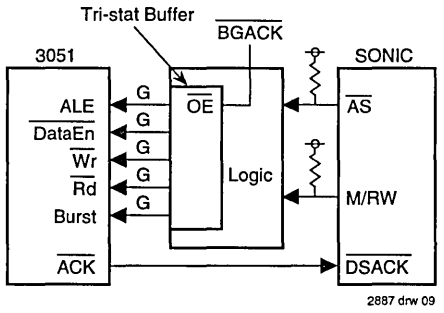


Figure 9. Master Interface Block Diagram

of the system bus, on the falling edge of $\overline{AS}$. The $\overline{Rd}$ or $\overline{Wr}$ signal is then deasserted on the falling edge of the last clock cycle. The block diagram for the master interface is found in Figure 9, while the logical implementation is shown in the master interface logic segment of the Control Logic section.

Physical Layer

Figure 10 contains a block diagram of the physical layer interface, while a schematic of the physical layer design is located on the last page of this application note. This design can be used in either a thin wire or standard drop cable Ethernet environment. When the design is used in a thin wire Ethernet application, the 5 volt supply must be connected to the DC-to-DC converter, so that the necessary –9 volt output can be supplied to National Semiconductor's Coaxial Transceiver Interface (CTI, DP8392). The CTI provides an interface between the 10 MHz Manchester encoded coax cable and the 10 MHz Manchester encoded differential signals of the SONIC's ENDEC. In the case of a standard drop cable Ethernet application, the 5 volt supply is left unconnected, so that the CTI will not receive power. This allows the signals of the SONIC's ENDEC to pass directly to the AUI cable, via the 15 pin AUI connector. In examining the schematic of the physical layer design, it can be seen that there is a pulse transformer at the AUI side of the CTI. This is placed here to isolate the CTI from the SONIC's ENDEC signals, when the AUI drop cable connection is being employed. This transformer also provides the IEEE 802.3 specified isolation between the coax and the differential AUI signals, when thin wire Ethernet is being used. It is also necessary to provide a termination for the 78Ω AUI cable's differential receive and collision pair (RX± and CD ±). This is the reason for the 39Ω –1%resistors and .01µF capacitors that are shown in Figure 10.

Additionally, there are 2 more significant considerations. First, each one of the transmit pairs (TX+ and TX-) requires a 270Ω non-precision pull down resistor to complete the internal source follower amplifiers that drive these signals. Second, there is an isolation transformer placed between the differential signals of the SONIC's ENDEC and the AUI cable. This isolation is necessary to guarantee that the SONIC meets the IEEE 802.3 fail safe specification of a 16V DC level appearing on the AUI cable's differential signals. This external isolation is necessary, because in the powered down state the CMOS process, in which the SONIC is manufactured, may not be able to withstand this voltage.

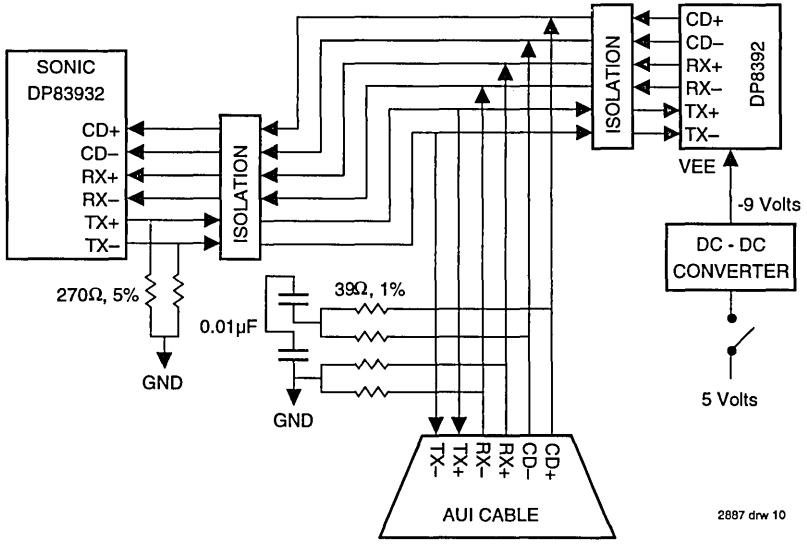


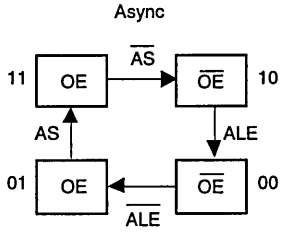
Figure 10. Physical Layer Interface Block Diagram

Control Logic

This application note was developed with the intention of displaying the necessary requirements for interfacing the SONIC to the R3051 system bus. Therefore, the actual implementation of the control logic will be graphically depicted in state machine form, as opposed to being partitioned into actual PAL devices. This leaves the freedom for the designer to incorporate this logic into his / her system in PALs, ASICs, FPGAs, etc.

BUS INTERFACE LOGIC

Address



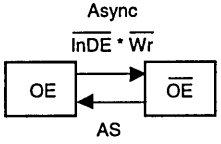
$Q1^{\wedge} = Q1 * Q2 + Q1 * \overline{ALE} + Q2 * AS$

$Q2^{\wedge} = \overline{Q1} * Q2 + \overline{Q1} * \overline{ALE} + Q2 * AS$

$OE = Q2^{\wedge}$

Data

$$OE = \underbrace{(\overline{DataEn} + CS)}_{\text{1st case}} + \underbrace{(BGACK + DataEn)}_{\text{2nd case}}$$



$OE1 = DataEn + CS$

$OE2 = Q^{\wedge} = AS * InDE + AS * \overline{Q} + Wr * Q + InDE * Q$

$$OE = \underbrace{DataEn + CS}_{\text{case 1}} + \underbrace{AS * InDE + AS * OE2 + Wr * \overline{OE2} + InDE * OE2}_{\text{case 2}}$$

2887 drw 11

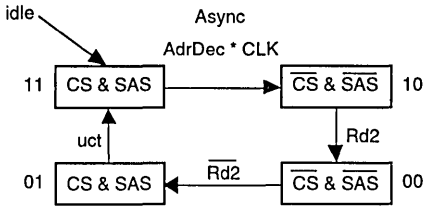
Note:
1. Q1^ refers to the first state machine bit and Q2^ refers to the second state machine bit (10: Q1^=1 & Q2^ = 0)

SLAVE INTERFACE LOGIC

$Q1^ = Q2 + Q1 * \overline{Q2} * \overline{Rd2}$

$Q2^ = Q1 * Q2 * \overline{AdrDec} + \overline{Q1} * \overline{Q2} * \overline{Rd2} + \overline{Q1} * Q2$

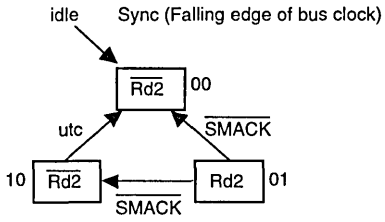
$CS = SAS = Q2^$

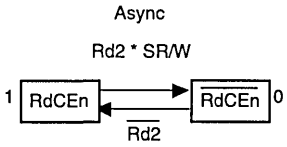


$Q1^ = \overline{SMACK} * Q2$

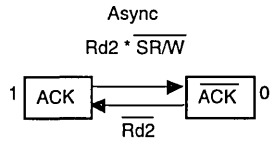
$Q2^ = \overline{Q1} * \overline{Q2} * \overline{SMACK} + Q2 * SMACK$

$Rd2 = Q2^$





$RdCEn = Q^ = \overline{Rd2} + Q * \overline{SR/W}$



$ACK = Q^ = \overline{Rd2} + Q * SR/W$

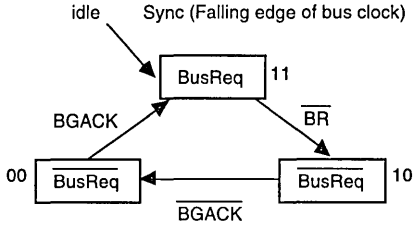
2887 drw 12

BUS REQUEST INTERFACE LOGIC

$Q1^ = BGACK + Q1 * Q2$

$Q2^ = BR * Q2 + BGACK * \overline{Q1}$

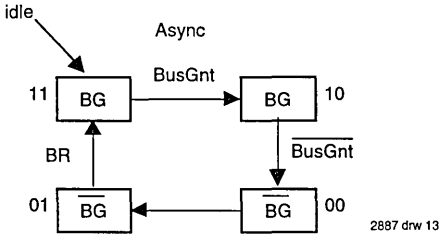
$BusReq = Q1^ * Q2^$



$Q1^ = Q1 * Q2 + Q2 * BR$

$Q2^ = \overline{Q1} + Q2 * \overline{BusGnt}$

$BG = Q1^$



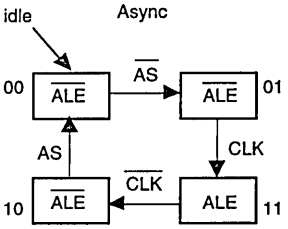
2887 drw 13

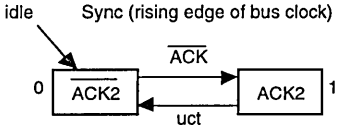
MASTER INTERFACE LOGIC

$Q1^{\wedge} = Q1 * \overline{AS} + Q2 * CLK + Q1 * Q2$

$Q2^{\wedge} = \overline{Q1} * \overline{AS} + \overline{Q1} * Q2 + Q2 * CLK$

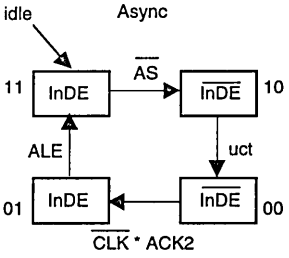
$ALE = Q1^{\wedge} * Q2^{\wedge}$

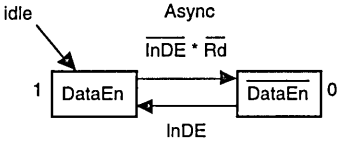




$Q^{\wedge} = \overline{Q} * \overline{ACK}$

$ACK2 = Q^{\wedge}$





$Q^{\wedge} = InDE + Rd * Q$

$DataEn = Q^{\wedge}$

$Q1^{\wedge} = Q1 * Q2 + \overline{Q1} * Q2 * ALE$

$Q2^{\wedge} = \overline{Q1} * Q2 + \overline{Q1} * \overline{Q2} * \overline{CLK} * ACK2$

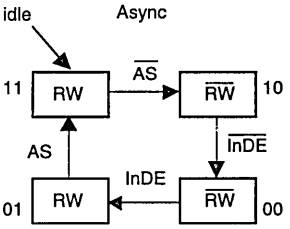
$+ Q1 * Q2 * ALE$

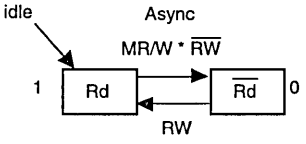
$InDE = Q2^{\wedge}$

$Q1^{\wedge} = Q1 * Q2 + Q1 * InDE + AS * Q2$

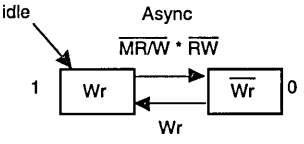
$Q2^{\wedge} = InDE * \overline{Q1} + \overline{Q1} * Q2 + Q2 * AS$

$RW = Q2^{\wedge}$





$Rd = Q^{\wedge} = RW + Q * \overline{MR/W}$



$Wr = Q^{\wedge} = RW + Q * MR/W$

2887 drw 14

Integrated Device Technology, Inc.

IDT79R3051™ ADDRESS/DATA BUS TURN AROUND BEHAVIOR

APPLICATION NOTE AN-97

by Andrew Ng

INTRODUCTION

This application note describes the behavior of the R3051's multiplexed Address/Data, "A/D" bus and presents the issues of a particular topic called "Bus Turn Around." Bus Turn Around will be defined, design issues will be presented, and design solutions will be given for conventional R3051 systems, as well as a "DMA $\overline{\text{BusReq}}$" design solution for very low speed and very high speed systems.

Definition of the R3051

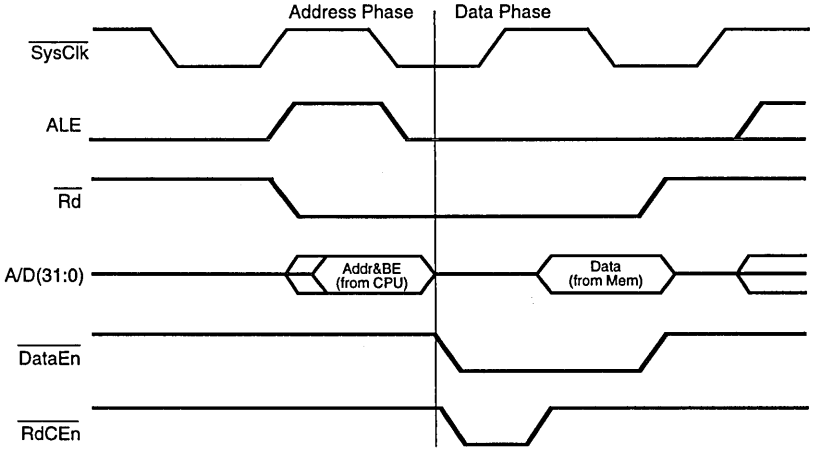
The IDT79R3051™ RISController™ is a highly integrated MIPS™ R3000™ instruction set compatible microprocessor that minimizes system cost and power consumption. The R3051 includes 4kB to 8kB of instruction cache, 2kB of data cache, an optional on-chip TLB memory management unit, 4-deep read and write buffers, on-chip DMA arbitration, a simple external bus interface, as well as the R3000A CPU execution engine — all in a single compact plastic 84-pin package.

Definition of the A/D Bus

One of the key features of the R3051 is its low pin count. The low pin count is largely a result of its simple control interface and its use of a multiplexed Address and Data bus, called A/D(31:0). As shown in Figures 1 and 2, the multiplexed A/D bus drives its address during the first phase of a read or write memory cycle. In the 2nd phase of a read memory cycle, the CPU expects the external memory system to drive the bus and return the data. In the 2nd phase of a write memory cycle, the CPU drives the data out to the memory system. Thus in a typical R3051 system, the address can be latched using a bank of transparent latches such as with the 54/74FCT373T or 54/74FCT841T as shown in Figures 4 and 5 so that the address is de-multiplexed from the data lines.

In systems using an ASIC, such as for a DRAM or DMA Controller or as an Integrated I/O Subsystem/Controller with on-chip programmable registers, the multiplexed A/D bus has an advantage over separate Address and Data busses in that the ASIC requires substantially fewer pins. The ASIC can latch the 32 Address bits internally, using the Address Latch Enable output from the CPU called "ALE", and then use the same input pins to provide data. In addition, the CPU has less noise from simultaneous switching of the 32 A/D lines than if it had to switch 64 separate Address and Data lines. Thus R3051 systems can often save cost and space by using inexpensive and low pin count ASICs.

Although a multiplexed bus may be thought of as a disadvantage in terms of system performance, this is rarely the case in R3051 systems. An analysis of memory behavior and the bus shows that in conventional memory systems (those that do not use exclusively high-speed, single cycle SRAMs for the entire memory system), the R3051 bus structure causes no real performance loss.



2531 drw 01

Figure 1. R3051 Read Cycle

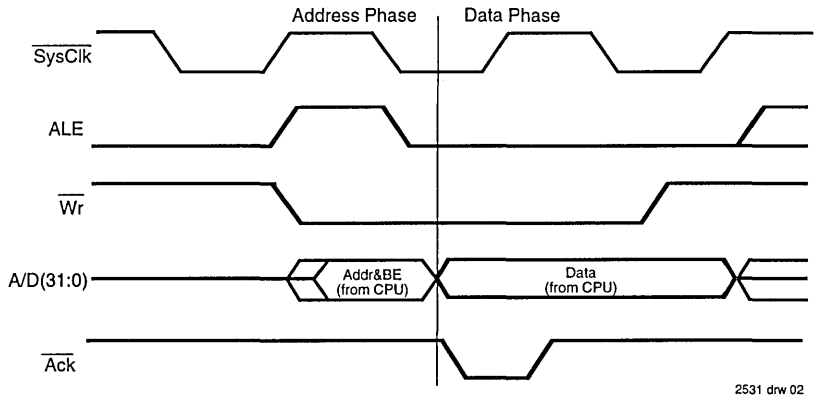


Figure 2. R3051 Write Cycle

For example, conventional memory systems use the address before the data is generated on read cycles or needed by write cycles. On read cycles, the address is always needed before the data array can be accessed. The multiplexed R3051 bus provides the address as early as a non-multiplexed bus would; thus, the read access is not delayed. Since memory read performance is described as "Address and Chip-Select valid to Data Available", the multiplexed bus causes no performance loss on reads.

Similarly, on write cycles, most memories (except for self-timed memories) require the address before the data in order to properly coordinate the write strobe with the correct internal row and column address decode/selects. The R3051 bus provides the write target address for one-half cycle, and then immediately presents the write data. That half cycle is required to perform address decoding, and to provide a Chip-Select to the memory device. Thus, once the address and Chip-Select are available to the memory, the data is also available.

Further, the R3051 decouples the system bus performance from processor performance based on the integration of on-chip resources. Specifically, the large on-chip caches minimize the number of main memory reads, thus making system read performance less criticial. The on-chip 4-deep write buffer isolates the processor from the memory system write speed, allowing it to continue execution while store operations are actually updated into the memory. Thus, R3051 performance, while somewhat dependant on memory system performance, is largely isolated from the memory system. Thus, high-performance systems using relatively slow EPROM and DRAM devices can be easily realized.

Definition of Bus Turn Around

The other consequence of a multiplexed bus arises from the fact that during a particular transaction, as well as from one transaction to the next, transitions between sources of the bus can occur. For example, a read transaction begins with the processor driving the address on the bus, and ends with the memory driving the data on the bus. Similarly, at the end of a read, the next transaction on the bus will begin again with the CPU driving an address on the bus.

Note that similar concerns are present even for non-multiplexed busses. For example, a read followed by a write results in the data bus first being driven by the memory, and then being driven by the CPU. Thus, bus turn-around is also a consideration in non-multiplexed bus systems.

Bus Turn Around behavior is the action that the CPU takes when its address/data bus transitions between the CPU and the memory, particularly when it changes direction from being a driver to being a non-driver or vice-versa. The actions that the CPU can take are:

1. Drive the address.
2. Drive the data.
3. Tri-state.

There are two basic times when the A/D bus will transition:

1. Intra-Cycle — Within a memory cycle as the address phase transitions into the data phase.
2. Inter-Cycle — Between two memory cycles when the data phase transitions into the address phase of the next memory cycle.

Intra-Cycle Bus Turn Around

A typical case of an address to data transition happens during a read cycle. As shown in Figure 1, when the Address Latch Enable (ALE) is negated, the address is externally latched and the CPU turns the bus around by tri-stating the A/D bus, so that the external memory system can begin to drive the expected data back to the CPU. The second case occurs during write cycles when the CPU finishes driving the address, it begins driving the data to the memory system. Since the CPU drives both the address and data during write cycles, bus turn around is not a significant issue during write cycles. The two intra-cycle transition cases are listed in Table 1, which shows the state of the CPU A/D output buffers during the address and data phases of the transaction.

Note that the processor provides an output, $\overline{\text{DataEn}}$, to indicate that this transaction has occurred. During the addressing phase, $\overline{\text{DataEn}}$ is negated, indicating the CPU is driving the A/D bus. During the Data Phase, $\overline{\text{DataEn}}$ is asserted, indicating that the bus is to be driven by the external memory system. During write cycles, and during idle cycles, $\overline{\text{DataEn}}$ is guaranteed to be negated, indicating that the external memory system should not be driving the A/D bus.

READ	A,Z
WRITE	A,D

Note: A — Address, D — Data, Z — Tri-State

Table 1. R3051 Address to Data Bus Transitional Behavior Within Memory Cycles

Inter-Cycle Bus Turn Around

A typical case of the transition between two memory cycles occurs on a read cycle that is immediately followed by a write cycle as shown in Figure 3. In this case, the memory system is required to turn the bus around by tri-stating the bus before the next write cycle begins to drive its address onto the A/D lines. Table 2 lists the R3051's behavior on each of the cases of inter-cycle memory transitions. The table lists the state of the CPU output buffers at the end of the first transaction, followed by the state of the buffers at the beginning of the next transaction. Note that if a read or write cycle occurs while the CPU is executing instructions from its internal cache, the next external memory cycle might not occur until many clocks later, in which case the A/D bus is tri-stated since it is idle. Also, many of the cases, such as the transitions after writes have both the data and address driven by the CPU. Thus bus turn around is not a significant issue after write cycles. Other transitions may not actually be possible. For example, it is impossible to have a read followed by a read. At least one idle cycle is required, to accomodate the internal fix-up cycle required by the processor (see the R3051 Hardware User's Manual for more detail).

From To	READ	WRITE	DMA	IDLE
READ	Z,A	Z,A	Z,Z	Z,Z
WRITE	D,A	D,A	D,Z	D,Z
DMA	Z,A	Z,A	Z,Z	Z,Z
IDLE	Z,A	Z,A	Z,Z	Z,Z

Note: A — Address, D — Data, Z — Tri-State

Table 2. R3051 Data to Address Bus Transitional Behavior Between Memory Transactions

TYPICAL SYSTEMS AND BUS TURN AROUND

To handle the timing associated with the bus turn around within a memory cycle, the Data Enable output, $\overline{\text{DataEn}}$ is provided by the R3051. As shown in Figure 1, on read cycles, $\overline{\text{DataEn}}$ gives an indication when the CPU has tri-stated the A/D bus. Thus after $\overline{\text{DataEn}}$ asserts, the memory system can begin driving data onto the A/D bus. The system designer can also look for the rising clock edge of $\overline{\text{SysClk}}$ after $\overline{\text{Rd}}$ asserts before allowing the memory system to drive data.

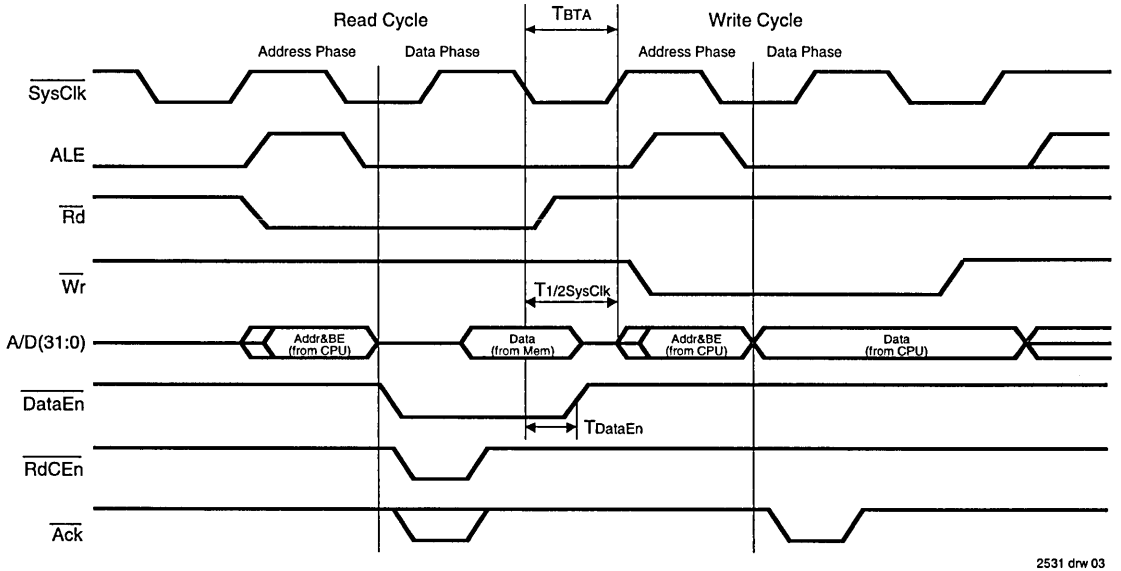


Figure 3. R3051 Read Cycle Followed by a Write Cycle

To handle the timing associated with the bus turn around between two memory transactions, consider the case of a read cycle immediately followed by a write cycle. The read cycle output enable control of the memory system must be such that the output drivers of the memory system turn off within 1/2 clock before the next address is driven by the write cycle. If the memory devices have an output disable to tri-state time (TOEZ) of more than 1/2 clock, then they can be isolated from the A/D bus with a bank of data transceivers such as the 54/74FCT245T, 54/74FCT861, or 54/74FCT623T or with latched data transceivers such as the 54/74FCT543T or 54/74FCT646T as shown in Figure 4. All of these transceivers have very fast output disable times.

VERY FAST $\overline{\text{SysClk}}$ OR VERY SLOW TOEZ AND BUS TURN AROUND

The majority of systems will use evenly matched memories relative to the system clock speed or use transceivers. However, two exceptions may occur:

1. Very Fast $\overline{\text{SysClk}}$ — Even with the highest speed transceivers, their output disable times (TOEZ) are around 5-8 nsec. Thus at 40 MHz, if $\overline{\text{DataEn}}$ is used, it has a clock to de-assert time of 4 nsec. (Assume that the transceiver has two internally And'ed output enable inputs. For example, as shown in Figure 4, the FCT543T transceiver bank can use $\overline{\text{DataEn}}$ and the bank select for inputs to the output enables). If 1 nsec is allowed for clock skew, this just meets the worst case timing criterion of:

   $T_{1/2SysClk}$ (12.5) ≥ T_{DataEn} + T_{OEZ} + $T_{ClkSkew}$ + T_{Cap} (4+6.5+1+0)

   Some choices of transceiver and PLA-based output enable control combinations may need more time than is allowed by the above equation. Solutions to this problem will be given in the section below, "Using DMA $\overline{\text{BusReq}}$ to Match CPU and Memory Speeds."

2. Very Slow Memories — The second case occurs when relatively slow TOEZ memories are attached directly to the A/D bus as shown in Figure 5. Such systems require these memories to turn off within 1/2 clock. A 20 MHz R3051 has a T_{DataEn} for the de-asserting edge of $\overline{\text{DataEn}}$ of 7 nsec. Assume that additional output enable control circuitry adds an additional delay of 10 nsec. 1 nsec is allowed for clock skew. For an inexpensive, slow 120 nsec EPROM, the output disable time is about 50 nsec, which seems to limit the clock speed to about 7 MHz:

   $T_{1/2SysClk}$ (71.4) ≥ T_{DataEn} + $T_{OutputEnableControl}$ + T_{OEZ} + $T_{ClkSkew}$ + T_{Cap} (7+10+50+1+0)

   However, as will be explained below in the section called, "Using DMA $\overline{\text{BusReq}}$ to Match CPU and Memory Speeds," the overall CPU speed does not have to be slowed down just because a slow TOEZ memory is attached directly to the A/D bus.

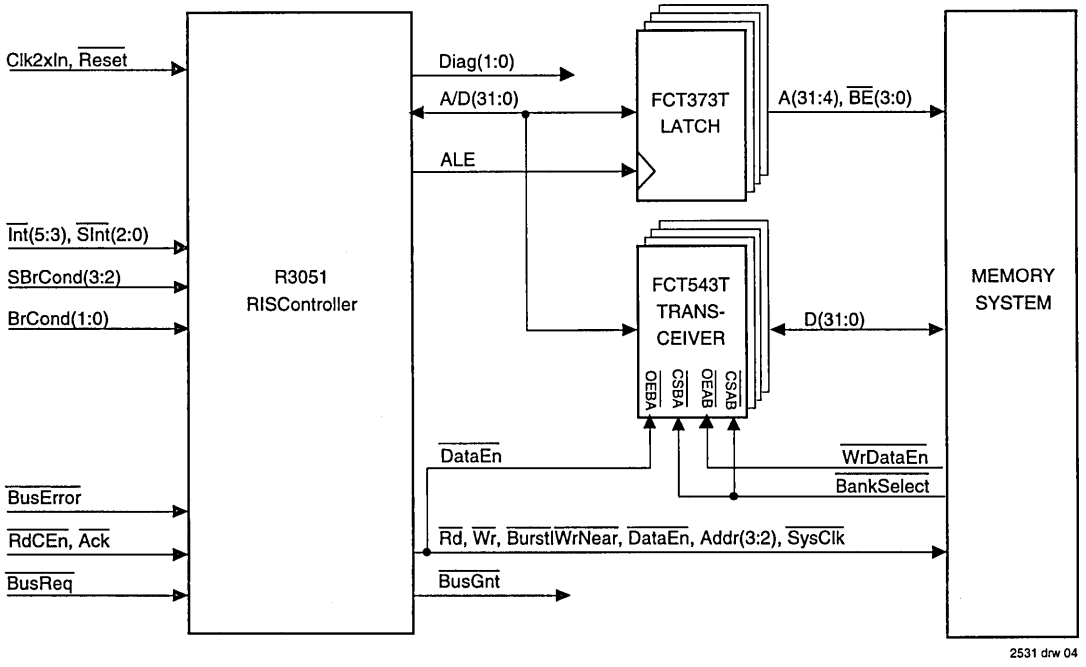


Figure 4. R3051 Memory System Isolated with Transceivers

USING DMA BusReq TO MATCH CPU AND MEMORY SPEEDS

For systems with very fast SysClk or very slow memories, a solution exists to the bus turn around timing constraints by using the Direct Memory Access (DMA) interface on the R3051. The R3051 DMA interface consists of two pins called BusReq and BusGnt as shown in Figure 6. Normally these pins are used for giving an external device control of the CPU bus instead of giving control of the bus to the R3051. In the R3051, when BusReq is asserted, DMA always has the highest priority immediately after the current memory cycle completes. The BusReq input is always sampled on the rising edge of SysClk. After the BusGnt is given, all of the CPU control line outputs, except SysClk and BusGnt are tri-stated. When the DMA device is finished with the bus, it de-asserts BusReq which then causes the CPU to de-assert BusGnt. The BusGnt output is always asserted on the rising edge of SysClk and de-asserted on the falling edge of SysClk.

Because a BusReq always has the highest priority, in a very fast SysClk system or a very slow memory system, asserting BusReq during the read cycle insures that the DMA request will always be granted at the end of the read cycle. After this happens, the BusReq pin can be de-asserted after the desired number of inter-cycle wait-states have been inserted. For example, as shown in Figure 7, by attaching the buffered read line, Rd to BusReq, the R3051 will grant the BusReq and immediately release it. Note that Rd needs to be buffered to meet the hold time of the BusReq input. Examine Figure 3, where a write cycle normally can follow a read cycle after 0.5 clocks and then compare it with Figure 7. In Figure 7, by using BusReq, it can be seen that a minimum of 1.5 clocks is guaranteed before the next memory cycle is started by the CPU.

Note that when using DMA, the system may choose to resistively pull-up or down its control signals since the DMA when granted will tri-state the CPU control output signals. Thus ALE could use a pull-down, while Rd, Wr, DataEn, and BurstlWrNear could use pull-ups. The resistor value of the pull-ups and pull-down is not that critical since the R3051 always drives the control signals to their de-asserted states before tri-stating them. Also, if the BusReq is needed for conventional DMA, a fixed-priority based arbiter can be used to allow bus turn around wait-state injection the highest priority and to allow conventional DMA the next priority.

Various improvements can be made to using the Rd line for BusReq. For example, instead of using the buffered Rd line, use the decoded chip select of the particular memory (e.g., the EPROM) that has the relatively slow TOEZ. Thus the extra wait-states are only asserted as needed (that is, after the slow memory is accessed).

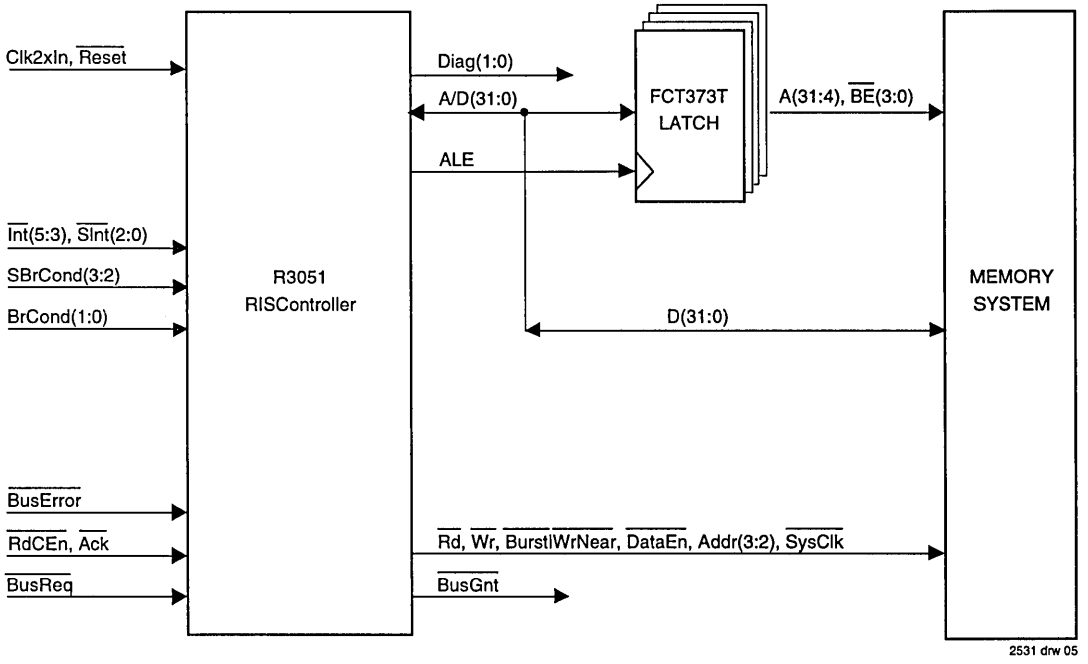


Figure 5. R3051 Memory System Connected Directly to the A/D Bus

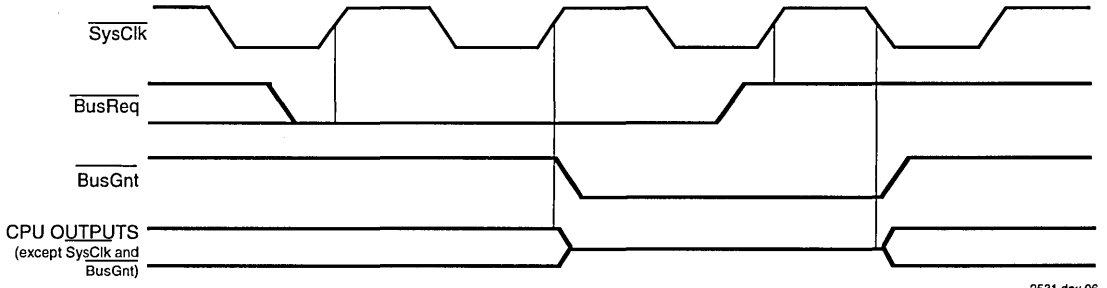


Figure 6. R3051 DMA BusReq and BusGnt Timing

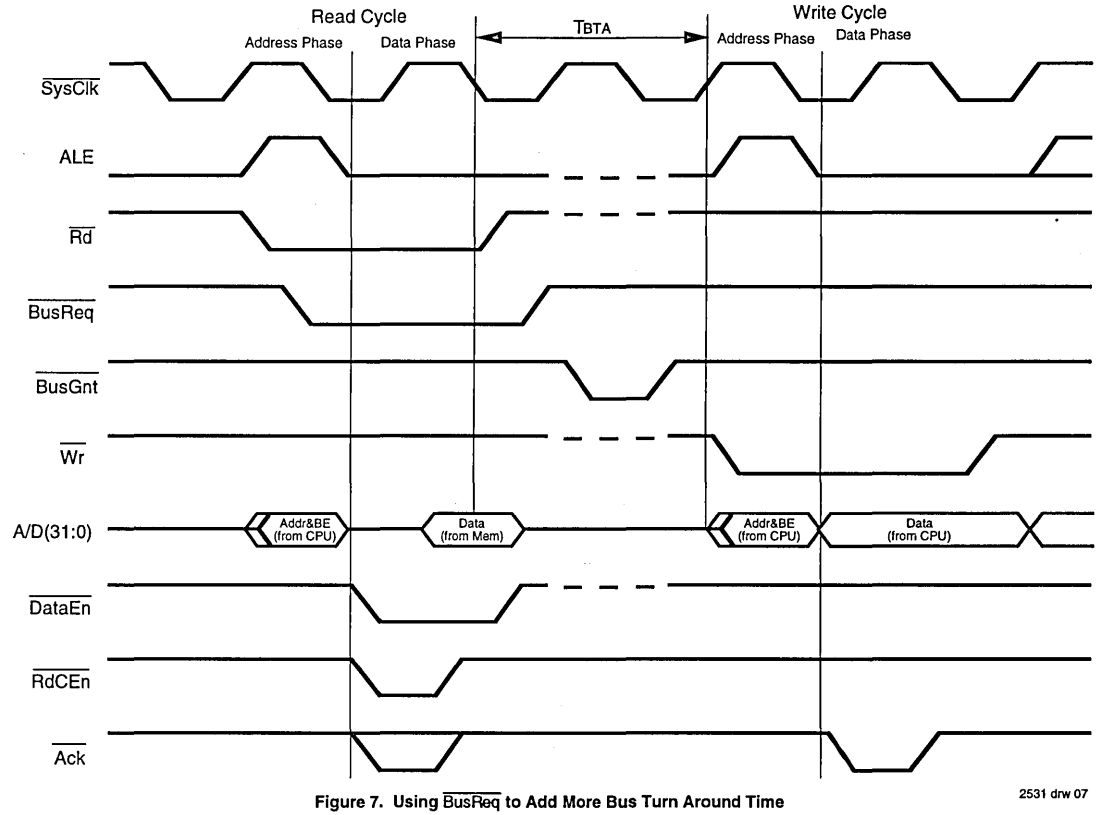


Figure 7. Using BusReq to Add More Bus Turn Around Time

SUMMARY

The R3051 allows inexpensive systems to be designed with the high throughput R3000 RISC instruction set architecture. The small 84-pin count is achieved with a multiplexed address and data bus, called "A/D". The use of the multiplexed A/D bus allows ASICs and Memory Controllers such as the R3721 DRAM Controller to have fewer interface pins, with no real loss of system performance or real added complexity. However, as for any high-speed bus (either multiplexed or not) care has to be taken to avoid bus clashes as the bus transitions from one device to another. This applications note describes these considerations.

As shown in the text, the use of the A/D bus does not inherently limit the overall clock speed of the system, since either transceivers, or the described method of using the DMA BusReq input gives a solution for memory/CPU mismatches. Thus any memory or I/O system can use the multiplexed A/D bus and be designed to run at the full CPU clock frequency.

FOR FURTHER INFORMATION:

1. IDT79R3051 Family Hardware User's Manual, Integrated Device Technology, Inc., MAN-RISC-00051, Santa Clara, CA, 1991. — Describes the H/W features and functionality of the device as well the bus interface.
2. IDT 1991 RISC Data Book, Integrated Device Technology, DBK-RISC-00021, Integrated Device Technology, Inc., Santa Clara, CA, 1991. — Contains the data sheet with packaging, pinout, AC/DC electrical and thermal parameters.
3. G. Kane, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1988. — Describes the R3000/ R3051 instruction set architecture from a systems and assembly level programming perspective.
4. IDT 1991 Logic Data Book, Integrated Device TechnologyInc., Santa Clara, CA, 1991. — Contains the data sheets of many different high-speed FCT transceivers, latches, and buffers.

Integrated Device Technology, Inc.

USING THE R3081™ IN R3051™-BASED SYSTEMS

APPLICATION NOTE AN-109

By Peter McDonald

INTRODUCTION

The IDT79R3081™ RISController™ is the newest member of IDT's family of high-performance and price-competitive 32-bit microprocessors. Designed to provide the high-performance MIPS® RISC architecture to low-cost and system integration-sensitive solutions, this processor adds to the growing family of RISControllers from IDT. The R3081 RISController is superset and pin compatible with the R3051/52, and includes 20kB of cache, a Floating-Point Accelerator, Hardware Cache Coherency support, and a series of system integration and interface features.

With its larger caches, FPA and interface features, incorporating the R3081 in an existing R3051 design can dramatically increase system performance without adding design complexity. Often upgrading to the R3081 is as simple as placing an R3081 in the R3051 socket. This applications note describes common considerations when upgrading existing R3051 systems with the R3081. As an example, this application note describes how to upgrade the 7RS385 evaluation board from an R3051 processor to an R3081 processor.

NEW FEATURES BROUGHT BY THE R3081

The R3081 is superset pin-compatible with the R3051. That is, in general it is possible to remove an R3051 from a system and replace it with an R3081. The system should run without any hardware or software changes. However, the R3081 adds additional capabilities to the R3051 family; some systems may wish to take explicit steps to take advantage of these new capabilities.

Before discussing system changes needed to implement the superset features of the R3081, a definition of these capabilities is needed. As mentioned above, the R3081 includes larger Instruction and Data Caches, a Floating-Point Accelerator, Hardware Cache Coherency support, and a series of integrated control options. All the hardware options are selected by either the mode initialization vectors (values sampled on the interrupt input lines during reset) or programmed through the new CP0 Configuration register. Below is a summary of the new R3081 features. A more detailed list of these features along with a list of the differences between the R3051 and R3081 are included in the IDT79R3081/3081E Integrated RISController Hardware User's Manual.

- *Larger Instruction and Data Caches*
  The R3081 instruction and data caches total 20kB. The default (reset) configuration is 16kBI and 4kBD, although they are dynamically programmable to 8kB apiece. Both instruction and data caches are parity protected over the data and tag fields. This differs from the R3051, in that both caches are larger than the caches supported by the R3051 or R3052, the cache is configurable and the caches are parity protected.
- *Addition of a Floating-Point Accelerator*
  A full-featured R3010A-compatible floating-point accelerator is incorporated on the R3081 adding single- and double-precision add, multiply, and divide instructions to the instruction set. Which of the six integer unit Interrupts inputs is used for the floating-point interrupt signal is programmable. $\overline{\text{Int3}}$ is the default FP interrupt. Thus, one of the six interrupt inputs of the R3051 is used for the floating-point interrupt and coprocessor 1 instructions will be directly executed by the on-chip floating-point units.
- *Cache Coherency Interface*
  The R3081 has a hardware-based cache coherency interface for multi-master systems. If selected, DMA cycles between memory and I/O can invalidate lines within the R3081 cache, insuring that there is no stale data and avoiding software directed cache flushing. This mechanism can be disabled to achieve full R3051 compatibility; alternately, the system designer can choose to increase the performance of multi-master systems, by performing hardware cache coherency.
- *Power Reduction Mode*
  The R3081 RISController can be dynamically programmed to reduce its operation frequency. In this mode the execution clock, and therefore the output clock, is internally divided by 16. This function allows the power reduction benefits of a lower speed clock to be achieved during idle periods, without requiring external clock shaping logic.
- *Programmable Halt Mode*
  This programmable mode forces the R3081 RISController to stall until either an interrupt or reset is issued. This mode has two effects: it further reduces power consumption; and, it allows software to halt until some external event occurs.
- *Half-Frequency Bus Mode*
  A selectable mode allows the R3081 bus interface to operate at one-half the frequency of the processor core. For example, the execution core can run at 33MHz, and the bus interface at 16MHz. Given the substantial amount of cache on-chip, the slow system interface will not dramatically degrade performance. The end result is a high-performance system with very low system cost.
- *1x or 2x Clock Input*
  The R3081 can operate with either an R3051 compatible double-frequency clock input (2x clock mode), or can operate from a clock at the execution rate (1x clock mode). This capability both simplifies EMI at high frequency, and also

allows for "clock doubling" when used in conjunction with the one-half frequency bus mode.

- *Slow Bus Turnaround*
  A common problem for a high-speed I/O bus is the amount of time available for mastership changes. The R3081 allows software to specify a larger minimum time when transitioning from the memory driving the bus (i.e. read data) and the processor driving the bus (e.g. writes). This reduces the speed requirement of data transceivers, with minimal performance impact.
- *Dynamically programmed data cache refill*
  The R3081 allows software to dynamically select between single word and quad word refill on data cache miss. This allows for additional performance tuning, by enabling the kernel to select the best algorithm for a given section of code. The default refill size is selected at reset time, the same as for the R3051.

POSSIBLE CHANGES

The R3081 hardware options are either mode selectable at reset or programmed through an internal register. Hardware cache coherency support and all clocking modes, half-frequency bus mode and 1x or 2x clock input mode, are selected at reset based on the level of the $\overline{\text{Int}}[5:3]$. In the R3051, $\overline{\text{Int}}[5:3]$ are required to be driven HIGH during reset initialization.

The interrupt inputs, $\overline{\text{SInt}}[2:0]$ are already used by both the R3051 & R3081 to select data cache refill sizes, tri-state test mode, and big or little endian system architectures. The complete table of the R3081 reset mode vectors is listed in Table 1.

A complete description of these modes is provided in the IDT79R3081/3081E Integrated RISController Hardware User's Manual.

Floating-Point Interrupt

The one area where hardware changes may be necessary are with respect to the Floating-Point Accelerator. In the MIPS RISC architecture, the floating-point interrupt is fed into a general purpose interrupt. Interrupts cause the processor to jump to the system's exception handler which then decodes its status to determine the exception cause. One of the six external R3081 interrupts (by default $\overline{\text{Int3}}$) is programmed to be the FPA interrupt. All activity on the external interrupt pin corresponding to the FPA interrupt is ignored.

Although software can use a different interrupt input other than the default, it is still the case that only five external interrupt pins remain available to external peripherals. Therefore, systems that required six external interrupts will need to modify their external interrupt structure, perhaps by causing multiple peripherals to share a single interrupt input. Obviously, software would then need to decode which device on that interrupt actually signalled the exception.

Systems that have defined an interrupt other than $\overline{\text{Int3}}$ for the FPA need to modify their startup code so as not to ignore the assertion of $\overline{\text{Int3}}$.

Some software applications incorporate exception handlers that allow the user to set the FPA interrupt through software. The IDT/sim™ diagnostics uses this method. This adds system flexibility at the cost of the extra performance required to decode the interrupt.

Table 1. R3081 Mode Selectable Features

Interrupt Pin	Mode Feature
$\overline{\text{Int5}}$	$\overline{\text{CoherentDMAEn}}$
$\overline{\text{Int4}}$	$\overline{\text{1xClockEn}}$
$\overline{\text{Int3}}$	$\overline{\text{Half-frequency Bus}}$
$\overline{\text{SInt2}}$	DBlockRefill
$\overline{\text{SInt1}}$	$\overline{\text{Tri-State}}$
$\overline{\text{SInt0}}$	BigEndian

The Config Register

Selecting which interrupt is used by the on-chip FPA, the cache configuration, power reduction mode, current size of data cache refill, halt/stall mode, or slow bus turnaround are all accomplished by writing to the new CP0 configuration register. The Configuration Register data format is shown in Figure 1.

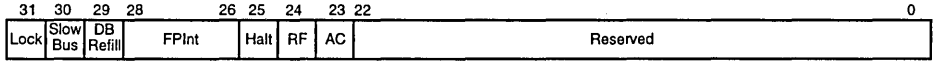
The reset initialization value of the config register depends somewhat on the mode vectors selected at reset. Specifically, the initial values of the Data Block Refill bit, and of the slow bus turnaround bit, are dependent on the reset vectors. At reset, the FPInt field will correspond to $\overline{\text{Int3}}$, and the Lock, Alt. Cache, Halt, and RF bits will be cleared.

Reading and writing all CP0 registers is accomplished by issuing coprocessor load and store instructions. The configuration register is CP0 register 3. An interactive tool to read and write the R3081 configuration register, "the R3081 Configuration Tool", is available as a demo tool through your local sales office, and runs on IDT/sim-based platforms. To insure strict software compatibility with older applications, the Config register can be isolated from subsequent writes by writing a '1' to the configuration register "Lock" field.

Software Compatibility

The R3081 will directly execute applications written for the R3051. The larger on-chip caches will directly benefit existing applications, and thus bring an increase in system performance. Additional gains are possible, depending on the application code, by taking advantage of the hardware FPA on the R3081. Whereas the R3051 must either trap and emulate floating-point instructions, or perform explicit calls to software floating-point libraries, the R3081 can directly execute these operations.

It may be advantageous to generate two distinct binaries from one source; one, which uses software libraries to emulate floating-point operations, and is used with the R3051 or R3052 and another, which uses the on-chip FPA to perform floating point. However, if the prospect of two distinct binaries



Lock: 1 -> Ignore subsequent writes to this register
Slow Bus: 1 -> Extra time for bus turnaround
DB Refill: 1-> 4 word refill
FPInt: Power of two encoding of FPInt <-> CPU Interrupt
Halt: 1 -> Stall CPU until reset or interrupt
RF: 1 -> Divide frequency by 16
AC: 1 -> 8kB per cache configuration
Reserved: Must be written as 0; returns 0 when read

Figure 1. CP0 Configuration Register Data Format

is too onerous for a particular application, the binary could include FPA instructions; with an R3051 processor, a trap will be generated, and software could emulate the operation. Although a single binary suffices for both processors, the cost is reduced performance for the R3051.

Software can dynamically determine whether there is an FPA available, by performing simple FPA diagnostics. Such diagnostics is included in IDT/sim, IDT/c™, and IDT/kit™ startup code. Thus, the boot software could check for the presence of an FPA, and initialize the Coprocessor One useable bit according to the results. This allows a single binary to dynamically determine whether a hardware FPA is available, and can be used to enable the FPA instruction trap mechanism of the R3051 and R3052.

Manipulating the Cache Characteristics

Another possible performance gain may exist by dynamically manipulating the cache characteristics of the R3081. The Config register allows the cache configuration to be dynamically changed from 16kB I-Cache and 4kB D-Cache to 8kB I-Cache and 8kB D-Cache. A kernel may choose to dynamically change the cache organization, depending on the nature of the task about to be executed. The only caveat is that when changing the cache configuration (from 16kB/4kB to 8kB/8kB or vice versa), both the instruction and data caches need to be flushed.

In addition, software could dynamically alter the D-Cache refill size. Changing this bit does not require a cache flush.

Note that to insure compatibility amongst multiple generations of R3051 family members, cache flushing routines that assume a constant cache size are discouraged. The R3081 Hardware User's Manual presents an algorithm where software can determine the cache size available.

UPGRADING THE RS385 BOARD WITH THE R3081

Upgrading the RS385 board with the R3081 RISController is easy to accomplish. Simply remove the R3051 and replace it with the R3081. Both share the same footprint and pinout. The 1xClockEn, Half-frequency bus, and Coherent DMA modes are all disabled in a default 7RS385, thus no further hardware modifications are necessary. $\overline{\text{Int}}$[5:3] are pulled HIGH during reset disabling these three modes.

The IDT/sim included with the 7RS385 automatically sizes the cache available; thus, the increased cache sizes of the R3081 pose no problem. IDT/sim will not, however, write to the Config register. Thus, the FPU interrupt will default to $\overline{\text{Int3}}$, unless explicit steps are taken.

Currently on the RS385, the R3051 $\overline{\text{Int3}}$ is used for the Centronics port interrupt. If using the Centronics port and the R3081 FPA, the system and/or software must be modified so that the FPA is allowed its own dedicated interrupt. This needs to be done by either re-writing the boot prom to modify the config register or using a different Centronics interrupt and modifying the Centronics driver.

If the 7RS385 has been used as a porting target for another application, the types of software changes needed will be application dependent. Applications developed with IDT/kit and/or IDT/c include startup code that resizse the cache every time they are executed. IDT/sim startup code does not resize the cache at each execution. In addition, it may be desirable to recompile for any floating-point instructions that are implemented with software emulation.

Implementing Additional Reset Modes

When using any of the three reset mode features unique to the R3081, minor modifications to the RS385 board are necessary to implement the interrupt input signal multiplexing during reset. As a general note, the RS385 uses a tri-statable interrupt bus to implement the multiplexing for the $\overline{\text{SInt}}$[2:0]. An asserted MRES# enables the reset mode vector driver. A modification to the RS385 board was made to enable or disable any of the six mode selectable features with jumpers, including the new mode vectors of the R3081. Figure 2 shows the modified R3051/R3081 interface to allow enabling and disabling of the six reset modes. A buffer, U1A, was added to provide the tri-state mux for the three new reset modes.

Other solutions to implement the reset mode selection abound, depending on one's application. All R3051 designs should already pull $\overline{\text{Int}}$[5:3] HIGH during reset as specified in the IDT79R3051 Family Hardware User's Manual. Therefore, only the new modes being selected need to be added to the current muxing on the RS385. If only one additional mode is needed, jump the one remaining output on the current 74FCT244 reset mode mux (U37 pin 18) to the appropriate

interrupt input. The interrupt PAL, U28, can be reprogrammed to do some of the muxing. (If the PAL can not be easily removed from the board, an additional device can be added to the wire-wrap area.)

An Interesting Upgrade

One of the more interesting upgrades possible is to increase the execution speed while decreasing the bus clock. To do this, select 1x clock mode and half-frequency bus from the new mode reset logic, and replace the R3051 osciallator with a 40MHz oscillator. The result will be a CPU core executing at 40MHz rather than 25MHz, although the bus speed has been reduced to 20MHz.

UPGRADING OTHER R3051 SYSTEMS

Upgrading any R3051-based system with the R3081 RISController is very similar to updating the RS385 board. The one hardware item that may differ has to do with DRAMs and their refresh.

Specifically, if the refresh period is based on counting SysClk cycles, then using the reduced frequency mode of the R3081 may violate the reset period (reduced frequency mode also divides the frequency of the output clock). There are two solutions to this, depending on the application:

- Reprogram the counter to a smaller number of SysClks. This is possible with devices such as the R3721 DRAM controller.
- Use a different reference clock for refresh. Choices include a UART clock, or the clock used to generate the input clock to the processor.

The RS385 board refresh request is generated from a clock independent of $\overline{\text{SysClk}}$. The clock used is derived from the UART clock.

CONCLUSION

Incorporating the high-performance R3081 RISController into existing R3051-based systems is often as simple as merely swapping processors. Little design complexity is added, yet system performance increases due to the larger caches, Floating-Point Accelerator, and other features. Using more of the R3081 features to increase performance even more can be accomplished with minimal hardware and software modifications.

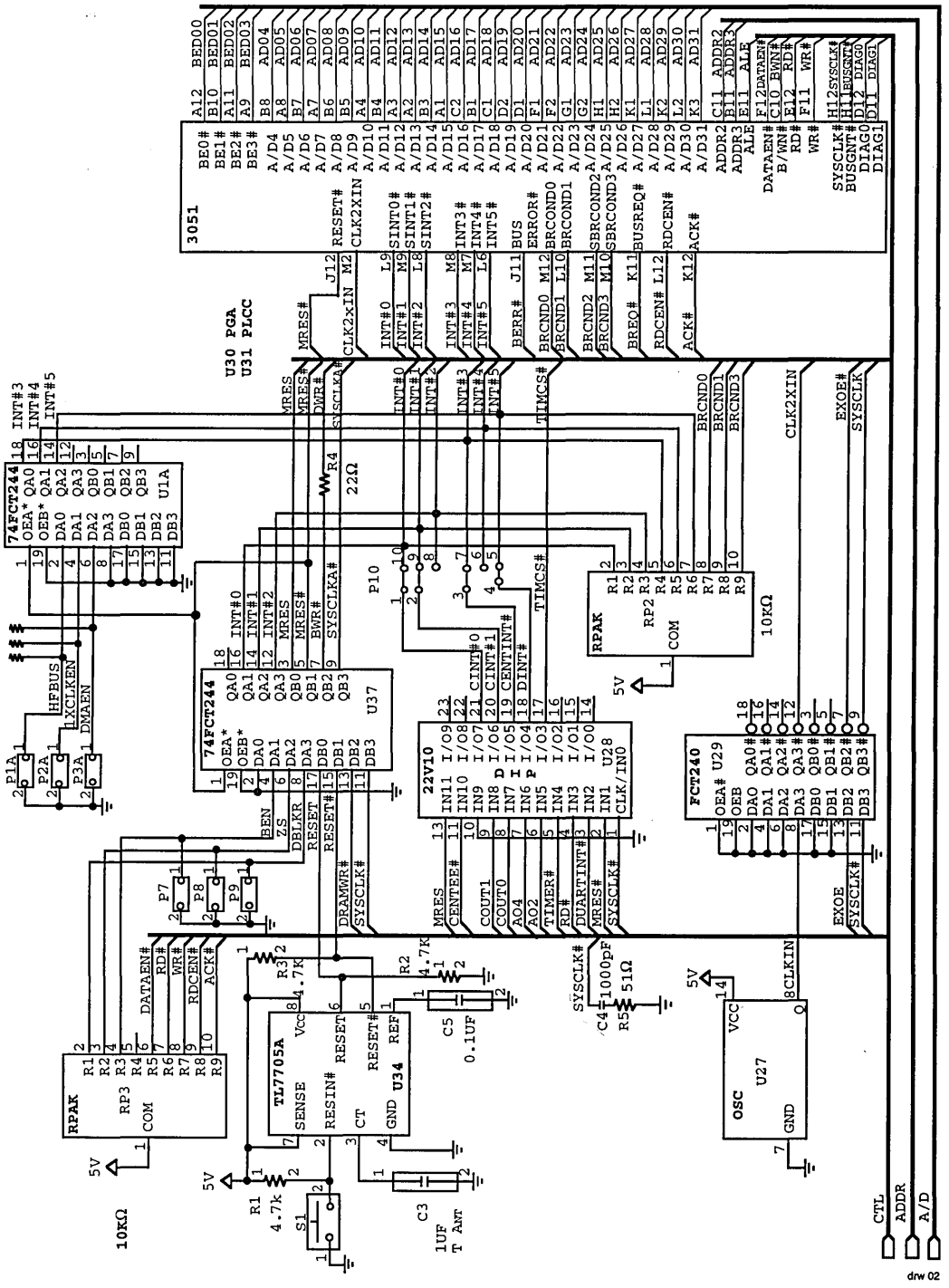


Figure 2. 7RS385 Mode Vector Logic Upgraded for R3081

Integrated Device Technology, Inc.

UPGRADE STRATEGIES FOR IDT79R3051™-BASED DESIGNS

APPLICATION NOTE AN-113

By Phil Bourekas

INTRODUCTION

The IDT RISController™ family includes various highly-integrated microprocessors providing high levels of performance with low system cost. Currently, the R3051™ family includes three different devices, each providing differing levels of price performance, yet each pin-compatible with each other. This allows the system designer to implement a single base system, yet offer various end products at different capability levels. The end result to the customer is reduced time to market for a product family, and the amortization of a single development effort over a wider variety of end products. This wide range of pin-compatible performance is not currently achieved by any other RISC processor family.

This application note describes system design techniques that insure a high degree of interchangeability with no real design impact.

THE R3051 FAMILY

Common characteristics of the R3051 family include high integration at low cost. All current family members are pin-compatible. All family members include:

- Substantial amounts of separate instruction and data caches integrated on-chip. Although the amount of caches varies across different family members, all devices contain enough cache on-chip to achieve extremely high performance with low-cost memory systems. The caches on the R3052 and on the R3081™ are actually larger than the cache on the Intel 80486 high-end processor, enabling these devices to offer higher performance at lower cost.
- MIPS R3000A compatible integer CPU. The R3051 family was designed by integrating cache and a low-cost bus interface around the standard MIPS R3000A CPU. This RISC core is widely recognized as an extremely high-performance execution engine, with powerful compiler and development tools. Some of the features of the core include a large register file, single cycle ALU, rich set of branch instructions (including compare operations as part of the branch), and separate, autonomous integer multiply and divide. Since the R3051 was designed using the standard core, 100% software compatibility is guaranteed. Thus, compiler tools, real-time operating systems, and other software tools developed around the standard R3000A work without modification on the R3051 family.
- Optional Translation Look-aside Buffer (TLB). The "E" (Extended Architecture) versions of the RISController family feature a 64-entry, fully associative TLB. The TLB allows virtual addresses to be translated into physical addresses on a 4kB page basis. The TLB is useful in providing memory protection and debug utilities in any application; in other applications, such as those using a real-time operating system, or in an X-windows server, the TLB allows increased system functionality to be provided.
- Simple, low-pin count bus interface. The R3051 family uses a time-multiplexed 32-bit address and data bus to communicate with memory. Internal to the processor are 4-deep read buffer and write buffer FIFO's to decouple the speed of the internal execution core from the slower speed memory system. The multiplexed bus arrangement has many advantages, such as lower-cost interface chips and ASICs, without impacting system performance.

Currently, there are three family members. These are:

- The R3051/51E. This device features 4kB of Instruction cache and 2kB of Data Cache. There is no hardware floating-point unit available on this device.
- The R3052/52E. This device features 8kB of Instruction cache and 2kB of Data Cache. As with the R3051, there is no hardware floating-point unit available on this device.
- The R3081/81E. This device introduces a number of new features to the family. The primary features of interest are changes to the caches, and inclusion of a hardware floating-point unit; other features will be described throughout this application note. The R3081 implements 16kB of Instruction Cache and 4kB of Data Cache; kernel software can dynamically reconfigure the on-chip caches as 8kB of Instruction and 8kB of Data Cache.

POTENTIAL UPGRADE OPPORTUNITIES

A number of possible system upgrades from a single, base design are possible. Elsewhere in this application note, design considerations to assure interchangeability are described.

Possible upgrade strategies include the following techniques:

Upgrading Cache Size

As all devices are pin compatible; it is possible to increase performance of an application by upgrading the amount of cache available on-chip. Thus, holding all other components the same, an R3051 may be removed and replaced by an R3052 to double the instruction cache. An R3052 can be removed and replaced with an R3081, doubling both the instruction and data caches.

Add Hardware Floating-Point

One upgrade to higher performance involves upgrading an R3051 or R3052 to an R3081 and taking advantage of the on-chip floating-point accelerator. Later in this applications note, software considerations for such an upgrade are described.

This upgrade will obviously substantially increase the performance of software containing floating-point operations; while the IDT software floating-point environment is very efficient, the floating-point unit of the R3081 dramatically outperforms integer emulation, and may result in a significant speed-up of some applications.

Increasing Frequency

Obviously, one way to increase performance is to increase the system frequency. This may or may not be easy to do, depending on the exact system design. Obviously, such an upgrade will typically require the replacement of multiple devices on the PCB.

Note, however, that R3051 family packaging insures that the same footprint and pinout is available across the full frequency range of the family, and for all of the family members. Thus, the same 84-pin PLCC footprint used for a 20MHz R3051 accommodates the package for a 40MHz R3081, even though that device consumes more power. This obviously simplifies upgrading a design to a higher frequency processor. Design techniques for increasing frequency may include:

- Using faster memory devices to achieve the same relative access time.
- Using faster control logic, such as faster PALs or transceivers, to increase set-up time and reduce propagation delays. For example, a 15ns PAL may be replaced with a 10ns PAL, effectively allowing the clock period to be reduced 5ns.
- Re-programming PALs and control logic to increase the number of wait cycles. While this will reduce the frequency normalized performance, the absolute performance will be increased substantially, since the processor will execute (typically out of its internal cache) at a higher rate.

"Clock Doubler" Operation

The R3081 presents a particularly unique opportunity to upgrade systems using an R3051 or R3052. This is particularly due to the "half-frequency bus" mode of operation of the R3081.

A dramatic system upgrade can be achieved by:

1. Removing a 20MHz R3051 or R3052 and replacing it with a 40MHz R3081.
2. Selecting the "half-frequency bus" and "1x clock" modes via the reset vectors.

The resulting system bus will continue to operate at 20MHz, but the CPU will execute out of its internal cache at 40MHz. The resulting system will typically see its performance more than double (recall that the upgrade to the R3081 will also increase the on-chip caches and add hardware floating-point, relative to the R3051 or R3052).

It is also interesting to note that the performance impact of running a 40MHz processor with a 20MHz bus is not as severe as one would intuitively guess. This is due to the fact that memory access time is really in units of time, rather than in wait states. That is, 200ns access memory is 4 clock cycles at 20MHz and is 8 cycles at 40MHz; the absolute time is not improved by running the bus faster.

Intel has estimated that for the i486 with clock doubling, running the bus at one-half the CPU execution rate is approximately 11% less efficient than running the bus at the full CPU rate on benchmarks such as the SPEC benchmark suite. The R3081 contains more than twice the amount of on-chip cache as does the i486, and thus will be even less dependent on bus performance; thus, the performance degradation should be even less.

DESIGN CONSIDERATIONS FOR UPGRADING

The remainder of this applications note details specific techniques which facilitates the interchange of various members of the R3051 family. In general, all devices are pin and footprint compatible, so there are no PCB issues to be concerned about. In general, the only things needed to upgrade a design are:

- Design it around an R3051. The R3081 does include some superset features relative to the R3051 which simplifies high-speed systems; however, if a system works for the R3051, it will work for an R3081.
- Make the software independent of cache size. The various devices include varying amounts of cache on-chip. An algorithm to determine the amount of cache available is presented in this applications note.
- Have a strategy for software floating-point versus hardware floating-point. The R3081 adds a high-performance hardware floating-point accelerator, as well as increasing the cache size. This applications note describes various software techniques for dealing with software emulation versus hardware acceleration of floating-point.

Thus, this application note details specific hardware choices and software choices which facilitate interchanging CPUs. In addition, the application note illustrates techniques for determining the presence or absence of the R3081 config register, the R3081 FPA, and the amount of cache on-chip.

SOFTWARE CONSIDERATIONS FOR UPGRADING SYSTEMS

Some of the system upgrade considerations should be accommodated in the application software (especially the kernel). It is possible to develop a single binary set of code which performs across all of the family members.

Sensitivity to Cache Size

Obviously, one characteristic difference among the various family members is the amount of Instruction and Data cache available. Thus, to insure interchangeability among these devices, the software should be written to be insensitive to the cache sizes.

Typically, very little of the actual application will be functionally sensitive to the amount of on-chip cache; the primary difference will be in the performance achieved. This is the primary advantage of caches with respect to memory mapped zero-wait state RAM; caches are transparent to the software, and do not affect the memory map.

Typically, the only part of the software that may be sensitive to the cache size will be the boot/initialization software, which may perform certain memory (including on-chip cache) diagnostics, and which must initialize the on-chip cache by performing a cache flush.

Figure 1 shows a listing of a routine to perform cache sizing. This routine uses bits of the on-chip status register to isolate the cache (to prevent writes or cache misses from propagating to memory), and to swap the cache (to perform the algorithm on the Instruction cache). In order to determine hit or miss, the algorithm places a marker in the first word of the cache, and then looking for the cache size such that a read of the cache forces a wrap-around to reading location zero. Once this occurs, the maximum cache size has been exceeded, and thus the cache size is known. Other algorithms could use the cache miss bit of the status register, rather than a marker value. This capability is provided in the IDT/kit™ and IDT/sim™ software packages from IDT.

Once the cache size has been determined, it is used in the cache flush routines (for example) to completely flush the caches. Note that if the only time the cache is flushed is at system start-up, it is acceptable to assume a worst case (large) cache size and flush that amount of cache; caches smaller than the size assumed will merely be flushed multiple times, resulting in wasted execution time but correct functionality. On the other hand, applications which perform cache flushing as part of ongoing operation (e.g. to assure cache coherency when DMA operations are used) would be sensitive to performance, and thus would desire to flush only the proper amount of cache.

Floating-Point Presence

Another difference between various family members has to do with the presence or absence of the floating-point. This distinction may have two impacts on the software environment:

- The initial setting of the coprocessor 1 usable bit should reflect whether or not a hardware floating-point is available. It is possible to create a software environment which can dynamically determine the presence or absence of the FPA.
- The actual binary executable of the application may be best optimized according to the presence or absence of a hardware floating-point. This is discussed below.

How to Determine Floating-Point Presence

There are at least two different methods for determining whether a floating-point is present. One way is to perform floating-point operations and determine whether the results are reasonable; these operations could be as simple as moving data into and out of the FPA registers to see if they are present, through performing floating-point calculations and examining the results (or even possibly seeing if an exception is reported). If the floating-point is detected as present, coprocessor 1 should be marked as usable by the kernel.

Another method would be to use the CpCond(1) (coprocessor 1 condition) flag. The hardware could tie the CpCond(1) to a known state (e.g. HIGH); software could then perform a compare operation (or move to the fp cscr register) to cause CpCond(1) to report the opposite polarity. A simple branch on coprocessor (1) condition will then determine whether the CpCond(1) signal is driven by an on-chip FPA, or by the off-chip pull-up resistor.

FPA Impact on the Binary Code

There are two methods for dealing with the software which may or may not have a hardware floating-point unit. The optimal method depends on trade-offs between a single binary set operating either with or without a hardware FPA, versus a single source set compiled twice resulting in two binaries (one targeted to a hardware FPA and one targeted to an integer only environment).

Using a Single Binary with and Without an FPA

If the system designer chooses to implement a single binary capable of taking advantage of a hardware FPA when one is available, all that needs to be done is to tap into the inherent capabilities of the MIPS coprocessor architecture. Specifically, if the kernel marks the coprocessor 1 FPA as unavailable, FPA instructions will cause a trap to occur. The kernel can then perform an integer interpretation of the FPA instruction. The application software is then compiled to assume the availability of a hardware FPA: if one is available in the system fine; if not, traps will occur when FPA operations are encountered, and the kernel can perform an emulation of the function.

Using this technique requires two things in the software:

- Boot software must perform the diagnostics described above to determine the appropriate setting for the coprocessor 1 usable bit.
- The kernel must include the capability to emulate the entire FPA unit, including the FPA operations, the register file, and the FPA exception mechanisms used by the application.

While this technique has the advantage of resulting in a single binary which works in either environment, the result is added complexity and a loss of performance in the environment in which no FPA is available. Specifically, the kernel must provide an emulation library of the entire FPA; and, software FPA operations will include additional overhead from the CPU exception model and from emulating all aspects of the FPA, even though a given operation only requires a subset of the FPA functionality.

Developing Two Binaries from a Single Source

Another technique exists whereby two distinct binaries are developed from a single source tree. Each of the resulting binaries is fully optimized for either an integer only environment, or for an environment in which a hardware floating-point is available.

This is accomplished by taking advantage of the software floating-point library capabilities of the IDT/c™ environment. IDT/c includes a compile time flag which can be used to control whether hardware FPA instructions (coprocessor 1 instructions) are generated, or whether direct calls to a software floating-point library are generated. Thus, software floating-point is not forced to emulate the register set and data type conversions of the hardware FPA, and execution is not forced to go through the CPU exception model. The resulting binary operates much more efficiently than one which goes through the trap and emulation model described above.

A separate applications note describes how to determine the optimal compilation environment for a given application.

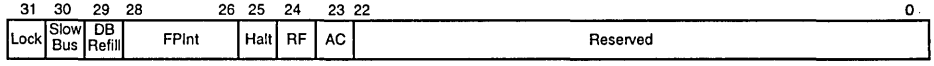
```
/*****************************************************************************
**
** _size_cache()
** returns cache size in v0
**
*****************************************************************************/

FRAME(_size_cache,sp,0,ra)
      .set        noreorder
      mfc0        t0,C0_SR                    /* save current sr */
      and         t0,~SR_PE                   /* do not inadvertently clear PE */
      or          v0,t0,SR_ISC                /* isolate cache */
      mtc0        v0,C0_SR
      /*
       * First check if there is a cache there at all
       */
      move        v0,zero
      li          v1,0xa5a5a5a5               /* distinctive pattern */
      sw          v1,K0BASE                   /* try to write into cache */
      lw          t1,K0BASE                   /* try to read from cache */
      nop
      mfc0        t2,C0_SR
      nop
      .set        reorder
      and         t2,SR_CM
      bne         t2,zero,3f                  /* cache miss, must be no cache */
      bne         v1,t1,3f                    /* data not equal -> no cache */
      /*
       * Clear cache size boundries to known state.
       */
      li          v0,MINCACHE
1:
      sw          zero,K0BASE(v0)
      sll         v0,1
      ble         v0,MAXCACHE,1b

      li          v0,-1
      sw          v0,K0BASE(zero)             /* store marker in cache */
      li          v0,MINCACHE                 /* MIN cache size */

2:    lw          v1,K0BASE(v0)               /* Look for marker */
      bne         v1,zero,3f                  /* found marker */
      sll         v0,1                        /* cache size * 2 */
      ble         v0,MAXCACHE,2b              /* keep looking */
      move        v0,zero                     /* must be no cache */
      .set        noreorder
3:    mtc0        t0,C0_SR                    /* restore sr */
      j           ra
      nop
ENDFRAME(_size_cache)
      .set        reorder
```

Figure 1. Cache Sizing Software



Lock: 1 -> Ignore subsequent writes to this register
Slow Bus: 1 -> Extra time for bus turnaround
DB Refill: 1-> 4 word refill
FPInt: Power of two encoding of FPInt <-> CPU Interrupt
Halt: 1 -> Stall CPU until reset or interrupt
RF: 1 -> Divide frequency by 16
AC: 1 -> 8kB per cache configuration
Reserved: Must be written as 0; returns 0 when read

Figure 2. R3081 Config Register

The method of dealing with floating-point operations in an integer CPU only environment is particularly important in the evaluation of a compiler platform; techniques such as the "mix and match" approach supported by IDT/c allows the best capabilities of the MIPS compiler toolchain to be integrated with efficient software floating-point emulation.

The obvious advantage of this approach is the optimum performance achieved for both the integer only system and the R3081-based (hardware FPA) system. Using distinct EPROM sets at manufacturing time, or upgrading both the EPROMs and processor as a field upgrade, are obvious consequences, but in general are not particularly onerous (EPROM upgrade can be a replacement of EPROMs, or, for FLASH EPROM, a re-programming of the EPROMs resident on the board).

The R3081 Config Register

The R3081 includes, as part of coprocessor 0, an additional control register called "Config". The R3081 Config Register is shown in Figure 2.

The Config register controls various aspects of system functionality. If these features are used in an R3081 system, software must first determine whether they are available.

To determine whether the current device is an R3081 (and thus whether the config register is available), software can use various techniques. One straightforward technique is to determine whether or not there is an FPA; if so, the device is an R3081. Similarly, software could determine the cache sizes available, and see if these correspond to the organization the R3081.

Other techniques are also possible; for example, size the cache, then reconfigure the cache by writing to the config register; re-size the cache to determine that the change occurred. Obviously, if the change occurs, the config register is available.

Note that writes to this register location in the R3051 or R3052 will have no effect; no side effects occur, and no traps are signalled. Reads of the config register produce an undefined data result for the R3051 and R3052.

If the config register is used when an R3051 is in place, various other considerations exist. These are:

- *Floating Point Interrupt.* In general, if an R3051 application intends to also work with an R3081, one of the CPU interrupt inputs needs to be reserved for the hardware FPA of the R3081. The default interrupt is Int(3), but the config register allows a different interrupt assignment to be used. The corresponding interrupt input pin of the R3081 is then ignored. Thus, the PCB should contain a pull-up resistor at the interrupt pin; when an R3051 is used in the application, no interrupt will be signalled.
- *Reduced Frequency.* This mode dramatically reduces the power consumption of the R3081, by reducing its operation frequency. This mode is unavailable in the R3051. In general, the only real functional system change that occurs is that the SysClk output clock frequency is also reduced; thus, if DRAM refresh, for example, was derived from this clock, the counter value should be reprogrammed. If an R3051 is told to "reduce frequency", nothing will happen.
- *Halt.* This control bit forces the R3081 to stall until an interrupt input is asserted, or a reset is encountered. This mode is unavailable in the R3051, and no simple software equivalent exists.
- *Data Block Refill.* The R3081 allows the block size read on a data cache miss to be dynamically reconfigured by software. The initial value is set by the reset value. In general, this bit may affect the performance of software, but is unlikely to impact its functionality.
- *Alternate cache.* This bit allows the caches to be dynamically reconfigured for the R3081. A cache flush should be performed after the cache is reconfigured. An earlier section of this applications note discussed how to make software independent of the cache organization.
- *Lock.* This bit allows software to inhibit subsequent writes to the Config register. Thus, boot software can set up the operation mode, and then protect it from other software.
- *Slow Bus Turnaround.* This bit allows systems to enjoy longer time between A/D bus mastership transitions. However, this software control is not available on the R3051. If the system designer desires extra time, and also desires to be able to interchange R3051s and R3081s, the hardware technique described in applications note AN-97 is appropriate. This technique uses the DMA arbiter interface of the CPU to insure that new transactions are not begun until ample time for bus turn-off has passed. This hardware technique works equally well with both the R3051 and R3081.

HARDWARE DESIGN ISSUES

There are various hardware design considerations that may impact the ability to interchange various members of the CPU family. With proper design, these considerations can be dealt with no real system impact.

Slow Bus Turn

Bus turn is the amount of time allowed to change mastership on the A/D bus of the processor. In general, a read followed by a write can cause a change in bus direction in one-half bus cycle. At 33MHz, this is 15ns.

The system designer may implement an architecture which, by using appropriate transceivers and control signals, can tolerate a rapid bus turn. Alternatively, the designer may desire to increase the minimum amount of time.

Although the R3081 includes a bit in the Config register to slow the bus, this technique does not work with the R3051. Instead, the hardware technique of using BusReq to insure a longer tri-state time is recommended. This technique is described in applications note AN-97.

Coherent DMA

The R3081 includes a hardware interface to insure cache-coherency in systems using DMA. This interface is unavailable in the R3051.

Many MIPS applications perform multi-master cache coherency via software techniques, and thus do not require hardware-based coherency. While hardware-coherency will improve the performance of some applications, relying on software (which may, for example, flush the entire data cache once a DMA operation is completed to insure coherency. This technique will function equally well with either the R3051 or R3081.

Floating-Point Interrupt

The R3081 uses one of the interrupt input pins to report exceptions to the CPU. The hardware should reserve one of the input pins for this function, and provide logic or pull-up resistors to insure that this input is held HIGH for an R3051 or R3052.

CpCond(1)

The R3081 uses this input to report the results of comparisons back to the CPU; thus, the external input pin is ignored. R3051 systems should provide a pull-up resistor for this pin. Earlier in this applications note, a method to use this pin to determine the presence or absence of an FPA was described.

Reset Mode Vectors

Both the R3051 and R3081 use the same basic technique to perform reset mode selection of various options. Figure 3 illustrates the mode vector logic for the R3081. Note that for the R3051, Int(5:3) mode vectors are reserved, and must be held HIGH during reset.

Options include:

- *Tri-state.* This option is used to perform board testing, and is available in all devices.
- *BigEndian.* This option selects the data byte ordering convention, and is available in all devices.
- *Data Block Refill.* This option selects single versus four-word refill on data cache misses. Although this option is available in all devices, software (via the config register) can dynamically change the value for the R3081.
- *Coherent DMA Enable.* This option enables the coherent DMA interface of the R3081. For the R3051, this input must be HIGH at reset.

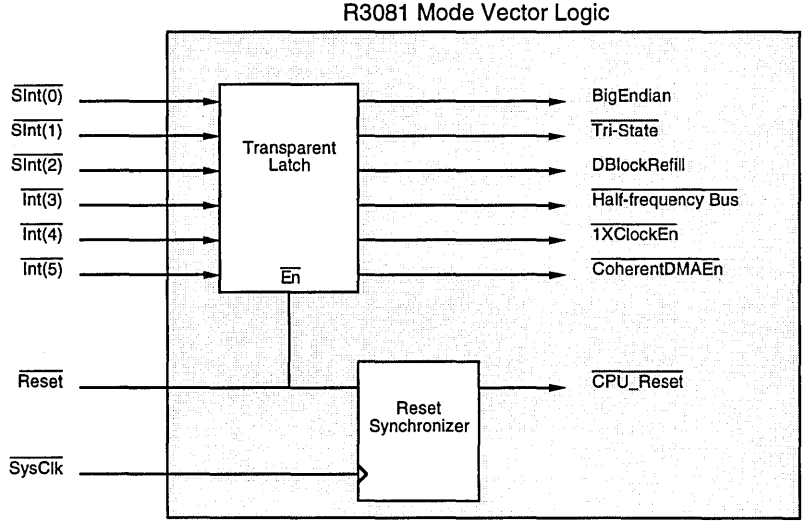


Figure 3. R3081 Mode Vector Assignment

- *1x Clock Mode.* This option instructs the R3081 that the input clock provided is at the CPU operation frequency, rather than at twice the frequency. In the R3051, only the "2x" clock is available, and this vector must be held HIGH.
- *Half-frequency Bus.* This option instructs the R3081 to operate its bus interface at one-half the execution rate. This option is unavailable in the R3051, and must be held HIGH at reset.

In order to design a system to accommodate either an R3051 or R3081, it may be desirable to include jumpers for the R3081-only options. Thus, when an R3081 is included in the design, various of the hardware options may be changed. This may open up other upgrade strategies, such as the clock doubling capability described earlier.

SUMMARY

By following a few simple rules, the system designer can implement a base R3051 system which can easily upgraded to higher performance. Upgrade options include more amounts of cache on-chip, the addition of hardware floating-point, and increases of frequency. With the R3081 half-frequency bus mode, the operation frequency of the execution engine can be substantially increased while maintaining the same (or even slower) bus interface frequency.

Thus, the IDT RISController family effectively reduces the time to market of new product families, and maximizes engineering return on investment by enabling one design effort to result in multiple end products.

Integrated Device Technology, Inc.

INTERRUPT HANDLER FOR THE IDT79R3051 RISCONTROLLER™ FAMILY

APPLICATION NOTE AN-131

by Dean Smith

INTRODUCTION

The reader is encouraged to refer to Chapter 5 of the IDT79R3051 RISController Hardware User's Manual for a thorough description of IDT RISController exception handling. In addition, the MIPS Programmers Handbook illustrates two alternative methods for interrupt prioritizing. This application note illustrates a third much faster method specific to the IDT RISController Family, as detailed in the Appendix - 'R3051/2 Priority Based Nested Interrupt Handler'. The corresponding latency cycles for this example interrupt handler are quantified in Table 1.

R3051/2	Service Latency	Restart Latency
Priority 1	4	9
Priority 2	14	13
Priority 3	16	13
Priority 4	19	13
Priority 5	25	14
Priority 6	25	14

3158 tbl 01

Table 1. IDT79R3051/2 Interrupt Latency (in cycles)

The following assumptions apply to the latencies quantified in Table 1:

- The corresponding algorithm/code is detailed in the Appendix.
- Service Latency, Restart Latency are as defined in this application note.
- The code and stack are resident in the R3051 on-chip cache.
- The R3051 pipeline is in a 'run' state at the instant the interrupt is detected.
- A higher priority interrupt is not already in progress.
- Service is not interrupted by a higher priority interrupt.
- Service is not interrupted by any other type of exception.
- Only 1 register is needed by PRIORITY 1,2,3,4 service routines.
- Only 3 registers are needed by PRIORITY 5,6 service routines.

The interrupt handler detailed in the Appendix is specific to the R3051/2. However, much of the content detailed in this application note equally applies to the other RISController family members with only minor code modifications being required. Where applicable these differences in the family members are detailed.

R3051 EXCEPTION MODEL

External interrupts are just one class of R3051 exceptions. The R3051 implements a 'precise' exception model. By definition, precise exceptions imply that exact processor context and the cause of the exception are known. In addition, the current process does not advance state (ie. all subsequent instructions are aborted) until the corresponding interrupt is serviced.

The following automatically occurs when the R3051 detects an interrupt:

- The current process is halted.
- The Exception Program Counter is loaded with the return address for the current process.
- The Cause Register is loaded with exception cause information.
- The Status Register KUc bit is cleared (ie. enter 'kernel mode').
- The Status Register IEc bit is cleared (ie. disable subsequent interrupts).
- Execution is continued at the General Exception Vector.

These activities preserve the necessary processor context to implement a precise exception model. The R3051 processor makes no assumptions about an external interrupt cause or servicing techniques. For instance, R3051 registers are not automatically stacked upon detection of an interrupt since this often causes unnecessary service latency. Instead, the software designer is allowed to fine-tune response to the corresponding service requirements. This technique allows for extremely fast interrupt handling.

INTERRUPT SERVICE LATENCY

Interrupt Service Latency is defined as the cycle count from the assertion of an external interrupt to the beginning of the corresponding service routine. This latency includes three components;

1) pipeline latency to the General Exception Vector
2) exception type decode
3) preserving context.

PIPELINE LATENCY:

The R3051 pipeline must be in a 'run' state for an interrupt to be recognized. Thus, pipeline stalls caused by such events as cache misses and multiply/divide interlock cycles delay detection of an interrupt. Once an interrupt is detected, the address of the General Exception Vector will be the next instruction fetched.

The R3051 has two types of external interrupt pins; ìsynchronousî interrupts, and ìdirectî interrupts. The ësynchronousí interrupts are internally synchronized and thus may be driven by an asynchronous source, with a corresponding pipeline latency to the General Exception Vector of two cycles. The ëdirectí interrupts are not internally synchronized by the processor, and thus must be externally synchronized. As a result, these interrupts have only a one cycle pipeline

latency to the General Exception Vector and are most useful for interrupting agents which operate off the R3051 SysClk output.

EXCEPTION TYPE DECODE:

The General Exception Vector is the start address for all types of R3051 exception handlers (except RESET and UTLB Miss exceptions) - interrupts being just one classification. Thus, the exact exception type must first be decoded before servicing can begin. This is typically accomplished by software interrogation of the R3051 Cause Register. The following example code details this procedure:

```
mfc0  k0,C0_CAUSE;      # k0 = CR(Cause Register)
sw    t1,t1_OFF*4(sp)   # use delay slot to stack gpr t1
and   k1,k0,EXC_MASK;   # isolate ExcCode field of CR
lw    v0,cause_table(k1); # fetch cause start address
and   k1,k0,IP_MASK;    # isolate IP field of Cause Register
j     v0;               # go to Exception handler start address
                        # (v0 = INT_EXTERN, if an interrupt)
sra   k1,k1,8;          # shift IP field 8 bits for word address

INT_EXTERN:
lw    v0,IP_table(k1);  # fetch service routine start address
sw    v1,v1_OFF*4(sp);  # use delay slot to stack gpr v1
j     v0;               # jump to corrresponding int(n) service
sw    t0,t0_OFF*4(sp);  # use delay slot to stack gpr t0
```

Even faster exception type decode can be achieved by using the R3051's BrCond(n) input pins. The MIPS ISA contains conditional branch instructions based upon the value of BrCond(n). These pins can be physically connected to interrupt pins for extremely fast decode. The following example code details this procedure:

```
bc0t  PRIORITY_1;       # int(0)?
sw    k0,EPC_OFF*4(sp); # stack EPC (use branch delay slot).
bc1t  PRIORITY_2;       # int(1)?
sw    k1,SR_OFF*4(sp);  # stack SR (use branch delay slot)
bc2t  PRIORITY_3;       # int(2)?
sw    v0,v0_OFF*4(sp);  # stack v0 (use branch delay slot)
bc3t  PRIORITY_4;       # int(3)?
sw    t0,t0_OFF*4(sp);  # stack t0 (use branch delay slot)
```

The interrupt handler detailed in the Appendix is specific to the R3051/2 by making use of the four available BrCond(n) pins. Minor code modifications are required for the other RISController family members due to the different number of available BrCond(n) pins for each.

RISController Family Member	Number of BrCond(n) pins
R3051/2	four
R3071/81	three
R3041	two

3158 tbl 02

PRESERVING CONTEXT:

Detection of an exception causes the R3051 to automatically disable subsequent interrupts. This makes it possible for immediate servicing of the interrupt without preserving Cause Register, Status Register, or Exception Program Counter context. Note that care must be taken by the software designer to ensure that execution of the interrupt handler and service routine do not generate any other type of exception. If 'nested' interrupts are allowed, then the Status Register and Exception Program Counter must be stacked. Otherwise the handling of the original interrupt can not be resumed. The IntMASK field of the Status Register can then be modified to re-enable higher priority interrupts. The following example code details this procedure:

```
bc0t  PRIORITY_1;       # int(0)?
sw    v0,v0_OFF*4(sp);  # use delay slot to stack gpr v0.

# PRIORITY 2,3,4,5,6 - must stack context for servicing of higher
priority interrupts.
subu  sp,sp,exc_stack_sz; # Initialize Stack.
mfc0  k0,C0_EPC;        # k0 reserved for kernel processes
mfc0  k1,C0_SR;         # k1 reserved for kernel processes
sw    k0,EPC_OFF*4(sp); # stack EPC.
mfc0  k0,C0_CAUSE;      # k0 = CR(Cause Register).
sw    k1,SR_OFF*4(sp);  # stack SR.
bc1t  PRIORITY_2;       # int(1)?

PRIORITY_2:
# Stack additional General Purpose Registers needed for servicing.
# re-enable int(0) - higher priority.
li    v0,x0000401;
mtc0  v0,C0_SR;
# PRIORITY 2 service here: . . .
```

Note that registers k0 and k1 are immediately available for interrupt handling. These registers need not be stacked since MIPS compiler and assembler conventions reserve k0 and k1 for kernel processes, and since subsequent interrupts are disabled during any interrupt handlerís use of these registers. However, the interrupt handler must stack any General Purpose Registers to be used for interrupt servicing. The number of registers required is of course interrupt service specific. The delay slots immediately following branch and load instructions are convenient locations to stack context without adversely affecting service latency.

Other features of the R3051 also help to minimize interrupt service latency. For instance, the on-chip cache is 'physically'

indexed. This means that virtual-to-physical address translation is performed prior to cache addressing. As a result, cache flushing is not required on a context switch (ie. jump to interrupt service routine). Other processors implement virtually indexed caches thereby dramatically slowing context switch performance. Also of importance is the R3051 PID (Process ID) field associated with each entry of the TLB (Translation Lookaside Buffer). The 'Extended' memory management option uses an on-chip TLB as a hardware cache for software managed page tables. The PID is compared to the contents of each TLB entry at the time of address translation, thereby providing a mechanism for multiple processes to share the TLB even if identical virtual page numbers are encountered. As a result, TLB flushing is not required on a context switch.

INTERRUPT RESTART LATENCY

Interrupt Restart Latency is defined as the cycle count from the end of the interrupt service routine to the restart of the parent process. This latency includes two components;

1) context restore
2) pipeline refill.

Context Restore:

Any processor context stacked prior to interrupt servicing must be restored after servicing is complete. Then the stack pointer must be restored to its previous value. Finally, execution can then return to the parent process. The following example code details this procedure:

```
li    k0,x000xxx0;          # disable int's prior to context restore.
mtc0  k0,C0_SR;
lw    k1,SR_OFF*4(sp);
lw    v0,v0_OFF*4(sp);      # restore gpr v0
lw    k0,EPC_OFF*4(sp);     # acquire parent process return address
addu  sp,sp,exc_stack_sz;   # restore stack.
mtc0  k1,CO_SR;             # restore SR(Status Register).
j     k0;                   # return to parent process
rfe;
```

Note that interrupts must be disabled prior to context restore. This is because k0 and k1 are not preserved prior to use by the interrupt handler. Otherwise, the context of these registers would be lost if another interrupt occurs during context restore for the current interrupt.

Pipeline Refill:

Figure 1 illustrates R3051 pipeline refill following an interrupt. Upon detection of an external interrupt, the three instructions less advanced than the ALU stage are aborted. These instructions must be restarted upon return to the parent process. This three cycle penalty must be considered when calculating the Interrupt Restart Latency.

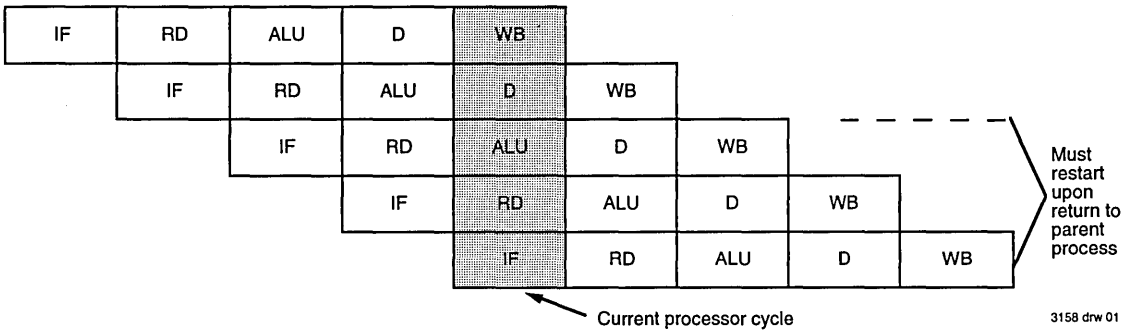


Figure 1. IDT79R3051 Instruction pipeline.

APPENDIX—R3051/2 PRIORTY-BASED NESTED INTERRUPT HANDLER

```
# This is an example R3051/2 priority-based nested
  interrupt handler.
# Other RISController Family members require minor code
  changes due to the different number
# of available BrCond(n) inputs
#     – prioritize up to four R3051/2 interrupts
#     – prioritize up to three R3081 interrupts
#     – prioritize up to two R3041 interrupts
# BrCond(n) is tied to corresponding int(n). This allows for
  fast interrupt decode:
# The following interrupt priority is assumed:
#     PRIORITY 1 = Int(5) = BrCond(0)
#     PRIORITY 2 = Int(4) = BrCond(1)
#     PRIORITY 3 = Int(3) = BrCond(2)
#     PRIORITY 4 = SInt(2) = BrCond(3)
#     PRIORITY 5 = SInt(1)
#     PRIORITY 6 = SInt(0)
```

```
# Exception causes execution to jump here:
  General Exception Vector.
.set    noreorder              # assembler directive–disable
                               pipeline scheduling.
bc0t    PRIORITY_1;            # PRIORITY 1?
subu    sp,sp,exc_stack_sz;    # use delay slot to Initialize
                               Stack.

# PRIORITY 2,3,4,5,6: Must stack CP0 context
  to allow for nested servicing.
sw      v0,v0_OFF*4(sp);       # stack gpr v0.
mfc0    k0,C0_EPC;             # k0 reserved for kernel processes
                               - no need to stack.
mfc0    k1,C0_SR;              # k1 reserved for kernel processes
                               - no need to stack.
sw      k0,EPC_OFF*4(sp);      # stack EPC.
mfc0    k0,C0_CAUSE;           # k0 = CR(Cause Register).
sw      k1,SR_OFF*4(sp);       # stack SR.
bc1t    PRIORITY_2;            # PRIORITY 2?
and     k1,k0,EXC_MASK;        # isolate ExcCode field of CR.
bc2t    PRIORITY_3;            # PRIORITY 3?
lw      v0,cause_table(k1);    # fetch exception cause start address.
bc3t    PRIORITY_4;            # PRIORITY 4?
and     k1,k0,IP_MASK;         # isolate IP field of Cause Register.

# PRIORITY 5,6: Evaluate Cause Register, jump to
  Exception cause start address.
# (process already started by using Branch Delay Slots
  above)
j       v0;                    # jump to Exception cause start
                               address.
sra     k1,k1,8;               # shift right 8 bits to create word
                               address.

# Exception cause start address = INT_EXTERN if an
  interrupt.
INT_EXTERN:
lw      v0,IP_table(k1);       # fetch Interrupt routine start address
                               from IP_table.
sw      v1,v1_OFF*4(sp);       # use delay slot to stack gpr v1.
j       v0;                    # jump to PRIORITY_5 or 6, per IP
                               field of Cause Register.
sw      t0,t0_OFF*4(sp);       # use delay slot to stack gpr t0.

PRIORITY_1:
sw      v0,v0_OFF*4(sp);       # stack gpr v0.
# Stack any additional gpr's needed for PRIORITY 1
  interrupt servicing.
# k0 & k1 are also available for PRIORITY 1 servicing.
# PRIORITY 1 service here.
        •
        •
        •
# Restore any gpr's used.
lw      v0,v0_OFF*4(sp);       # restore gpr v0.
# Restore Stack and return to parent process.
addu    sp,sp,exc_stack_sz;    # restore sp(Stack Pointer).
mfc0    k0,C0_EPC;
nop;
j       k0;                    # return from int svc.
rfe;

PRIORITY_2:
# Stack gpr's needed for PRIORITY 2 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1 (higher priorty interrupt).
li      v0,x0008001;           # re-enable PRIORITY 1—Int(5). 2cycle
                               inst'n.
mtc0    v0,C0_SR;
# PRIORITY 2 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
  process.
li      k0,0x00000000;         # disable interrupts prior to context
                               restore. 1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);       # restore gpr v0.
nop
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz;    # restore sp(Stack Pointer).
mtc0    k1,CO_SR;              # restore SR(Status Register).
j       k0;                    # return from int svc.
rfe;

PRIORITY_3:
# Stack gpr's needed for PRIORITY 3 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1,2 (higher priorty interrupts).
li      v0,x000C001;           # re-enable PRIORITY 1,2 - Int(5,4).
                               2cycle inst'n.
mtc0    v0,C0_SR;

# PRIORITY 3 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
  process.
li      k0,0x00000000;         # disable interrupts prior to context
                               restore. 1 cycle inst'n.
```

```
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);    # restore gpr v0.
nop
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz; # restore sp(Stack Pointer).
mtc0    k1,CO_SR;           # restore SR(Status Register).
j       k0;                 # return from int svc.
rfe;
PRIORITY_4:
# Stack gpr's needed for PRIORITY 4 interrupt servicing.
# v0 already stacked.
# Re-enable PRIORITY 1,2,3 (higher priorty interrupts).
li      v0,x000E001;    # re-enable PRIORITY 1,2,3 - Int(5,4,3).
                        2cycle inst'n.
mtc0    v0,C0_SR;
# PRIORITY 4 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
process.
li      k0,0x00000000;      # disable interrupts prior to context
                            restore. 1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);    # restore gpr v0.
nop
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz; # restore sp(Stack Pointer).
mtc0    k1,CO_SR;           # restore SR(Status Register).
j       k0;                 # return from int svc.
rfe;

PRIORITY_5:
# Stack gpr's needed for PRIORITY 5 interrupt servicing.
# v0,v1,t0 already stacked.
# Re-enable PRIORITY 1,2,3,4 (higher priorty interrupts).
li      v0,x000F001;        # re-enable PRIORITY 1,2,3,4
                            - Int(5,4,3,2). 2cycle inst'n.
mtc0    v0,C0_SR;
# PRIORITY 5 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
process.
li      k0,0x00000000;      # disable interrupts—1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);    # restore gpr v0.
lw      v1,v1_OFF*4(sp);    # restore gpr v1.
lw      t0,t0_OFF*4(sp);    # restore gpr t0.
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz; # restore sp(Stack Pointer).
mtc0    k1,CO_SR;           # restore SR(Status Register).
j       k0;                 # return from int svc.
rfe;

PRIORITY_6:
# Stack gpr's needed for PRIORITY 6 interrupt servicing.
# v0,v1,t0 already stacked.
# Re-enable PRIORITY 1,2,3,4,5 (higher priorty
interrupts).
li      v0,x0007F801;       # re-enable PRIORITY 1,2,3,4,5
                            - Int(5,4,3,2,1). 2cycle inst'n.
mtc0    v0,C0_SR;
# PRIORITY 6 service here.
        •
        •
        •
# Restore SR, gpr's used, Stack, and return to parent
process.
li      k0,0x00000000;      # disable interrupts - 1 cycle inst'n.
mtc0    k0,C0_SR;
lw      k1,SR_OFF*4(sp);
lw      v0,v0_OFF*4(sp);    # restore gpr v0.
lw      v1,v1_OFF*4(sp);    # restore gpr v1.
lw      t0,t0_OFF*4(sp);    # restore gpr t0.
lw      k0,EPC_OFF*4(sp);
addu    sp,sp,exc_stack_sz; # restore sp(Stack Pointer).
mtc0    k1,CO_SR;           # restore SR(Status Register).
j       k0;                 # return from int svc.
rfe;

.set reorder                # assembler directive - enable pipeline
                            scheduling.
```

Integrated Device Technology, Inc.

LOW POWER R3041 FOR WLAN APPLICATIONS

APPLICATION NOTE AN-138

By Robert Napaa

INTRODUCTION

In the past 10 years, the use of computer networks has increased many fold. Users at home and in the office need constant access to on-line corporate information such as data bases, files and email, etc. During the same time frame advances in low power microprocessors and battery technology gave rise to a new class of computing machines. These Laptop Computers, Notebooks and Personal Digital Assistants (PDAs) need to access a company's wired backbone infrastructure LANs to give users complete mobility. Mobile users can access the company LAN infrastructure through Wireless LAN access or Digital Cellular access. The Wireless LAN (WLAN) is used locally within the company office buildings or the parking lot. It is a high speed connection (1 - 20 Mbits/sec) with a limited radius of about 50 meters or so. Cellular access is used when the user is on the road. Cellular access is at much lower speeds (10 - 50 Kbits/sec) and is usually provided through cellular carriers, exactly like cellular phones and pagers. This paper describes the Wireless LAN implementation.

WIRELESS LANS

Wireless LANs (WLANs) provide the mobile users access to the company wired LAN infrastructure. They offer great flexibility because connection can be established immediately. There is no need for a wire or a wall connector. Users are not constrained to a particular work area. They can roam freely from one location to the other while maintaining full access to the wired LAN backbone. Furthermore, active sessions don't get interrupted as users roam around.

WLANs usually consist of two elements: the Access Point and the Mobile Unit. The Access Point provides connections to the wired LAN infrastructure. The Mobile Unit is the portable computer with an adapter to the wireless world. Figure 1 illustrates the topology for a WLAN network connected to the existing LAN backbone.

If connection to the wired LAN is not needed, Ad-Hoc WLAN networks can be created and dismantled among the Mobile Units as needed without having to change the existing wiring network. The WLAN standards are in the final definition stages in the IEEE 802.11 proposals.

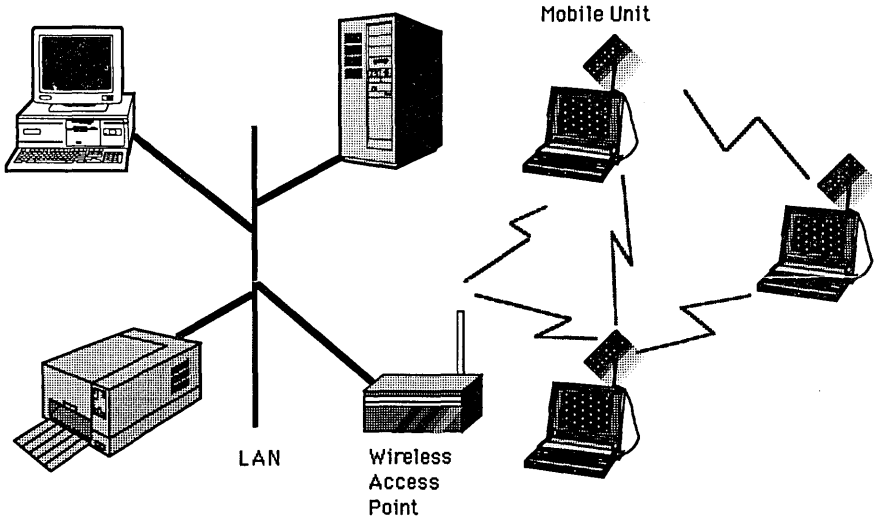


Figure 1. Wireless LAN Topology
802.11 Proposed Standards

The IEEE 802.11 proposed standards to be finalized in the middle of 1995 specify the lower two layers of the Open System Interconnects (OSI) seven layer model.

The physical layer interface is specified for 1 - 2 Mbits/sec rates using spread-spectrum techniques. Either direct sequence or frequency hopping methods can be used. The transmission can be carried by any of the ISM (Industrial, Scientific and Medical) frequency bands. The proposals call for even higher bit rates in the future (up to 20 Mbits/sec) as the technology becomes more mature.

There is also another proposal for an infrared physical layer.

In the wireless world it is difficult to deal with collision detect on the hardware level like wired protocols such as Ethernet. The main reason for this is that the transmitting station can't detect a collision, because its own transmitted signal overcomes signals from any other stations. Therefore, the collision detect for WLANs is implemented in the MAC layer interface, using a simple handshaking mechanism between the sending and the receiving stations. The sending station issues a Request_To_Send signal and waits for the Clear_To_Send signal from the receiving station. These two signals carry other embedded information to inform other potential stations to wait until the transmission is done. The transmitting station sends the data and waits for the Acknowledge signal from the receiving end to complete the transmission.

Finally, the proposed standards specify that the WLANs should function well in both a distributed control or a point control environment. In a distributed control environment, the WLAN network control is distributed among all the units (Access Points and Mobile Units). This allows ad-hoc networks among several Mobile Units to be formed without the need for any additional control or management. In the point control environment, a centralized node takes control of the WLAN management and allows only one unit to "talk" at a time.

THE MOBILE UNIT

The Mobile Unit consists of a host computer (usually a portable one) and a WLAN adapter. The WLAN adapter implements several tasks. It negotiates for the access to the airwaves and implements the MAC layer protocol. It also shares in the WLAN network control and management in a distributed environment. The WLAN adapter also plays a major role in roaming support. Roaming support for the WLANs is completely different from that for the cellular phones. In the cellular world, the base station determines when to hand-off an active session to the following base station. This is usually determined according to the quality of the signals received. In the WLAN world, the Mobile Unit determines when to switch to another Access Point that it can "hear" better. Again, this decision is based on the quality of the signals received. This transition from one Access Point to the other shouldn't interrupt any active sessions. All these tasks suggest that the intelligence must be built on the WLAN Adapter.

There are several implementations for the WLAN Adapters. Some vendors implement them as add-on cards to host computers while others implement them as PCMCIA cards for portable platforms. This paper concentrates mostly on the PCMCIA implementation. It is important to note that all the relevant concepts and requirements still apply to the add-on cards.

The PCMCIA is usually implemented using an "intelligent" PCMCIA card with an antenna attachment. The "intelligence" is in a microprocessor or microcontroller on the card. This implementation relieves the host CPU from the real-time requirements of servicing the radio. Furthermore, the host CPU is usually of limited compute power that is used for other system functionality. The antenna attachment could be part of the card itself or an external component.

REQUIREMENTS FOR PCMCIA CARDS

The PCMCIA standards place stringent requirements on the selection of the components. The devices selected must have low EMI emission and fit inside a Type I, II or III card. They must also consume a minimum amount of power and have power management capabilities. There is usually a very limited power budget available from the host computer to the PCMCIA slot. The power allocated to the PCMCIA slots varies from vendor to vendor and usually ranges from 1 - 1.25 Watt. These requirements point toward a reduction in the number of components used. This favors a software solution where much of the functionality that was implemented in dedicated hardware is now implemented in software. This software approach requires the use of a powerful microprocessor to implement all the different tasks. It offers the advantage of great flexibility and adaptability since only the software changes to adapt to new standards. Similarly, the use of a high performance microprocessor allows the software to add more functionality without incurring the cost of dedicated hardware.

THE IDT R3041

The IDT R3041 is a 32-bit RISC microprocessor designed for embedded applications. It is based on the MIPS R3000A microprocessor and is highly integrated, with large on-chip caches. There are 2 KBytes of Instruction cache and 512 Bytes of Data Cache. At 10 MHz it has a compute engine of about 8 MIPS. It is also available at higher speeds, up to 33 MHz. The R3041 features a flexible bus interface that connects directly to 8, 16, or 32-bit devices as well as memory. Most of the system control signals are also implemented on chip to reduce the external logic needed. It is available in a 100-pin TQFP package to fit the form factor of Type II PCMCIA cards.

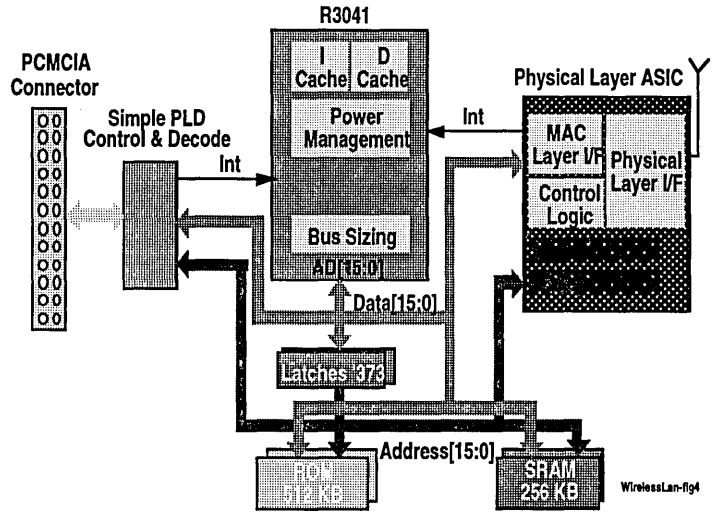


Figure 2. PCMCIA WLAN Adapter Based on the R3041
PCMCIA WLAN with the R3041

The design of a WLAN Adapter PCMCIA card based on the R3041 is illustrated in Figure 2. The components requirements for this design are minimal. The R3041 and the memory system are at the center of the design. The EPROMs store the execution code, while the system memory is implemented using SRAMs. The PCMCIA interface is accomplished using a simple PLD combined with a "soft" solution. To the host computer, the software emulates the I/O space of the PCMCIA cards. The system control logic is also part of the PLD. The wireless interface is implemented in a separate ASIC that handles the physical layer interface.

R3041 POWER CONSUMPTION

The R3041 is designed for power-sensitive applications. At 10 MHz, it only consumes about 0.25 Watts, which makes it ideal for portable applications. Furthermore, the R3041 can operate in a "reduced frequency" mode that is under the control of the software. In this mode, the internal and external clocks are divided by a power of two factor. By reducing the frequency to 1 - 2 MHz, the power consumption is almost halved to about 0.13 Watt, as is illustrated in Figure 3.

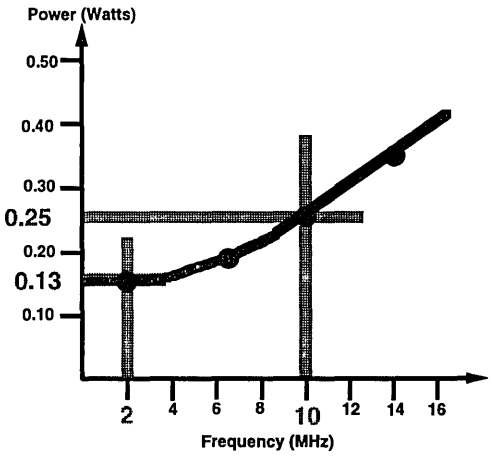


Figure 3. R3041 Power Consumption

This combination of low power consumption and power management fits extremely well with the WLAN data traffic, which is bursty in nature. Empirical data shows that, over a period of time, 25% of the time is spent servicing data (receive and transmit), while the remaining 75% is idle time. During this idle time, the WLAN Adapters only listen to passing traffic.

Software maximizes the advantage of this situation by putting the R3041 in the "reduced frequency" mode when there is no data to service. This reduces the average power consumption to even less than its already low levels. On the

average, the R3041 will consume about 0.16 Watts. This is obtained by taking 75% of 0.13 Watts plus 25% of .25 Watts. This average power consumption is well below the 1 - 1.25 Watts available from the host.

REAL-TIME INTERRUPT RESPONSE

The real-time interrupt response of the R3041 is a major part of this design. Both the "soft" PCMCIA and the MAC layer implementations are interrupt-driven and rely on the speed at which the R3041 responds to interrupts. An interrupt-driven architecture is much more dynamic than a polled architecture, since the system responds only to the port that needs service. It is a much more efficient use of the system resources.

The R3041 at 10 MHz can respond to interrupts in less than 3 µsec. This includes recognizing the exception, preserving the state, decoding the exception and restoring the state at the end of the exception. Figure 4 illustrates a sample code that accomplishes these steps. This fast response time, combined with the interrupt service routine executing from the caches, removes the need for dedicated hardware to implement the PCMCIA or the MAC layer interfaces.

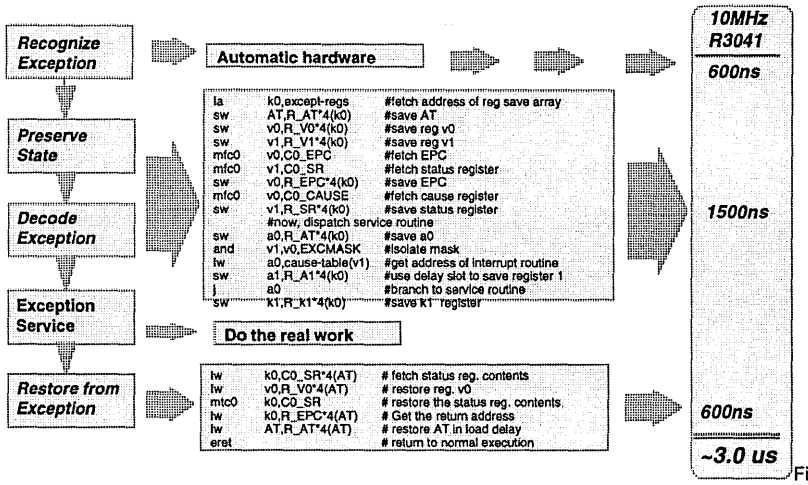


Figure 4. Sample Code for Interrupts

MAC Layer Implementation

The IEEE 802.11 standards specify a unique 48-bit IEEE device code for WLANs. A password might also be associated with a given ad-hoc network. The MAC layer must checks the hashing table for the address and the password to determine the validity of the incoming data. Similarly, it must issue the new address and password when transmitting messages. In the WLANs, the collision detect mechanism is also implemented in the MAC layer. A software solution can be much more flexible than a hardware one. Software can adapt to the emerging standards, while not becoming locked into dedicated hardware modules.

The "soft" MAC layer implementation takes advantage of the fast interrupt response time of the R3041. The internal caches play a major role in speeding up the software execution. The data cache stores the hashing tables, while the instruction cache stores the interrupt service routines. In this case, the external bus is accessed only to bring data into the R3041 and to write data to the external devices. The data manipulation is done on the fly while reading and writing to the external devices.

Receiving the Data

The WLAN Adapter receives the data from the external world when in the receiving mode. At 1 Mbit/sec interface a byte will be available every 7.8 µsec. The physical layer ASIC combines the bits into bytes and gathers four bytes before interrupting the R3041. As a result the R3041 is interrupted every 31.2 µsec. It takes the R3041 about 3 µsec to respond to the interrupt. The remaining 28.2 µsec are used to read the incoming bytes, manipulate the header and store the data in memory. It takes about 10 clock cycles to read the four bytes from the physical layer interface and a similar 10 clock cycles to write the data into memory. These two operations take about 2 µsec. This interface (excluding the header manipulation) consumes about 16% of the R3041 compute power. The remaining 26 µsec are used to manipulate the incoming data on the fly. Other functionality such as network management and roaming support can be serviced during this time.

Transmitting the Data

The transmission operation is the reverse of the receiving operation. The same analysis applies. WLAN protocols are full duplex, but since the transmission and reception are usually asymmetrical, the analysis for either should be used only.

PCMCIA INTERFACE

Similar to the software implementation of the MAC layer protocol, the PCMCIA interface is emulated in software. The basic approach is to emulate the I/O space of the PCMCIA cards using interrupts to the R3041. When the host requests a service, an interrupt is generated to the R3041. The R3041 determines whether it is a read or a write operation and acts accordingly.

THE SOFT ADVANTAGE

Implementing most of the functionality in software offers greater flexibility and adaptability. It becomes much easier and less costly to adapt to new standards without the need for an entire hardware redesign. Similarly, using a high level language offers the freedom to port the code developed to other platforms with minimal modifications. It is also much easier to maintain.

The "soft" approach allows to add more functionality for product enhancement and differentiation. For example, software compression and decompression can be used to increase the effective bandwidth by requiring less time for transmission and reception. Software provides the support for time-bound data such as voice and video for multimedia applications. Furthermore, encryption/decryption or other algorithms can be added in software to provide for secure systems for example transmission and receptions with little impact on the hardware design.

THE ACCESS POINT

The Access Point implements the same WLAN protocols as the Mobile Unit with the addition of a wired access to the company backbone. To preserve the investment in the design of the WLAN Adapter, most of the Mobile Unit modules should be reused in the Access Point. The physical layer ASIC is a good example of this. It saves time if the entire Mobile Unit code base can be reused for the Access Point. This assumes the use of the same architecture and/or the same microprocessor. The R3041 offers both of these choices. It can be used as is in the Access Point design while executing at higher frequencies. In that case there would be very little modifications to the software module written. The hardware would be modified to include the wired interface. Figure 5 illustrates the design of the Access Point around the R3041.

A second possibility is to use a more powerful microprocessor from the same family. The R3041 is 100% software compatible with the IDT RISControllers™ family. This family offers a wide selection of price/performance microcontrollers that fit most of the embedded applications. The family offers several options including on-chip instruction and data caches, hardware floating-point unit and a flexible bus interface. The Access Point can then be redesigned for use with another member of the family. However, the investment in the software will be maintained because the software can be reused without modifications.

CONCLUSION

The R3041 offers all the advantages of a software implementation. It can be used for both the Access Point design and the PCMCIA WLAN Adapter with minimum modifications to the hardware and/or the software. The "soft" approach reduces the need for dedicated hardware modules. It offers the capability to adapt to new standards. Additional functionality can be added without a major impact on the hardware.

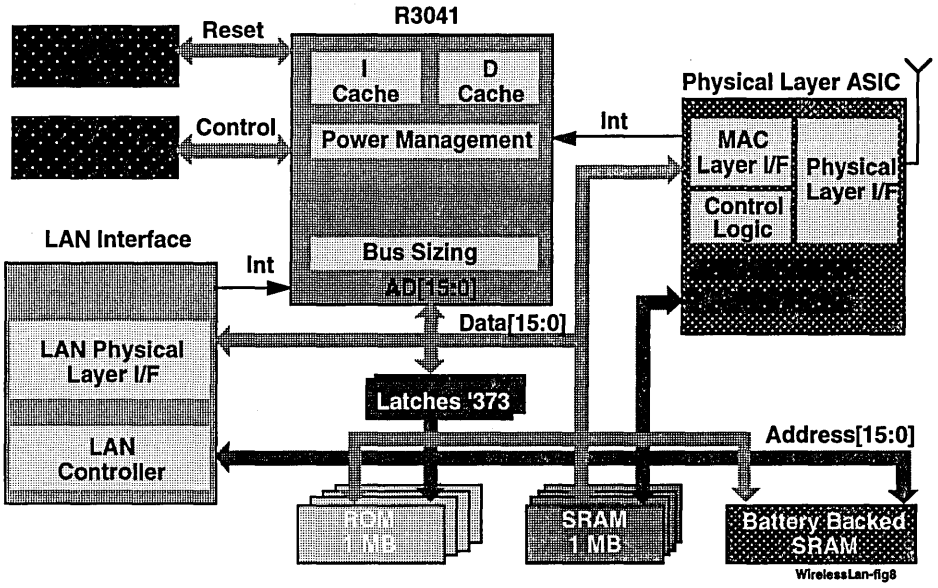


Figure 5. Access Point based on the R3041

Finally, the RISControllers™ family offers a wide spectrum of price/performance choices for the embedded applications. The software investment is preserved 100% across the entire family.

Integrated Device Technology, Inc.

DESIGNING MEMORY SUBSYSTEMS FOR THE R3051™ FAMILY

CONFERENCE PAPER CP-05

By Bob Napaa

INTRODUCTION

The IDT79R3051™ RISController™ family utilizes a high-performance computing core to achieve high performance across a variety of applications. Further, the amount of cache incorporated in the R3051 family allow these CPUs to achieve very high performance even with simple, low-speed low-cost memory subsystems.

The R3051 and the R3081™ RISController CPU families include a full R3000A core RISC processor, and thus are fully compatible with the standard MIPS processors. In order to provide high band-width to the CPU core, the families also incorporate relatively large instruction and data caches. The external memory interface from the R3051 family is very flexible and allows a wide variety of implementations depending on the price/performance goal of the application. The R3081 is upward compatible to the R3051 family with the same footprint and bus interface and the benefit of larger caches and a hardware floating-point coprocessor.

This paper will discuss the cost and performance impact of various trade-offs, and provide a concrete design of a DRAM memory subsystem around the R3051 and the R3081. This paper will specifically address the trade-offs between high-performance and low-cost memory systems, the impact of a high-frequency system on the memory interface and the impact of systems which are intended to be field upgradeable.

DIFFERENT TYPES OF MEMORY

SRAM, DRAM and EPROM are today's industry standard for memory subsystems. EPROMs usually provide boot code in most systems and are much slower and more expensive than SRAMs or DRAMs. SRAMs are typically less dense and more expensive than DRAMs; however, they provide faster memory access time with a simpler interface and can be used in systems where performance (rather than cost) is the primary criterion. DRAMs are the most popular choice for main memory because of their position on the cost/performance curve and the densities in which they are available.

MEMORY SYSTEMS

Most of today's systems use one of two memory architectures: Non-Interleaved or Interleaved architectures. In this paper, a memory array is defined as the group of memory devices that produce a full width CPU data bus. For example a 16-bit data bus CPU requires 4 "x4" DRAMs to compose a memory array while a 32-bit data bus CPU requires 8 "x4" DRAMs to compose a memory array.

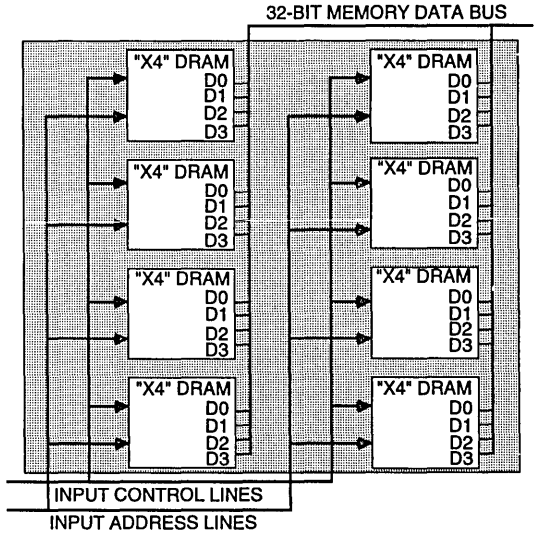


Figure 1a. Single-Bank Non-Interleaved System

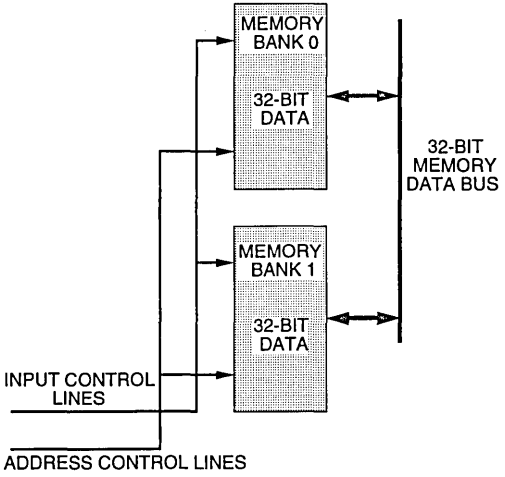


Figure 1b. Two-Bank Non-Interleaved System

In non-interleaved architectures, a memory bank consists of a single memory array with sequential addresses. Any read or write to a memory bank accesses a single location. Figure 1a illustrates the architecture of a single non-interleaved memory bank. Non-interleaved memory architectures are usually composed of multiple memory banks to satisfy the memory requirements of the system. In these topologies, the high order address lines select among the multiple memory banks and only one memory bank can be selected at a time. Figure 1b illustrates the architecture of a non-interleaved two banks memory system.

There are various types of interleaved architectures. The most popular one is the address interleaved. There are numerous variations of the address interleaved architectures. Mainly, 2-way address interleaved, 4-way address interleaved and so on. In a 2-way address interleaved architecture two memory arrays are grouped together in parallel to form a Super memory bank. This Super memory bank thus has double the data bus width and double the memory density of a single non-interleaved bank, and consists then of an even array and an odd array. A memory controller must be able to select both arrays together or independently based on the type of access. The memory controller uses the low order address bit to select between the two arrays. It must be able to direct the data path from every memory array independently to the CPU through some data buffers. Figure 2 illustrates the architecture of a 2-way interleaved single Super memory bank system. In a 4-way address interleaved architectures four memory arrays are grouped together in parallel to form a Super memory bank. This Super memory bank consists thus of four quarters. The memory controller must be able to select these four arrays together or independently using the two low

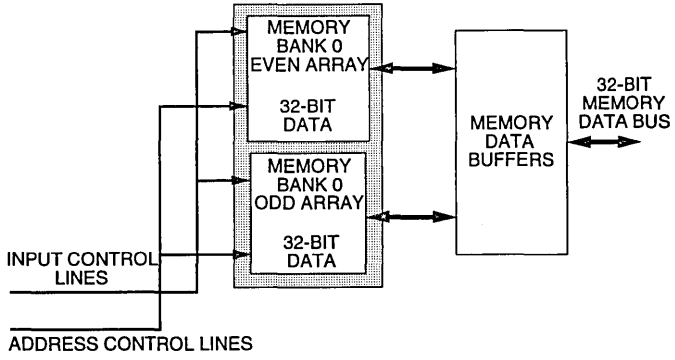


Figure 2. 2-Way Interleaved Single Super Memory Bank

order address bits. It must be able to direct the data bus of every quarter independently to the CPU through some data buffers.

Address interleaved memory systems are thus inherently more expensive than non-interleaved architecture since they require a much more complex memory controller and wider data paths. The basic amount of memory banks in address interleaved architectures is a multiple of the basic memory bank in non-interleaved architectures; however, for systems with large amount of memory, the same memory banks could be configured as interleaved or non-interleaved. The major advantage of interleaved systems lie in block of data elements accesses from/to the CPU. Interleaved systems can double or quadruple the memory band-width and thus dramatically improve the performance when the CPU reads or writes 4, 8, 16, 32... data elements at a time. Interleaved systems do not offer any advantage for single independent read or write accesses. Interleaved architectures are usually used in systems where performance (rather than cost) is of importance. For embedded cost sensitive applications, non-interleaved is usually the architecture of choice.

GENERAL DESCRIPTION OF THE DRAM SYSTEM AROUND THE R3051

The R3051 is designed around the R3000A MIPS RISC core and features a high level of integration with large on-chip instruction and data cache. It incorporates up to 8kB of instruction cache and 2kB of data cache. These relatively large caches achieve hit rates in excess of 90% and substantially contribute to the performance inherent in the R3051 family. The R3051 has also implemented on-chip a four-deep read and a four-deep write buffers that isolate the high frequency CPU core from the much slower external memory and modules. This high level of integration simplifies the interface between the R3051 and the external memory modules as is illustrated in Figure 3 and allows the use of low cost memory subsystems without penalizing the performance.

The R3051 family uses a double frequency input clock for its internal operation and provides a nominal frequency output clock for the external system. This output clock, Sysclk, synchronizes the external memory subsystems to the CPU. Memory transactions from the R3051 use a single, time multiplexed 32-bit address and data bus and a simple set of

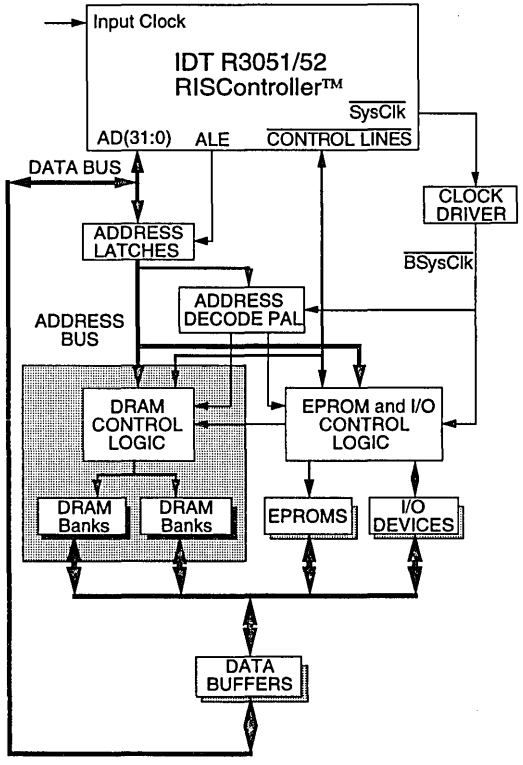


Figure 3. R3051 RISController Family-Based System

control signals. External logic then performs address demultiplexing and decoding, memory control, interface timing and data path control.

The system shown in Figure 3 is a 25MHz system with a 50MHz input clock. The R3051 interfaces to a DRAM system as the main memory, to an EPROM system and to various I/O devices and controllers. Address latches decouple the address bus from the data bus. Address decoders select among the various external modules. The output clock from the R3051 (Sysclk) is usually buffered to reduce the loading effect and to provide clock drive capability with minimum clock skew for the system.

The main DRAM memory system is based on 1 to 4 banks of non-interleaved DRAMs with 80ns of access time (t_{rac} = 80ns). The DRAMs used are 256k x 4 to provide a maximum memory space of 4MB. The DRAM memory space occupies the lower 4MB of the physical memory space. Figure 4 illustrates the architecture of the main DRAM memory system. The DRAM memory space resides between addresses 0000_0000 and 3FFF_FFFF. Address bits A(21:20) select among the four banks while the Rd and Wr outputs from the R3051 differentiate between read and write accesses.

Each memory bank (32-bit array) of DRAM, which corresponds to 1MB when using 256k x 4 DRAMs, is individually controlled by a separate RAS signal. RAS0 controls DRAM bank 0, RAS1 controls DRAM bank 1,... Each bank of DRAM is also controlled by an individual WriteEnable signal. WriteEnable0 controls DRAM bank 0, WriteEnable1 controls DRAM bank 1,... This architecture enables only a single DRAM bank for any DRAM read or a write access. The DRAM banks are arranged so that each bank represents a single, contiguous range of 1MB.

In an R3051 system, it is possible to perform a 32-bit read even when smaller data elements are requested. However on writes, it is important to enable only those bytes which are actually being written by the CPU. The R3051 bus interface provides four individual byte-enable signals to indicate which byte lanes are involved in a particular transfer. The DRAM subsystem encodes the byte-enable information from the R3051 into the CAS control signals of the DRAMs. In this encoding, CAS0 corresponds to byte lane 0, CAS1 corresponds to byte lane 1, etc. Each CAS signal is connected to the DRAM devices that correspond to the byte lane under its control in all four banks of the DRAM subsystem. That is to say that CAS0 is connected to the two DRAM devices that compose byte 0 in every DRAM bank.

Data buffers isolate the DRAM banks from the R3051 data bus to reduce the loading effect and to prevent contentions between the R3051 and the DRAMs. Note that this also alleviates concerns about the relatively slow tri-state times associated with DRAM devices. The data buffers selected are industry standard bidirectional transceivers (74FCT245). These data buffers actually isolate the data bus of the R3051 from all the external modules.

DRAM addresses are provided by multiplexing the latched R3051 address bus using the IDT FBT2827B memory drivers.

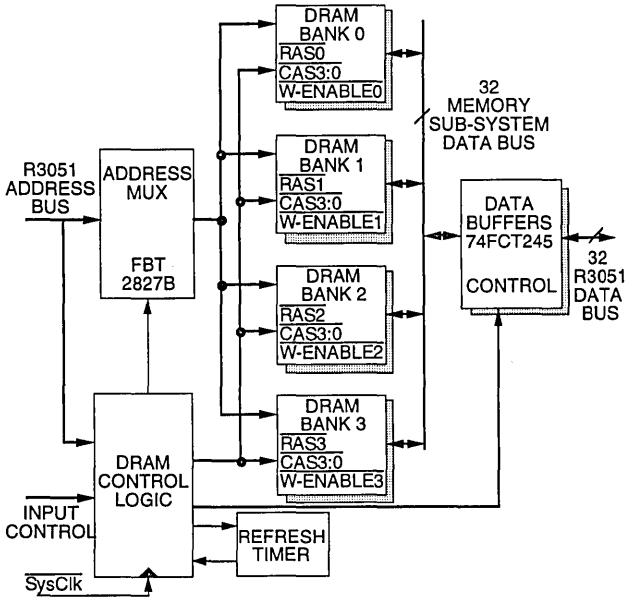


Figure 4. DRAM Memory Subsystem Architecture

This device type was selected based on its ability to drive large capacitive loading, such as found when driving 32 DRAM devices. A single FBT output has a series resistance incorporated in the output driver and is capable of driving all four banks of the DRAM subsystem. To minimize the signal skew among the DRAM devices, the address lines and the control lines to the DRAMs must use the "star" or the "fork" topology on the PCB board. In this method, all the loads on a given signal are lumped at the far end of the PCB trace. Series termination is also well suited to drive lumped (or forked) CMOS loads (like DRAMs) at the end of a PCB trace. The series termination minimizes overshoots and undershoots at the receiving end and does not add any power dissipation to the system.

Every DRAM cell consists of a MOS cell and a capacitor which encodes logic 1 and 0 in its charge. The capacitors in the DRAM cells tend to loose their charges with time through leakage. This is why DRAMs require to be refreshed at a regular time interval. The refresh mechanism is internal to the DRAMs where bits (cells) are rewritten with the same value to keep the capacitors charged. This refresh mechanism is enabled by the input control signals to the DRAM devices through the RAS and the CAS signals. In this design a refresh timer requests the refreshing of the DRAMs every 9.6µs. This refresh timer can be driven by the Sysclk from the R3051 or from an independent oscillator. The 9.6µs refresh interval chosen is more frequent than is actually required by the DRAMs. The use of this value simplified the control logic associated with page mode write. DRAMs require that RAS be maintained low no longer than 10µs; by choosing a refresh value smaller than this maximum time, the system is assured that maximum RAS low time will not be violated.

DRAM STATE MACHINE DESIGN

For the system described in this paper, a simple state machine performs the major aspects of DRAM control. The state machine uses a simple four-bit counter (C(3:0)) to dictate the timing for the DRAM control and CPU response, and is sequenced using SysClk. There are nine major states to the state machine as is illustrated in Figure 5. These states are dictated by the type of transfer requested and the state the DRAM control logic was left in by the prior transfer.

The DRAM control logic uses the Reset pulse to reset its internal states and to synchronize its operation to the R3051. During the RESET state, it also performs one refresh cycle before entering the IDLE state. In the IDLE state, the DRAM control logic arbitrates between a refresh cycle and a bus access. A DRAM bus access is started whenever the DRAM-Chip-Select and the Rd or the Wr signals are asserted. A refresh request is detected using the REF_REQ (Refresh_Request) pulse from the refresh timer. The DRAM controller supports 4 types of CPU bus accesses: "quad-word read", "Single-word read", "Single-word write" and "Page-word write". After a "Single-word write" or a "Page-word write" access, the DRAM control logic enters the IDLE RAS ASSERTED state which is an IDLE state with the RAS signals kept asserted. The RAS signals need to be precharged upon exiting this state.

Reset Cycle

A reset cycle is initiated by the assertion of the Reset signal. This is a hardware reset which initializes the control logic to the correct IDLE state. After the Reset signal is de-asserted, one DRAM refresh cycle is initiated. Most DRAMs require at least 8 refresh cycles for proper initialization. This DRAM

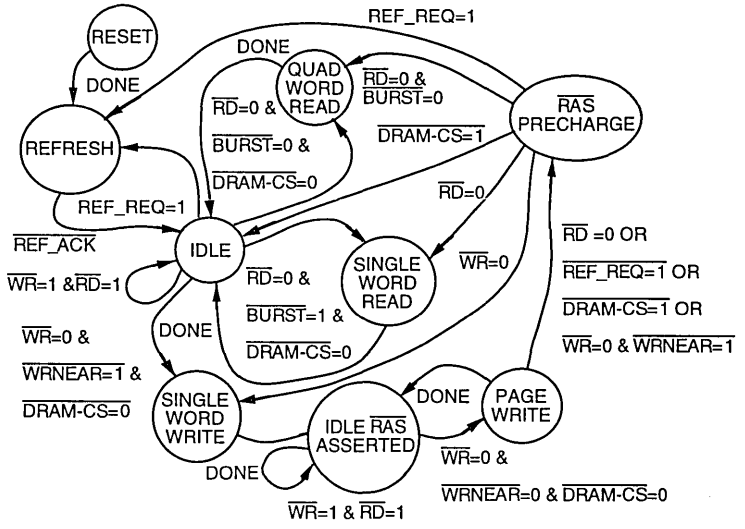


Figure 5. DRAM Control State Machine

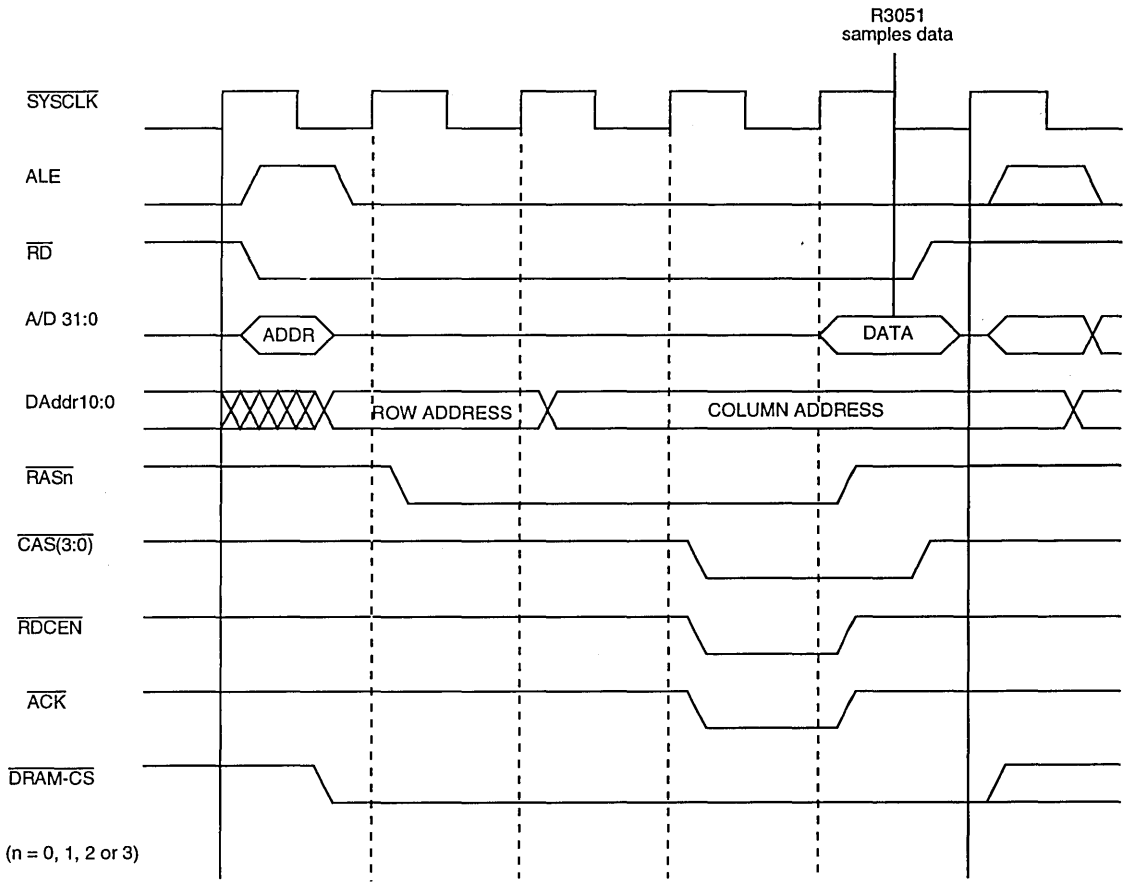


Figure 6. Single-Word Read Access Timing

control logic provides only one refresh cycle at reset time. It is the responsibility of the software to ensure that no DRAM access is made prior to the elapsing of 8 refresh periods. This can be insured by normal operation of the boot PROM; however, software could "spin-lock" for a predetermined number of loops to insure that sufficient time has elapsed.

Refresh Cycle

A refresh cycle is initiated every time a REF_REQ pulse from the refresh timer is detected. The refresh timer issues a REF_REQ pulse every 9.6µs. The DRAM control logic responds with a refresh acknowledge (REF-ACK) signal which locks the refresh timer until the refresh is serviced. The refresh interval has been set to 9.6µs which is shorter than the maximum 15.5µs refresh period that most DRAM require. The 9.6µs refresh period ensures that for an IDLE RAS ASSERTED state, where the RAS signals can be left asserted for long time periods, the maximum RAS pulse width of 10µs is not violated.

In the DRAM control logic, a refresh request has the highest priority over any other CPU requests. However, if a CPU bus requested is being serviced at the time the refresh is requested, the refresh cycle will be delayed until the end of the current bus cycle. The inverse is also true when bus requested are being delayed until the end of a refresh cycle. In this design, only the RAS-before-CAS refresh method is implemented.

Idle State

The Idle state is when the state machine is not performing any bus access or a refresh access but is constantly monitoring the bus for any access request. All the signals are deasserted and the operation of the 4-bit counter is halted.

Single-Word Read Cycle

There are two types of read transactions from the R3051: quad-word reads and single-word reads. A single-word read access is initiated by the R3051 by asserting the Rd signal.

The DRAM control logic responds by providing the R3051 with a single data element (32-bit word). Both the Ack and the RdCEn signals are used to terminate the single-word read access. In the system described in this paper, the Ack and the RdCEn signals are returned to the R3051 after 4 clock cycles, as illustrated in Figure 6.

Quad-Word Read Cycle

Quad-word reads from the R3051 occur only in response to internal cache misses. All instruction cache misses are processed as quad-word reads while data cache misses may be processed as either quad-word reads or single-word reads. The R3051 indicates quad-word read accesses by asserting both the Rd and the Burst signals. In the quad-word read access, address lines Addr(3:2) from the R3051 act as a two-bit counter to provide the address of 4 consecutive words, always starting on a word boundary.

The DRAM control logic handles quad-word read accesses using the Throttled Block Refill mode of the R3051. In a throttled read, RdCEn controls the data rate of the memory back to the CPU (latches the data into the on-chip read buffer). The Ack input is not provided back to the processor until the read transfer has sufficiently progressed such that the last word of the transfer is clocked into the on-chip read buffer (using RdCEn) one clock cycle before the processor core requires it.

In this non-interleaved system, the first word read of a quad-word read access takes the same time as a single read while the 3 subsequent words are read into the on-chip read buffer at the rate of 1 word every two clock cycles. The RdCEn is asserted for every word being read to latch the data into the R3051 read buffer. The Ack is asserted between the second and the third-word read. This ensures that for 4 subsequent falling edges of Sysclk the on-chip read buffer can provide data to the R3000A core at the rate of a word every clock cycle. Figure 7 illustrates the timing involved in quad-word read accesses.

Quad-word read accesses use the page-mode characteristics of the DRAM to obtain subsequent data word at a higher data rate. In this access, the RAS signal is kept asserted while the CAS signals are toggled 4 times to produce 4 data words.

Single-Word Write cycle

Unlike instruction fetches and data loads, which are usually satisfied by the on-chip caches, all write activity to the caches is seen at the bus interface of the R3051 as single write transactions. The R3051 indicates a single-word write access by asserting the Wr signal. The DRAM control logic enables the writing of the CPU word or partial word into the DRAMs and returns the Ack signal to terminate the write access. The Ack signal is returned to the R3051 after 3 clock cycles, as illustrated in Figure 8.

The DRAM memory system takes advantage of the WrNear signal from the R3051 by defaulting to the case that any single write to the DRAM subsystem will be followed by another write with the same upper 22 address bits. Based on this information the RAS signal must be kept asserted after every write access to enter the page mode of the DRAMs. The end of a single-word access is then different from a single read access in that the RAS signal is kept asserted.

Idle RAS Asserted State

At the end of a write access the DRAM control logic enters this idle state where a RAS signal is kept asserted while the

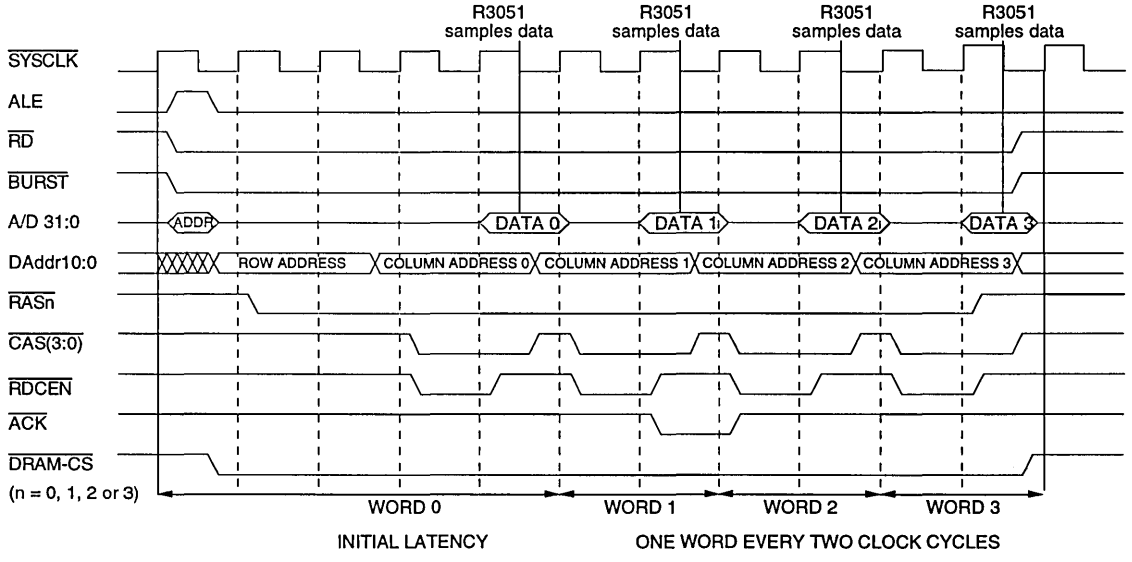


Figure 7. Quad-Word Read Access Timing

state machine awaits a subsequent transaction. If the next access is a local write (WrNear from the R3051 is asserted) the DRAM control logic enters the page write mode. If a different access type occurs, the state machine exits this state.

Page Write Cycle

A page write cycle is a single write access from the R3051 following a previous single write access with the same upper 22 address bits. The R3051 indicates a page write access by asserting the Wr and the WrNear signals.

The timing for a page write access is very similar to a single-write access but shorter since the RAS signal has been kept asserted from the previous write cycle. The Ack is returned back to the R3051 after 2 clock cycles. Figure 8 illustrates the timing for a page write access.

Precharge RAS

Any access, except a page write access, following an Idle RAS Asserted state needs to have the RAS signal precharged (driven to a level HIGH) before the access is responded to.

PERFORMANCE

The performance of the different types of R3051 bus accesses to the DRAM memory subsystem is usually measured by the number of clock cycles it takes to send the Ack back to the R3051. This time is computed from the beginning of the external access. The performance of the DRAM system can be summarized as follows:

- single read: 4 clock cycles
- block refill: 7 clock cycles
- first write: 3 clock cycles
- page write: 2 clock cycles.

This is a relatively high performance for a low-cost and easy-to-implement DRAM memory subsystem. The performance of the system can be improved by using more elaborate DRAM memory controller and/or more complex memory architectures such as address interleaving. Such systems should be able to achieve optimum performance.

FIELD UPGRADEABILITY

Many of today's systems are designed to allow for future fields upgrades of the base memory system to more memory banks and/or deeper DRAM devices. The ability to offer a base configuration (at a lower selling price) with upgrade capabilities is often a selling feature of the end product.

The system software should then run diagnostics at boot time to determine the maximum size of the available memory. Typical strategies for such diagnostics include writing distinct values into a given location within each bank, and then reading the data back to see if any of the writes did not occur properly

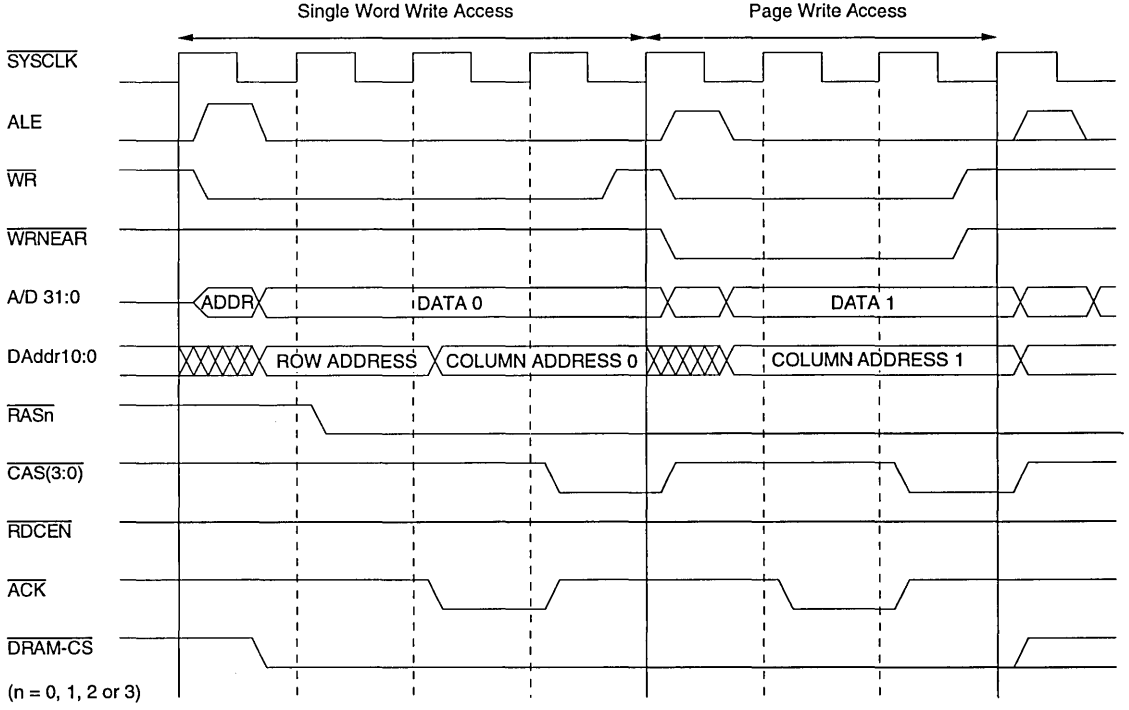


Figure 8. Single-Word Write Access Timing and Page Write Access Timing

or altered data previously written. Non-interleaved or interleaved memory architectures should be transparent to the system software.

The system hardware should make provision for extra memory banks or deeper memory devices by routing all the necessary signals to unused pins or sockets of future upgrade memory. The system hardware should try to minimize the use of jumpers to make the system much more user friendly.

In the system described in this paper, the user can upgrade to deeper memory by replacing the 256k x 4 DRAMs with deeper 1MB x 4 DRAMs to obtain a maximum memory space of 16MB. It is also possible to replace the R3051 with the R3081 to increase the performance of the system since they both have the same footprint. The R3081 with its on-chip FPA will have a great impact on the performance of floating-point intensive applications; a further benefit is the larger on-chip caches of the R3081.

CONCLUSION

The R3051 and the R3081 RISController families bus interface was designed to allow memory systems of differing complexity and performance to be implemented. Even a relatively simple DRAM system, as the one described here, offers very high performance. With simple modifications, this approach is applicable to higher frequencies (33 and 40MHz) and to interleaved memory systems yielding even higher performance. The R3081 can also be used for existing R3051 designs to improve the floating-point performance and the overall system throughput with no modifications of the external hardware.

REFERENCES

- AN-50: "Series Termination" Application Note, by Suren Kodical, 1990/91 IDT Logic Data Book.

Integrated Device Technology, Inc.

DESIGNING READ AND WRITE BUFFERS FOR THE R4000™ SYSTEM INTERFACE

APPLICATION NOTE AN-114

By Andrew Ng

INTRODUCTION

This article describes the basic concepts behind designing with the IDT79R4000 System Interface. The System Interface connects the R4000 CPU to external memory and peripherals. Topics include: (1) what the basic read and write memory transactions look like, (2) the basic architecture for designing buffers and transceivers into the address and data bus paths, and (3) explains the convention of using single level read buffers and multi-level write buffers. The read and write buffers can obviously be implemented with custom FPGAs or ASICs. However, read and write buffers can also be easily implemented using off-the-shelf discrete logic FIFOs and pipelined registers. Thus to more clearly illustrate a read and write buffer implementation, brief discrete logic examples are given using the 18-bit IDT Double-Density FCT16823T register with clock enable, the 16-bit IDT 73200 multi-level pipeline register, and the 8-bit IDT73210 2-level/1-level pipelined registered transceiver.

THE R4000 MICROPROCESSOR

The IDT79R4000 MIPS CPU brings high performance 64/32-bit computing to a single chip microprocessor and thus extends the family of R3000™ compatible parts from the lower cost 32-bit R3051™ CPU and R3081™ CPU/FPA. Benchmarks for R4000 systems show their performance to be from 35-54VUPS (VAX Units of Performance) and from 44-72 SPECmarks. Initial R4000 parts are being produced to run with an external 50MHz clock frequency and future parts with the same external bus interface are planned with larger primary caches and for frequencies over 75MHz. As shown in the block diagram in Figure 1, the R4000 has high performance in large part because of its superpipelined architecture which allows a 100MHz internal clock speed which is double the external clock speed. The R4000 also has an on-chip floating-point accelerator, on-chip write-back primary instruction and data caches, an optional writeback secondary cache interface, and on-chip memory management. The Reduced

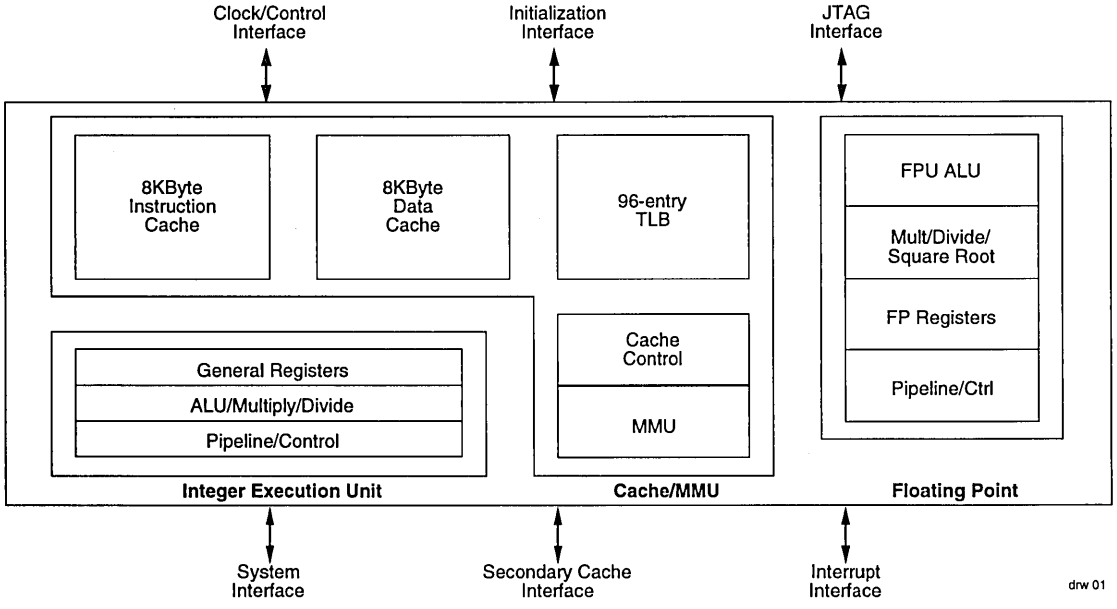


Figure 1. Block Diagram of the R4000

Instruction Set Architecture (RISC) and its development environment of optimized operating systems, compilers, and rescheduling assemblers place their emphasis on high performance and speed. The R4000 has 3 variants: (1) the 179-pin R4000PC which comes without a secondary cache interface, (2) the 447-pin R4000SC which comes with a secondary cache interface, and (3) the 447-pin R4000MC which comes with a secondary cache interface and also supports multiprocessing coherency.

R4000 Clock Interface

One outstanding characteristic of the R4000 bus, in contrast to most microprocessors, is that it uses fully synchronous timing. Thus, every output is generated relative to a clock edge, and has the same propagation delay relative to the clock. Also, every input has the same setup and hold time relative to the clock.

This allows the simplification of worst case timing analysis, so that hardware designers can concentrate on functional issues. In conjunction with the fully synchronous timing, the R4000 has a PLL, which allows it to match the input clock, MasterIn to the master (MasterOut), processor (PClock), system(SClock), and transmit clock (TClock). MasterOut is an output clock which the PLL matches up to MasterIn. PClock is an internal clock which runs at twice the frequency of the MasterIn clock. SClock is also an internal clock which is essentially equivalent to TClock and runs at the same frequency as the MasterIn clock. The PLL also allows the alteration of the slew rate of the outputs relative to the clock and provides an extra receive clock that leads the system clock by 25%, called RClock as can be seen in Figure 2. The SyncIn and SyncOut pins shown in the Clock/Control Interface of Figure 3 automatically compensate the clocks for external buffer delays. Finally, options exist which allow the system, transmit, and receive clocks to be slowed down relative to the processor clock, such that the bus interface can run at 1/2, 1/3, or 1/4 of the normal speed. These options provide flexibility in producing setup, hold, and access times appropriate for various interfaces.

R4000 SYSTEM INTERFACE

As shown in Figure 3, the R4000 System Interface consists of the signals that connect the CPU to the outside world of peripherals and memory. The System Interface has three major elements:

1. The 64-bit SysAD bus which carries the address and data.
2. The 9-bit SysCmd bus which encodes the type of memory cycle.
3. The control lines to condition the SysCmd bus and control the issue rates of the commands.

This article will discuss each of the System Interface elements in detail.

R4000 SysAD Bus

The SysAD(63:0) Bus is 64-bits wide and has 8 additional optional ECC/parity bits called SysADC(7:0). The multiplexed SysAD bus is shared between address and data phases. The addresses will be present during the clock cycles where a valid interface command is present on the SysCmd bus. Data will be present for the clock cycles where a valid data identifier is present on the SysCmd bus. During the address phase, only the least significant 36-bits, SysAD(35:0) are used for a 64 GB physical address space. By convention, the upper 28 physical address bits, SysAD(63:36) are driven to 0 with appropriate ECC/parity by the CPU.

R4000 SysCmd Bus

The SysCmd(8:0) bus is 9-bits wide and has 1 additional optional even parity bit called SysCmdP. The command bus encodes the type of transaction that is present on the system interface. For instance, block reads, block writes, single word reads, single byte writes, etc. are identified by the SysCmd encoding. The MSB (Most Significant Bit), SysCmd(8), indicates whether the cycle is a system interface command or data identifier. Thus SysCmd(8) breaks the encodings into two main cases, as listed in Tables 1, 2, and 3. Only the more common encodings are listed here, although a complete list is available in the User's Manual. Finally, some examples of the more typical 9-bit commands and data identifiers are given in Table 4.

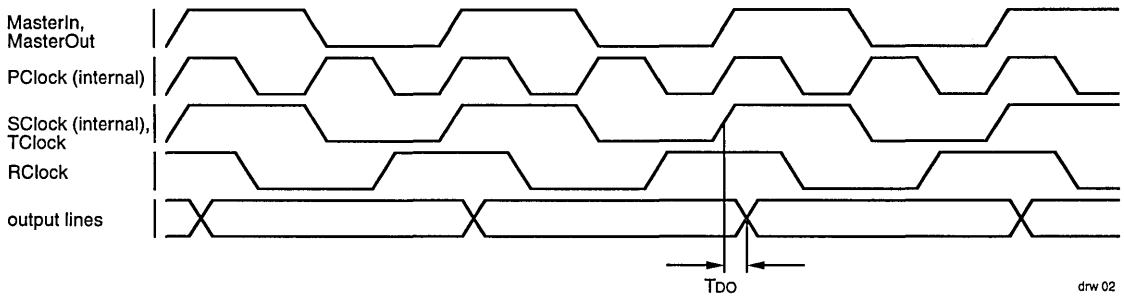


Figure 2. R4000 Clock Interface Timing (PClock to SClock divisor of 2)

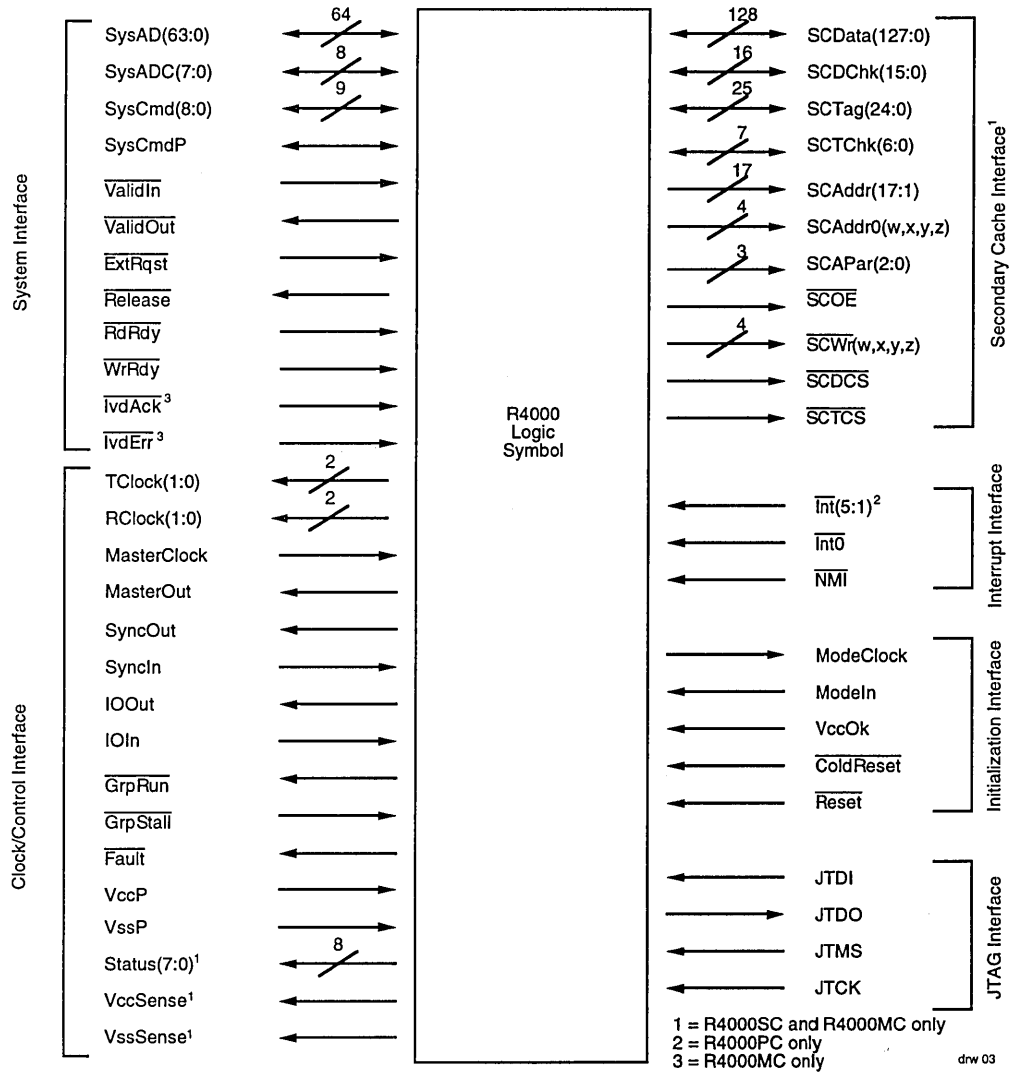


Figure 3. The R4000 Interfaces

R4000 System Interface Control Signals

The System Interface Control Signals communicate when the System Interface busses are valid, and if the external agent, (i.e., the memory), is ready to accept the command. Their descriptions are given in Table 5. Two signals, the output ValidOut and the input ValidIn are used by the CPU and the memory to indicate when they are driving valid signals onto the SysCmd and SysAD busses. For example, when the CPU is driving a valid command/address or write data on the SysCmd bus, it will assert ValidOut, and when the memory system is returning a data identifier on the SysCmd bus and read data on the SysAD bus, it will assert ValidIn. Two input signals, RdRdy and WrRdy, are used by the memory system to communicate whether or not it is ready to handle the next read and write. The output signal Release is used by the CPU or bus master to indicate to the memory system that the master is tri-stating the bus on the next clock. After Release asserts, the memory system can drive the SysAD read data and SysCmd data identifier back to the CPU. The input signal ExtRqst is used by a DMA controller or interrupt controller to

gain control of the bus from the CPU. Finally, the inputs $\overline{\text{InvAck}}$ and $\overline{\text{InvErr}}$ are used only on the R4000MC version to help manage cache coherency.

To illustrate the use of the System Interface, the following sections will give an example for a read memory cycle and a write memory cycle. The sections follow the custom used in R3000/R4000 terminology, to use the term "buffer" in the software sense, meaning, a register location to store data rather than the hardware interpretation of amplifying or isolating a signal without storing it. In the following sections, hardware buffers such as the 8-bit IDT74FCT244T will always be referred to as a "hardware buffer".

Encoding of SysCmd(8) Command or Data Identifier	
0	System Interface Command
1	System Interface Data

Table 1. SysCmd Encoding for SysCmd(8)

Encoding of SysCmd(7:5) Command	
0	Read Request
1	Read Request, Write Request Forthcoming (on the MC/SC only)
2	Write Request
Encoding of SysCmd(4:3) for Read and Write Requests attributes	
2	Noncoherent block read or write.
3	Double word, single word, or partial word read or write.
Encoding of SysCmd(1:0) for Noncoherent Block Read Requests or for Block Write Requests Block size	
0	Four words.
1	Eight words.
2	Sixteen words.
3	Thirty-two words.
Encoding of SysCmd(2:0) for Double Word, Word, or Partial Word Read Requests or Write Requests data size	
0	One byte valid (Byte).
1	Two bytes valid (Halfword).
2	Three bytes valid (Tri-byte).
3	Four bytes valid (Word).
4	Five bytes valid (Quinti-byte).
5	Six bytes valid (Sexti-byte).
6	Seven bytes valid (Septi-byte).
7	Eight bytes valid (Double Word).

Table 2. SysCmd Encodings for System Commands

SysCmd(7)	Last data element indication
0	Last data element
1	Not the last data element.
SysCmd(6) Response Data indication	
0	Data is response data, e.g., read data
1	Data is not response data, e.g., write data
SysCmd(5) Good data indication	
0	Data is error free.
1	Data is erroneous, e.g., a bus error
SysCmd(4) Data checking enable (on external agent data only)	
0	Check the data and check bits.
1	Don't check the data and check bits.
SysCmd(3) Reserved	
SysCmd(2:0) Cache state (on R4000MC only).	
0	Invalid
4	Clean Exclusive.
5	Dirty Exclusive.
6	Shared.
7	Dirty Shared.

Table 3. SysCmd Encodings for Data Identifiers

SysCmd(8:0) 876543210	Description of Command
000010001	Read request, Noncoherent block, eight words
000011011	Read request, Double word or smaller, four bytes valid
001010001	Write request, Block, eight words
001011011	Write request, Double word or smaller, four bytes valid
110000100	Read response data not end of block
100000100	Read response last data
100100100	Read response last data ignore ECC/parity
111000101	Write data not end of block
101000101	Write data, last data

Table 4. Examples of Typical SysCmd Commands and Data Identifiers

READ INTERFACE TRANSACTIONS

In Figure 3, the read interface state machine looks for ValidOut to assert along with one of the read commands as encoded by SysCmd(8:5). The SysCmd bus in the example is binary 000010001, which is an eight-word block read. Transactions involving a single double-word read are similar. By convention, the block size will either be the primary instruction cache or the primary data cache line size, or if present, the secondary cache line size. The SysAD bus contains the address for the transaction on the same clock as the read request command. The state machine should latch or register the address since the SysAD bus is multiplexed. Thus, each read transaction will only issue one start address even if it is a block read. If the state machine is not ready to handle the command, it should keep RdRdy de-asserted. RdRdy will delay the beginning of the read transaction by keeping the address on the bus. A caveat on RdRdy is that because it is synchronized to a clock edge, the CPU will not respond to it until 2 clock cycles later as shown in the example in Figure 4. The CPU asserts Release to indicate that the CPU is ready to tri-state the SysAD and SysCmd bus on the next clock cycle. The R4000 protocol allows Release to be either a variable number of clocks after ValidOut or possibly concurrent with ValidOut. Thus, the memory system must dedicate an extra state to allow for variable timed Releases. Along with Release, the memory system must also wait for any writes that are in progress to finish, since writes in R4000 systems are FIFOed (use First-In-First-Out buffering). After sampling Release and checking for on-going writes, the memory system can drive the bus and return data. In addition to the data, the memory system must drive a data identifier on the SysCmd bus and drive ValidIn to tell the CPU what it is returning. The memory system has direct control over the data return rate when it issues data identifiers. Some of the memory system return commands include data, end-of-data, and bus error (from Table 3, binary 110000100, 100000100, and 100100100, respectively).

Pin Name	Type	Description
ValidOut	Output	Valid Output Signals that the processor is now driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
ValidIn	Input	Valid Input Signals that an external agent is now driving a valid address or data on the SysAD bus and a valid command or data identifier on the SysCmd bus.
RdRdy	Input	Read Ready Signals that an external agent can now accept a processor read, invalidate, or update request in both non-overlap (non-secondary cache) and overlap (secondary cache) mode or can accept a read followed by a potential invalidate or update request in MC secondary cache overlap mode.
WrRdy	Input	Write Ready Signals that an external agent can now accept a processor write request in both non-overlap (non-secondary cache) and secondary cache overlap mode.
Release	Output	Release interface Signals that the processor is releasing the system interface to slave state.
ExtRqst	Input	External Request Signals that the system interface needs to submit an external request.
IvdAck, IvdErr	Inputs	Invalidate Acknowledge and Invalidate Error Signals on the R4000MC which indicate successful or unsuccessful completion of a processor invalidate or update request for cache coherency. Must be pulled high on other packages (SC).

Figure 5. R4000 System Interface Control Lines

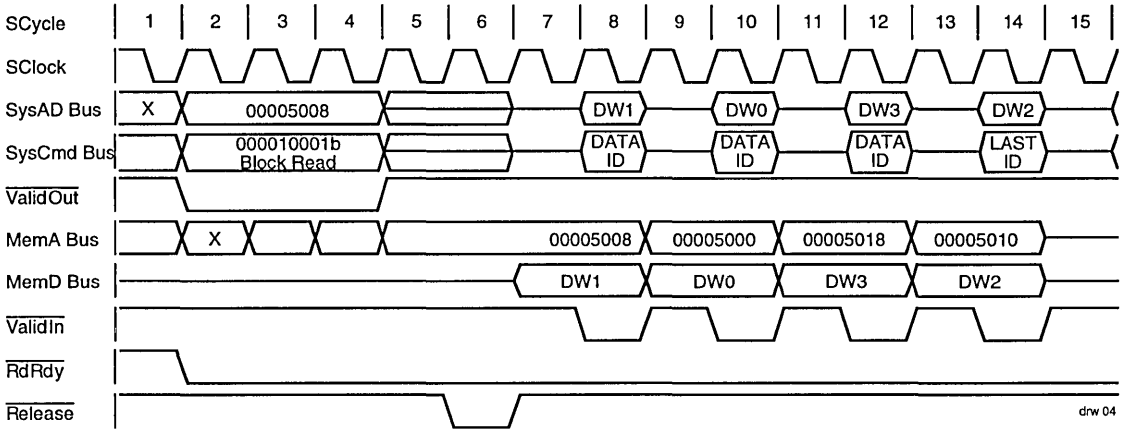


Figure 4. R4000 Block Read Cycle

On a block read, the state machine must increment the double word (8 bytes) LSB (Least Significant Bit) address bits of the block and keep returning more double words until the block is finished. These double-word LSB address bits are changed either in a sub-block order [hex (00,08,10,18,...), (08,00,18,10,...), (10,18,00,10,...), or (18,08,10,00,...)] or in a sequential wrap-around order [hex (00,08,10,18,...), (08, 10, 18, 00), (10, 18, 00, 08), or (18, 00, 08, 18)] depending on the package type and boot-strap configuration. Sub-block ordering requires the original double-word start address to be XOR-ed with the block counter. Sub-block ordering is used to simplify the internal controls, since the word that is needed within a block, (e.g., the instruction), can always be returned in the same place. Sub-block ordering is required on the R4000PC and is optional on the R4000SC/MC.

Note that bus errors on block reads still require the memory system to return an end-of-data command to signal the end of the block, thus allowing the memory system to finish the rest of the block if it desires. Also, uncached memory, and especially I/O interfaces, can ignore ECC/parity generation/checking by using SysCmd(4) to indicate to the CPU that it doesn't want ECC/parity checked.

The Sieve Search Algorithm

Because the address is generated on the same clock as the command and the $\overline{\text{ValidOut}}$ signal, the address register state machine usually has to implement a "door-to-door search algorithm". In the sieve algorithm, the address registers are enabled and constantly register new addresses on each clock. This means the registers are normally clocking in invalid addresses until the right one comes along. When $\overline{\text{ValidOut}}$ is detected, the address register should stop clocking and will hold the address until the end of the read or write. Thus, the address register is constantly searching for a valid address and incidentally latching in many of incorrect addresses until the correct one comes along.

R4000 Read Buffer Size

To implement the read buffer, enough buffer locations must be present to store the incoming memory. For the R4000PC, which puts incoming main memory data directly into the primary cache, the maximum incoming memory read rate of 2 words per clock is matched by the CPU's capability to put these words into the primary cache. If a secondary cache is

Secondary Cache Write Time	Memory Speed	Max. Buffer Levels Needed
1–2 SCycles,1–4 PCycles	D	1
3 SCycles, 5–6 PCycles	DDx	1
4 SCycles, 7–8 PCycles	DDxx	1

Table 6. Examples of the Maximum Processor Read Data Rates for the MC/SC

present, then enough time is needed to put the data/instruction into the secondary cache. For the R4000MC/SC, the secondary cache write rates may bandlimit the main memory read buffer if they are slower than main memory. However, this often is not a realistic case, since one of the purposes of the secondary cache is to provide a faster access time than main memory, in addition to isolating microprocessing systems from one another. In Table 6, the number of SCycles (assuming a PClock divide by 2 divisor for SClock) is shown along with the equivalent number of processor PCycles, since the on-chip secondary cache interface uses PCycles to time the secondary cache. The memory speed of the external system is indicated with a D, which means one double word per clock, and possibly followed by one or more x's, which indicate idle clocks. Thus a DDxx pattern indicates 2 double words can be returned every four clocks. A case in which more than one level of read buffering may be desired is shown in the next section.

Secondary Cache Overlap Mode

Some complexity is added to the state machine and the interface. The R4000MC/SC (but not the PC) uses a secondary cache overlap mode along with regular reads and writes that can issue a read command, which, in turn, issues a write command between itself and the expected data. For example, when the read command is issued, the write address and the write data are issued, which must be handled or buffered by the memory system. Only then can the memory system return the data for the read. The purpose of the secondary cache overlap mode is to allow the memory interface to better utilize the read access time, if it chooses to do so. Therefore, a DRAM memory system could begin a $\overline{RAS}$ precharge for the read while buffering the write data, as an example.

Figure 5 displays an example of a secondary cache overlapped read and write. This example uses a 4-word block size. For the secondary cache overlap mode, the state machine should latch/register the read address and then buffer the write. It must also use a signal to indicate that the write has to be delayed until the memory system is done with the read. In these cases, the read buffer needs a set of address registers separate from the write address registers. Note that since secondary cache overlapped writes are caused by writeback misses, the MSBs corresponding to the Secondary Cache address (minus the block size LSB bits) will be the same for a secondary cache overlapped read and write. Even though most address bits must have separate read and write registers, the Secondary Cache block address bits only need one set of registers.

Additional complexity exists for the multiprocessing R4000MC version, such that a potential invalidate or update might come between the read and write portions of the cluster. Therefore, R4000MC interfaces may require an additional address latch/register for the LSB portion of the potential invalidate/update double word address.

In addition, since the $\overline{Release}$ is definitely delayed by the secondary cache overlapped write data, it is possible for very fast memory systems to want to begin to return data before the CPU can possibly accept it. In these cases, a cost/performance trade-off having more than 1 level of read buffering can be made.

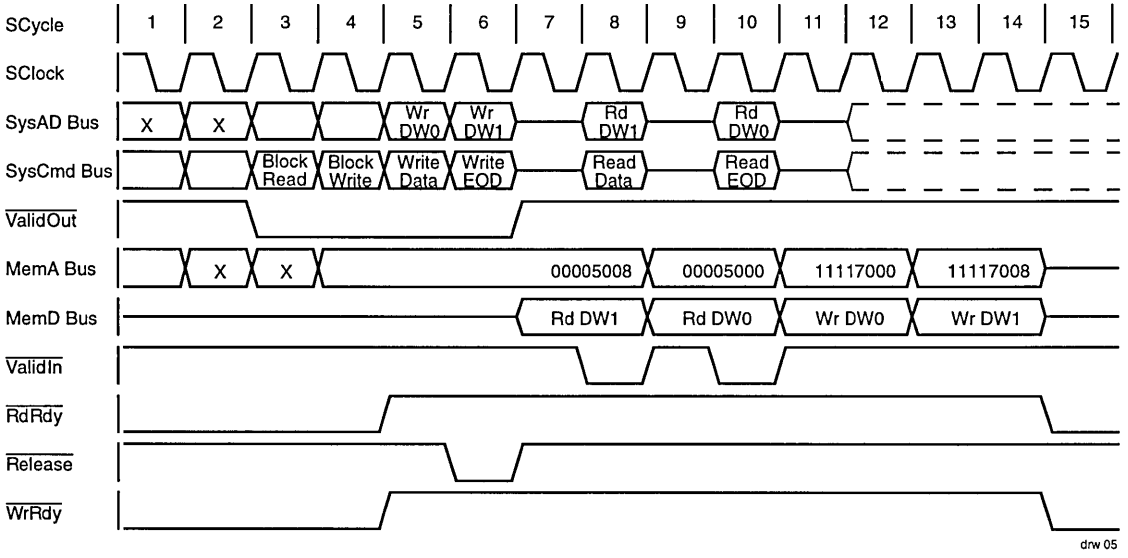


Figure 5. Secondary Cache Overlap Timing

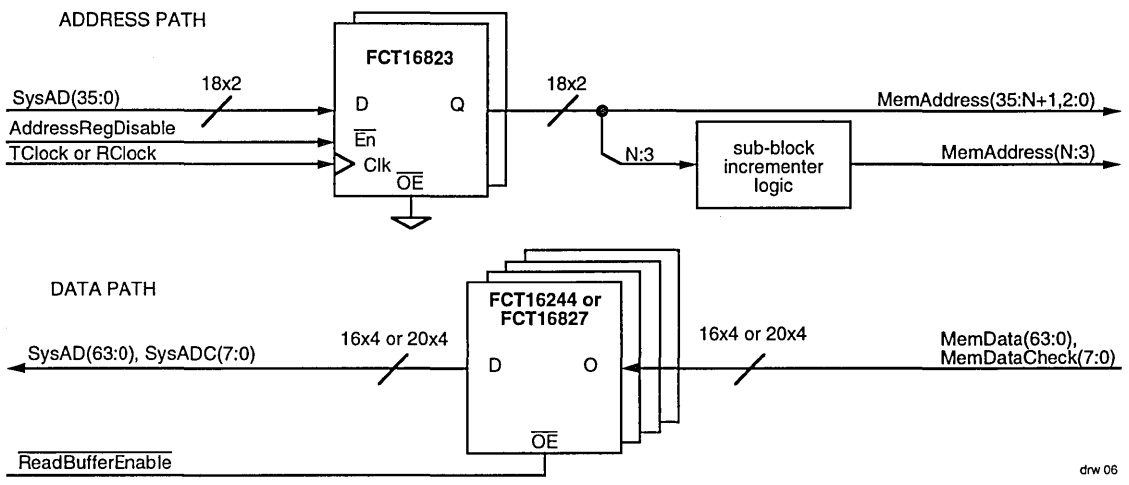


Figure 6. R4000 Single Level Read Buffer

EXAMPLE OF AN R4000 READ BUFFER

The address latch/register for an R4000 memory interface can be built from parts such as the 18-bit FCT16823T register with clock enable. The critical parameter in the latch/register portion of the read interface is the latch's data hold time for the R4000 SysAD bus as shown in Figure 2. This can be solved two ways.

In the first method, the worst case hold time for a typical latch/register such as 16-bit FCT-T logic is 1.5ns which is added to the worst case clock skew from the R4000 is 0.5ns. The 2.0ns total of worst case factors is just met by the 3.5ns minimum data propagation delay (T_{DO}) of the R4000. If additional margin is needed — for instance if external clock buffering has additional clock skew — then the following can be done: The characteristic hold time for high-speed CEMOS™ 16-bit FCT-T logic is typically 0ns or less, especially at low temperature. Also, the 3.5ns minimum data propagation timing of the high-speed CEMOS R4000 outputs, which only occurs at low temperature, can be guaranteed to be indirectly delayed upto an additional 2.5ns by changing the slew rate of the outputs. The rise and fall slew rates can be adjusted by programming the serial boot initialization register interface at reset time. By using slower slew rates, which change the rise and fall times and, therefore slightly delay the outputs of the R4000, enough data hold time can be provided to memory interface latches/registers, even when considerable clock skew is taken into consideration.

A second method for providing additional hold time, especially for interfaces made from ASICs and FPGAs, is to use the RClock, as previously shown in Figure 2. The RClock leads the TClock by 25% of the TClock and therefore, at 50MHz provides 5ns of additional hold time. The disadvantage of using the RClock is either the latches/registers must be immediately staged with a set of TClock latches/registers and/or very fast control logic for the clock enable (which typically is TClock based) must be used.

Since the memory system access time is usually equal or greater than the secondary cache access time for the MC/SC systems and the PC systems can handle data as fast as the main memory system can return it, a simple hardware buffer is all that is needed for the data path, such as the 16-bit IDT FCT16244T or 8-bit FCT244T as shown in Figures 6 and 7. Alternatively, a pipeline register with clock enable, such as the 18-bit FCT16823T, could be used for the data path.

In systems with interrupt or external invalidate controllers, if the controller is isolated from the SysAD bus and on the memory side of the system interface, then the address registers may need to be bi-directional. An example of bi-directional registered transceivers with data clock enables is the 16-bit FCT16952T and the 8-bit 74FCT52T.

R4000 WRITE INTERFACE TRANSACTIONS

A typical write sequence is shown in Figure 8. The write interface state machine looks for $\overline{\text{ValidOut}}$ to assert along with one of the write commands, as encoded by SysCmd(8:5) in Tables 1-3. The SysCmd bus in the example is binary 001010001, which is an eight-word block write. Single double-word transactions are similar. If the state machine is not ready to handle the command, it should keep $\overline{\text{WrRdy}}$ de-asserted. The caveat on using $\overline{\text{WrRdy}}$ is that because it is synchronized to a clock edge, the CPU will not respond to it until 2 clock cycles later. Thus, when $\overline{\text{WrRdy}}$ asserts, the address and $\overline{\text{ValidOut}}$ will remain on the bus for 2 more clocks. The SysAD bus contains the base address for that transaction on the same clock as the write request command. Block writes always increment the address sequentially, i.e., hex (00,08,10,18,...). The state machine should latch or register the address, since the SysAD bus is multiplexed. Each write transaction will only issue one start address, whether it is a single write or a block write, thus, external logic is needed to increment the base address for the memory system.

After the address is generated, $\overline{\text{ValidOut}}$ will be asserted

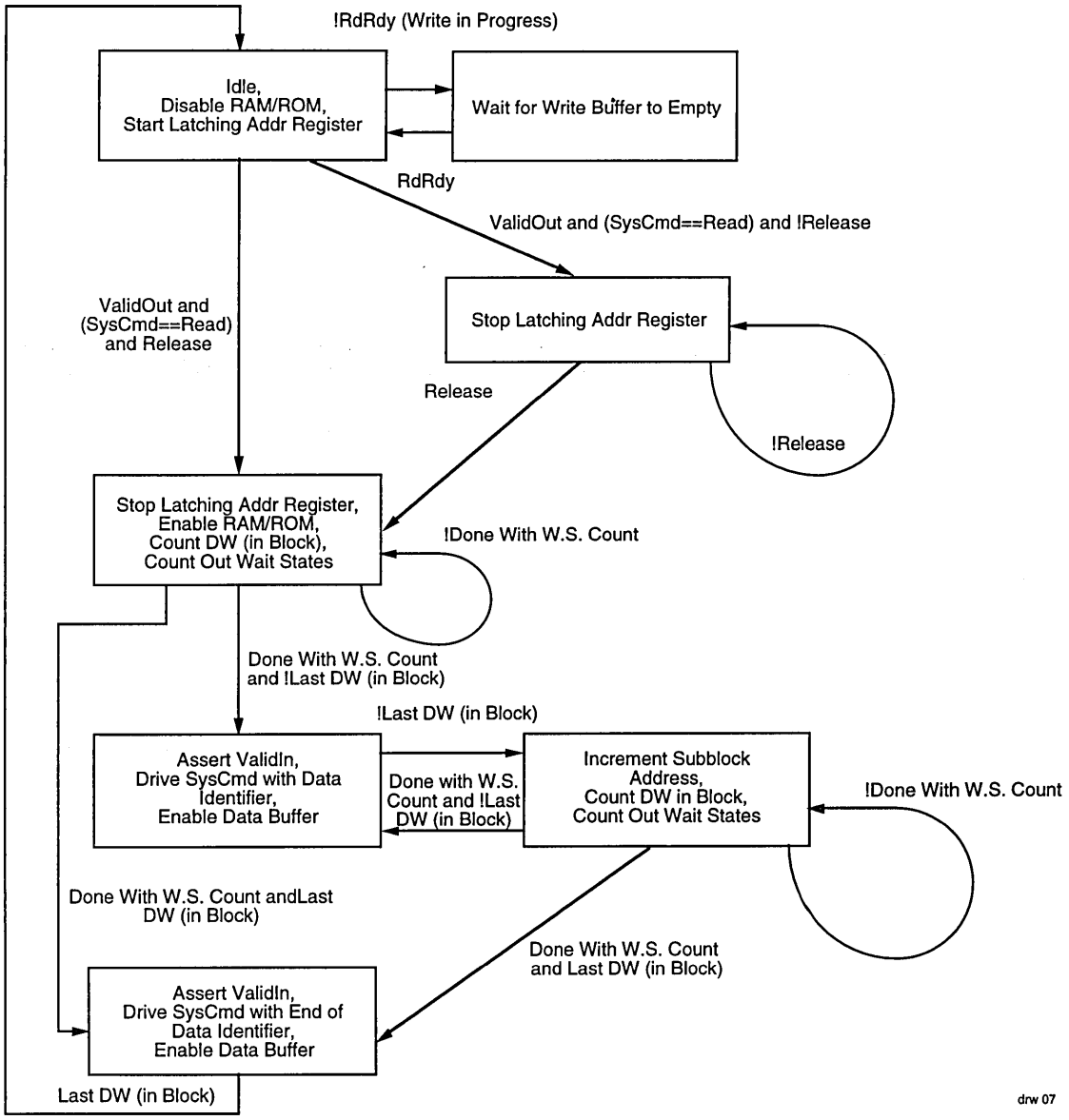


Figure 7. R4000 Read Interface State Machine

along with the first data of the write immediately, or a variable number of clocks later. The state machine must add a condition for the variable number of clocks between the address and the first data. If the data is a block write, the remaining data will be generated in a pattern selected by the initialization boot prom as shown in Table 7. In Table 7, Dxx means that a Data clock is followed by two idle clocks between each of the data items. However, no idle cycles are guaranteed after the last Data clock. Because the data rate pattern on writes is preselected at reset time using the serial boot initialization the register interface, the memory system cannot dynamically slow down any further and still control the data rate. Unless data can be written to memory at this preselected rate, the data must be buffered until the memory system can handle it.

Thus, data is written at a rate that further requires external buffering via a FIFO. The major reason for this arrangement is to allow memory writes from the buffer/FIFO to occur at the same time as cached reads. This allows the CPU to execute

cached instructions in parallel with the retiring of write data. In addition, the data caches use a writeback protocol, where data stores are always written to cache, but main memory is only updated when necessary (i.e., when another cache access needs to replace the cache location that is holding the freshly written data). Thus cached load and store fetches can also occur in parallel with the retiring of external system interface writes.

In contrast to reads, writes must indicate bus errors through an interrupt or some other external hardware mechanism. The CPU has an internal write buffer and also expects the memory system to have an additional external write buffer. Therefore, the CPU cannot match a bus error indication to a precise address and data pair, because it is decoupled from when the memory system actually tries the write. The system can choose to save address and data information with external hardware if it needs to match the error to the precise address and data within the write buffer. Uncached writes which are less than a double word wide, (e.g., 1byte), still produce data on the other bytes and the appropriate ECC/parity. However, the data for the unused bytes is pseudo-random, in that the CPU drives out what was last contained in an internal data buffer.

R4000 Write Buffer Depth

In general, to implement the write buffer, enough buffer locations are needed to store all of the double words in the block write. However, as write data is being written into the buffer at the preselected data pattern rate, it is possible that the first few double words in the block write have been retired to main memory, much like a FIFO. Thus, theoretically, those buffer locations could be reused for the last few double words of the block write, as long as the buffer does not overflow. For memory which has predictable and consistent access time for each word (Static RAM) see Table 8. Not all data rate patterns and buffer sizes are shown, but the other cases can be derived using queuing theory producer/consumer model. Similar to block reads, the maximum block size is the largest primary or secondary cache-line size. For most systems, the control portion of the write buffer is simplified if the number of buffers matches the maximum block size.

DRAM systems complicate the optimal cases due to the first word possibly taking longer than the others because of $\overline{RAS}$ precharge, $\overline{RAS}$ address hold time, or because of the delay from a $\overline{CAS}$-before-$\overline{RAS}$ Refresh. In such cases, de-asserting $\overline{WrRdy}$ until the precharge or refresh is done and then choosing a slow enough data pattern rate to handle burst DRAM column page accesses prevents having to select a very deep buffer.

Byte Enables

On the memory system logic, the 8-byte enables must be generated from the SysCmd and address for writes that are less than a double-word wide (from 1 to 7 bytes wide). Note that in contrast to most microprocessors, the R4000 will never generate an unaligned write. Thus, the 1 to 8 bytes written will always be contained within a double-word boundary. In addition, if only 1 to 4 bytes are written, they will always be contained within a word boundary. In other words, whenever 5 to 8 bytes are read or written little endian/ big endian, either the LSB/MSB must be at address offset 0 or the MSB/LSB must be at address offset 7, and whenever 1 to 4 bytes are read or written little endian/big endian, either the LSB/MSB must be at address offset 0 or the MSB/LSB must be at address offset 3.

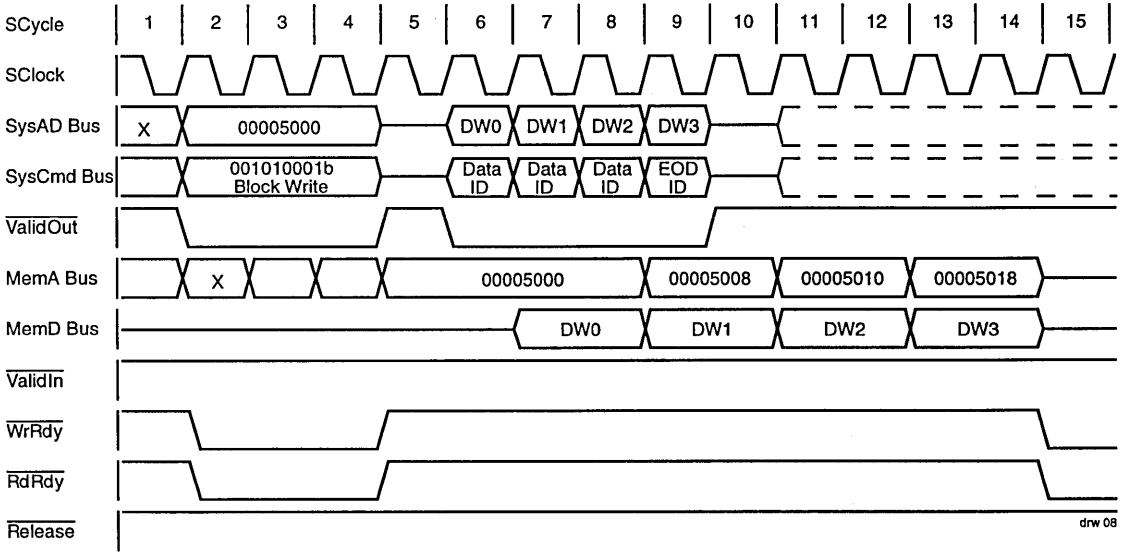


Figure 8. R4000 Write Block Cycle

Serial Init Bits 14:11	Data Rate Pattern
0	D
1	DDx
2	DDxx
3	DxDx
4	DDxxx
5	DDxxxx
6	DxxDxx
7	DDxxxxxx
8	DxxxDxxx
9–15	Reserved

Table 7. Possible Data Rate Patterns for Block Write

Cache Line Size	CPU Rate	Memory Speed	Max. Buffer Levels
4	DD	1 clock	1
		≥ 2	2
	DxD	≤ 2 clocks	1
		≥ 3	2
	DxxD	≤ 3	1
		≥ 4	2
	DxxxD	≤ 4	1
		≥ 5	2
8	DDDD	1 clock	1
		≥ 3	4
	DxDxDxD	≤ 2 clocks	1
		2	3
		≥ 3	4
	DxxDxxDxxD	≤ 3	1
		4–5	2
		6–11	3
		≥ 12	4
	DxxxDxxxDxxxD	≤ 4	1
		5–7	2
		8–15	3
		≥ 16	4
16	DDDDDDDD	Max. Case	8
32	DDD...DDD	Max. Case	16

Table 8. Maximum Write Buffer Depth Needed For Various Cache Sizes

For example, for a little endian system, a five-byte write or read, with bytes 0 through 4 enabled, could happen, but a five-byte write or read, with bytes 1 through 5 enabled, could never happen. A non-reduced PLA equation for one of the eight byte enables is shown in Table 8. The other seven byte enables are similar, and the equation can be simplified if the endianess is predetermined, or if it is known that the 64-bit mode won't be used. The re-alignment load/store–left/write instructions lwl, lwr, ldl, ldr, swl, swr, sdr, and sdw are used to develop the byte enable equations.

EXAMPLE OF AN R4000 WRITE BUFFER

The address buffer for writes is similar to the address buffer for reads and can use the 18-bit FCT16823T. On the R4000PC which does not have secondary cache overlapped commands, the read address buffer can also be used for the write address buffer. The caveat is that $\overline{\text{RdRdy}}$ needs to be asserted during the write so that any potential reads will wait until the write is done with the address buffer before continuing. On the R4000MC/SC, separate registers are needed, as previously discussed, for the read address and the write address so that read, followed by write secondary cache overlap clusters, can be handled. The write address buffer needs to use the same door-to-door search algorithm to hold the address as the read address buffer. The primary difference between the two is that after latching/registering the address, the write buffer needs to increment the addresses for block writes sequentially instead of sub-block ordering. Similar to read, a write address register looks for a write SysCmd along with $\overline{\text{ValidOut}}$ before disabling the clock enable.

The write data buffer could consist of an ASIC or FPGA, however, the write buffer can also be easily implemented using discrete logic FIFOs or pipeline registers. An example is the IDT73200 pipeline register, 16-bits wide and 8 levels deep. It can either load a specific register slot through its instruction pins or automatically ripple data through, similar to a FIFO. Either method is acceptable with the R4000, because the block size is known at the beginning of the transaction. The block size will either be the primary cache line size or, if present, the secondary cache line size. If 16 or 32 locations are needed, then the IDT73200 can be expanded by using two or four in series in the ripple-through mode. Two separate state machines are needed, one for controlling the CPU-to-buffer interface and the other to control the buffer-to-memory interface. On the CPU side-state machine, block writes require the IDT73200 to start latching/registering in new data by incrementing the write pointer so a new register is selected to be written. On the last double-word of a block, the IDT73200 needs to be told when to stop latching new data, since re-pointing to the first location could possibly destroy that data too early. This can either be controlled with a special hold command on its instruction pins, I[3:0] = hex F, or by de-asserting the $\overline{\text{ClkEn}}$ pin after latching the last double word. The memory side needs to implement a state machine which checks to see if a read is in progress from a secondary cache overlapped read. Once ready, the state machine can initiate the write to the memory and select the register to output via the select pins. The logic to select the output register can also be used to generate the sequentially ordered least-significant double-word address bits. $\overline{\text{WrRdy}}$ can be de-asserted during a write to indicate that the buffer is full and to keep any subsequent writes from occurring until the IDT73200 (or a FIFO) can accept more data. A key control issue is to de-assert $\overline{\text{RdRdy}}$ while data is being written to memory, so that subsequent reads will wait for the memory bus to become free. Because $\overline{\text{RdRdy}}$ takes two clocks to react, the de-assertion must take place during the write command.

Other options include using a 4-deep pipelined register such as the 74FCT520, or a 2-deep pipelined register such as the IDT73210. An example using the IDT73210 will be given in the next section.

EXAMPLE OF AN R4000 INTEGRATED READ AND WRITE BUFFER

Some systems, as shown in Table 7, can retire their writes at a fast enough rate to only require a 2-deep write buffer. These cases are especially prevalent when the cache line size is 4 words. In these cases, the IDT73210 can be used. The part was originally designed for embedded R3000 read and write buffering, and also works well for integrated R4000 read and write buffering. It is an 8-bit transceiver with an extra data input which can generate parity. In one direction it is registered once, while in the other direction, it is registered twice. Thus, by setting it up so that the write buffer uses the 2-deep path, and the read buffer uses the 1-deep path, the part can be used in R4000 systems. The $\overline{BEN}$ and SEL pins can be used to control register ripple-through. The most straightforward way to use the controls requires Y-register loading by the first double-word, followed by ripple-through enabling, so the first double word is put into register Z as the second double word is loaded into register Y. Thus, the second double word must come on the clock cycle immediately following the first double word. This data rate pattern can be achieved by selecting a D or DDx pattern from Table 7 with the serial boot interface reset initialization. Other methods which use features of the IDT73210 not detailed here can be implemented to handle other kinds of data patterns. However, the controls will be more complicated than the above case.

```
!BE_B/ {BYTE ENABLE FOR THE LANE FOR DATABITS 55:48}
 := ((RESET/ AND !VALIDOUT/ AND SYSCMD[8:5]==b'1X1X) AND
    !BIGEND AND (MEMADDR[2:0]==b'110) AND (SYSCMD[2:0]==b'000) OR {LIT BYTE  }
    !BIGEND AND (MEMADDR[2:0]==b'110) AND (SYSCMD[2:0]==b'001) OR {LIT 1/2 WD}
    !BIGEND AND (MEMADDR[2:0]==b'100) AND (SYSCMD[2:0]==b'010) OR {LIT 3BYTE }
    !BIGEND AND (MEMADDR[2:0]==b'101) AND (SYSCMD[2:0]==b'010) OR {LIT 3BYTE }
    !BIGEND AND (MEMADDR[2:0]==b'100) AND (SYSCMD[2:0]==b'011) OR {LIT WORD  }
    !BIGEND AND (MEMADDR[2:0]==b'011) AND (SYSCMD[2:0]==b'100) OR {LIT 5BYTE }
    !BIGEND AND (MEMADDR[2:0]==b'010) AND (SYSCMD[2:0]==b'101) OR {LIT 6BYTE }
    !BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'110) OR {LIT 7BYTE }
    !BIGEND AND (MEMADDR[2:0]==b'001) AND (SYSCMD[2:0]==b'110) OR {LIT 7BYTE }

     BIGEND AND (MEMADDR[2:0]==b'001) AND (SYSCMD[2:0]==b'000) OR {BIG BYTE  }
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'001) OR {BIG 1/2 WD}
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'010) OR {BIG 3BYTE }
     BIGEND AND (MEMADDR[2:0]==b'001) AND (SYSCMD[2:0]==b'010) OR {BIG 3BYTE }
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'011) OR {BIG WORD  }
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'100) OR {BIG 5BYTE }
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'101) OR {BIG 6BYTE }
     BIGEND AND (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'110) OR {BIG 7BYTE }
     BIGEND AND (MEMADDR[2:0]==b'001) AND (SYSCMD[2:0]==b'110) OR {BIG 7BYTE }

                 (MEMADDR[2:0]==b'000) AND (SYSCMD[2:0]==b'111) OR {DOUBLE WD }
                                           (SYSCMD[4:3]==b'1X)  OR {BLOCK     }
    (!BE_B/ AND !MEM_ACKNOWLEDGE/)
);
```

Table 9. Byte Enable PLA Equation

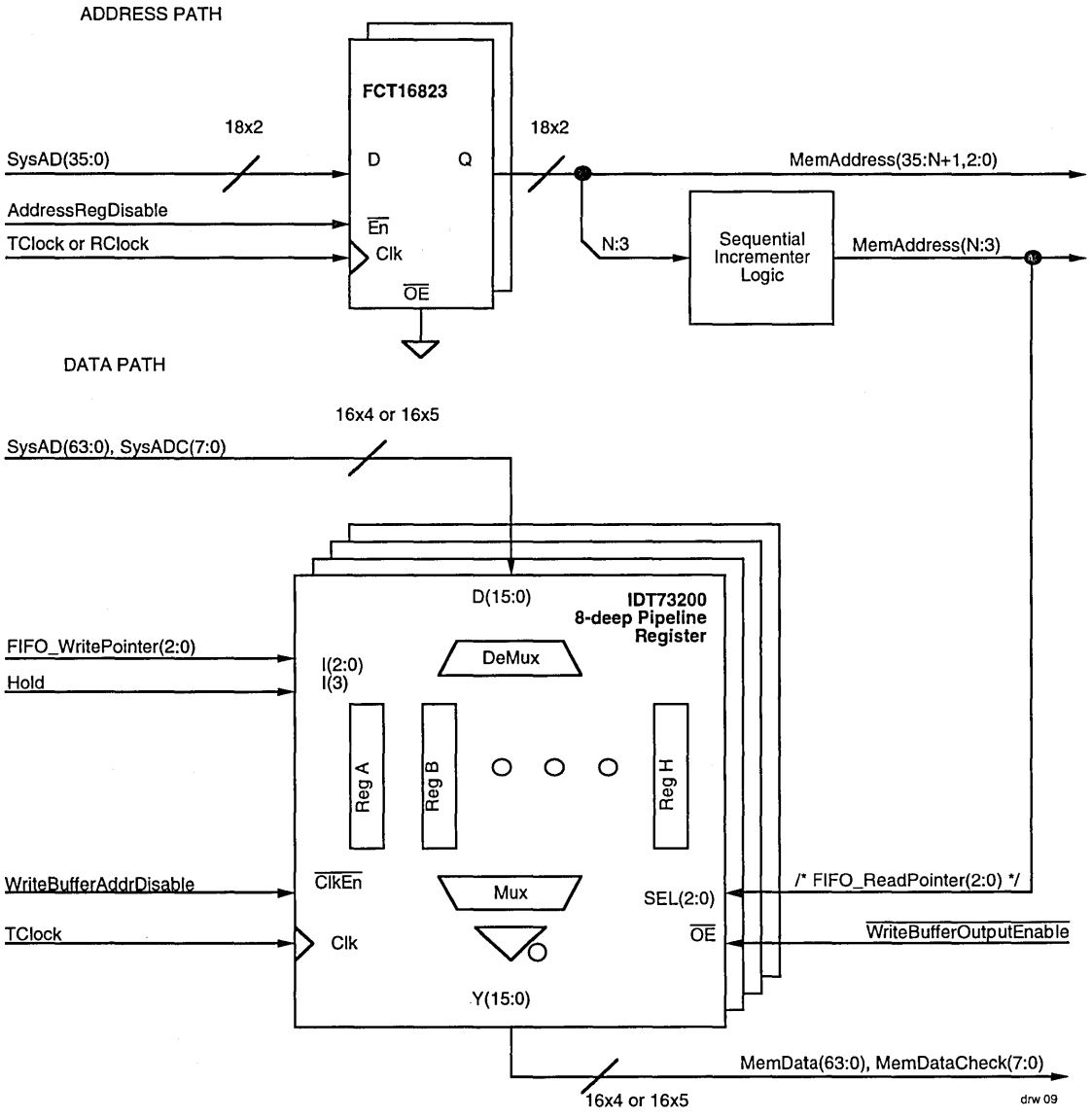


Figure 9. R4000 8-Level, 64-bit Wide Write Buffer

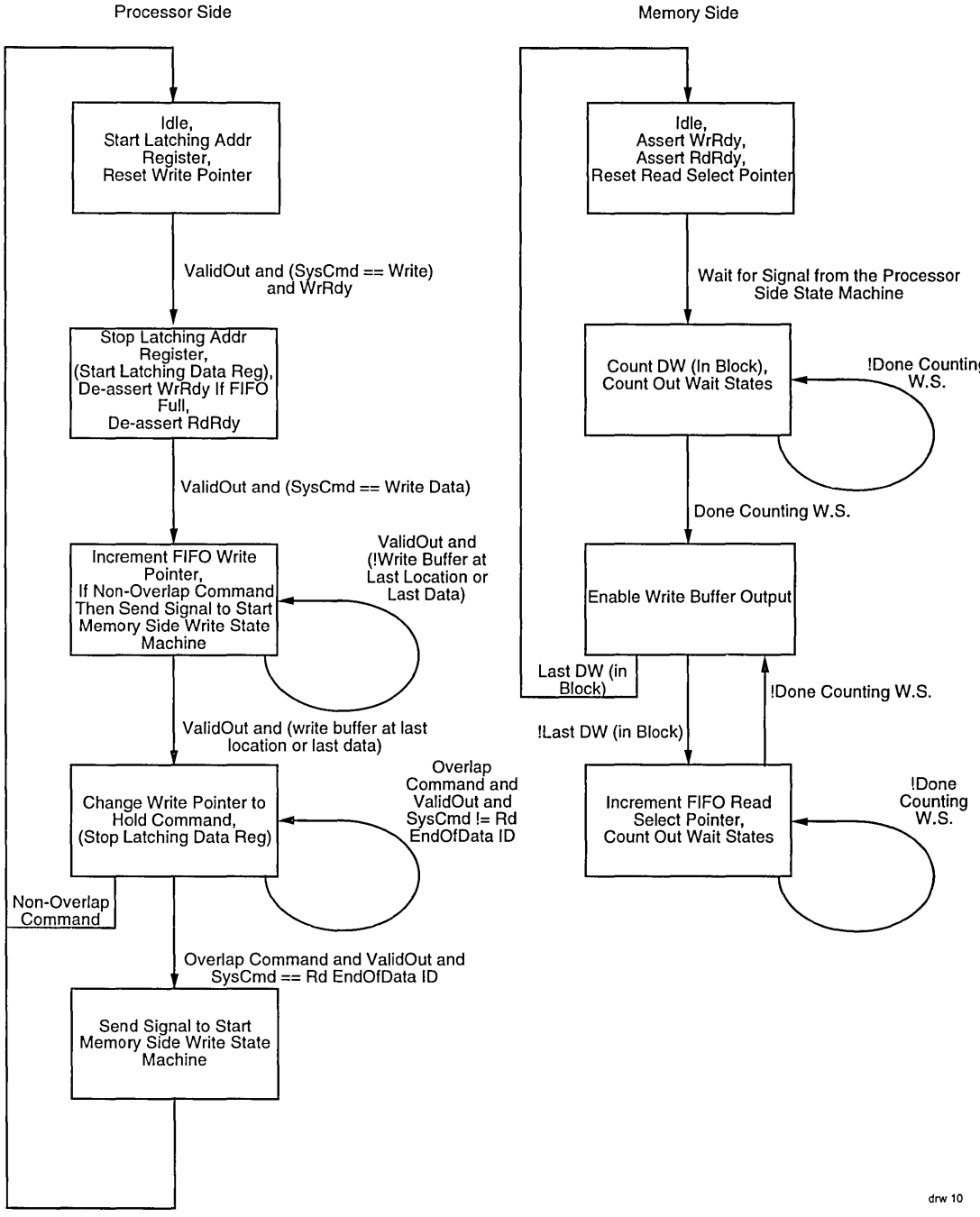


Figure 10. R4000 Write Buffer State Machine

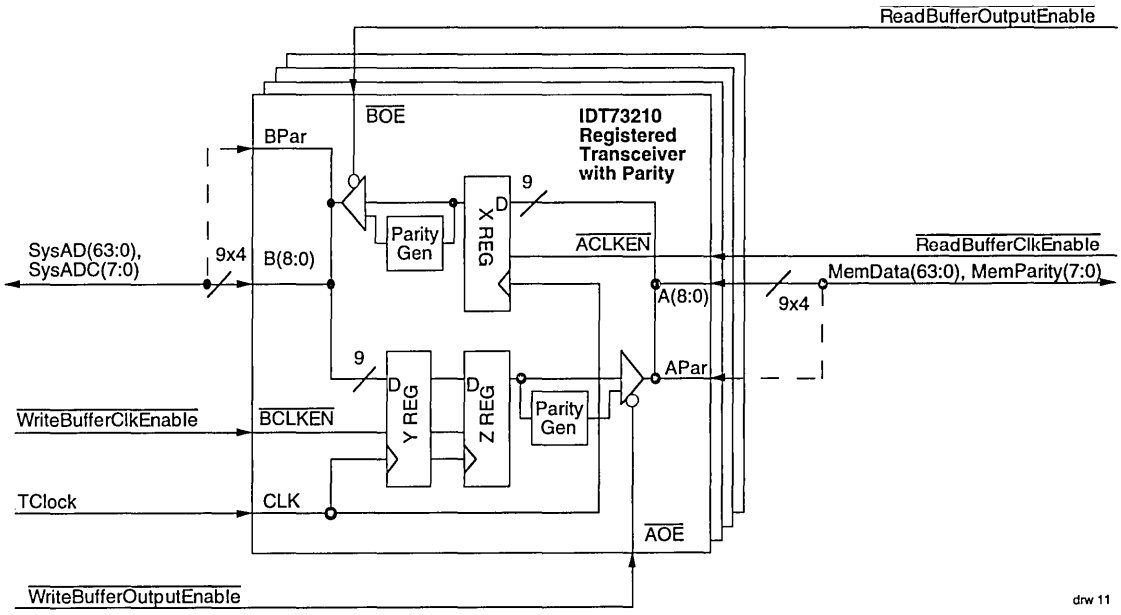


Figure 11. R4000 Integrated Read and Write Buffer

SUMMARY

The R4000 uses three groups of signals for its System Interface between the CPU and main memory, consisting of the SysAD bus, the SysCmd bus, and a small group of control signals. Even though the R4000 uses a high-speed 50MHz bus, worst case timing issues with the R4000 System Interface are greatly simplified because of the completely synchronized bus and control signals. The read and write system interface on the R4000 uses a concept of multi-level buffering/registering to maintain high throughput, by preventing unnecessary stalls and by allowing operations such as writes to happen in parallel with cached instructions and data. By using multi-level buffering on writes, the CPU can continue to run from cache while the main memory system retires writes at its own speed. Examples using off-the-shelf interface parts such as the FCT16823T 18-bit register with data enable, the 16-bit IDT73200 pipelined register, and the IDT73210 8-bit 2-level/1-level registered transceiver show how to easily implement read and write buffers for the R4000.

Integrated Device Technology, Inc.

HARDWARE AND SOFTWARE BOOT INITALIZATION OF THE IDT79R4000

APPLICATION NOTE AN-119

By Andrew Ng

INTRODUCTION

This application note is aimed at engineers that are bringing up or debugging an R4000 system prototype for the first time. Various debug techniques, pitfalls, and diagnostics are discussed that are based on similar experiences of other engineers here at IDT. The discussions will be a mixture of both hardware and assembly code software, since both hardware and software skills and techniques are required to initialize the part. In places, the R4000 User's Manual [2] will need to be referred to for more detail. The topics will proceed in a chronological order that begins with power on, continuing through the Reset sequence and finishing with some simple diagnostics — similar to the order that one might take when actually debugging a prototype board.

The first section details the hardware Reset sequencing, which includes managing the various Reset control lines and loading the R4000s serial configuration register. After the Reset sequence, the R4000 issues its initial instruction fetch. Logic analyzer connections are discussed so that the instruction fetch and other System Interface reads and writes can be verified. Then, the first few lines of boot assembly code, which determine some of the software programmable configuration options that the R4000 can do, are discussed. Some example assembly code for the initial testing of uncached read and write cycles to memory and I/O is given. Finally, in the last section, initialization of the caches is discussed so that block reads and writes can be executed and debugged. After reaching this stage of the debug, the chances of an operating system kernel booting up with a prompt are fairly good.

HARDWARE RESET SEQUENCE

In Figure 1, the R4000 Reset Interface requires the generation of several control signals, including VCCOk, $\overline{\text{ColdReset}}$, and $\overline{\text{Reset}}$. Primarily, these signals distinguish between power-on Resets, power-on-cold resets and power-on-warm resets, and to allow sufficient time for the PLL (Phase Locked Loop) circuitry to stabilize. Only the power-on reset is discussed in detail, since the cold and warm resets controls are a subset of the power-on case.

The first requirement is that VCCOk, which indicates that the supply voltage has reached at least 4.75V for 100ms or more, be de-asserted. The 100 ms de-assertion time is typically accomplished by using a power management chip which delays a power-up signal until a fixed time period or RC (Resistor/Capacitor) constant has elapsed. The power-up signal can be double-registered so that it is synchronized for the assertion of VCCOk. $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ must be de-asserted sometime before VCCOk is asserted. De-asserting VCCOk holds both the ModeClock and the output clocks, such as MasterOut, HIGH. (Although the ModeClock is guaranteed to be HIGH, the value of MasterOut is not guaranteed, technically, until after the PLL synchronizes). If MasterOut is used to clock the reset circuitry state machine, $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ must be de-asserted asynchronously from the output clocks. Technically, $\overline{\text{ColdReset}}$ and $\overline{\text{Reset}}$ are sampled synchronously when asserting and de-asserting. Therefore, while using the input clock, MasterIn to clock the reset circuitry state machine may make more sense than using MasterOut.

In Figure 2, 128 Master Clocks (either MasterIn or MasterOut) after VCCOk is asserted, the ModeClock will begin toggling by first going LOW and then 128 Master Clocks after that going HIGH for the first time. Thus the ModeClock period is 256 Master Clocks. On the first rising edge of the ModeClock, the R4000 starts accepting serial data on the ModeIn pin. Many systems use an Nx1 bit serial PROM for this function. Because the setting of the mode bits can be somewhat experimental when first bringing up a system, one might choose a reprogrammable serial bit EEPROM, or, perhaps, use a signal generator. Most serial bit PROMs have a built-in address incrementor/counter which requires a Clock input pin and a Reset input pin, in addition to the Data output pin. Thus, the serial PROM has an internal counter to generate the address for the mode bit data. When using a signal generator, one should consider designing in an inverter to invert the ModeClock, so the pattern generator can synchronize on the first falling edge of ModeClock, and, thus drive valid data in time for the first ModeClock rising edge. Using the inverted the ModeClock also provides ample hold time.

Sometime after the mode bits have been read, the R4000 will begin driving the output clocks. From the point where VCCOk is asserted, the R4000 needs to see a minimum of 64K Master Clocks (either MasterIn or MasterOut, which is just enough to read all the mode bits). A time of at least 100ms is more realistic before $\overline{\text{ColdReset}}$ can be de-asserted, so internal syncing of the PLL can be completed and fully stabilized throughout the system. Several ways exist to count out this period (a 50MHz MasterIn clock is assumed). One is to use a 24-bit counter based off the MasterIn clock. Another is to use a RC circuit to generate a 100 ms delay from VCCOk and then synchronize the resulting $\overline{\text{ColdReset}}$ signal by double-registering it. Another is to use a 16-bit counter based off the ModeClock, which, although not specified, continues to toggle, even after the mode bits have been read in. A fourth method can use some serial PROMs, which have a count/done pin that asserts LOW after all the bits have been read. If the number of bits is greater than 32K, then an adequate delay can be generated.

After $\overline{\text{ColdReset}}$ is de-asserted, then $\overline{\text{Reset}}$ must be de-asserted after a minimum of 64 Master Clocks have occurred. This requires a 6-bit counter, since $\overline{\text{Reset}}$ must be de-asserted synchronously.

The sequence for cold Resets is the same as power-up Resets, except that VCCOk needs only to be de-asserted for 64 Master Clocks, instead of 100ms. The sequence for warm Resets requires only that the $\overline{\text{Reset}}$ has asserted for 64 Master Clocks.

After the reset sequence, the R4000 will assert $\overline{\text{ValidOut}}$ along with an uncached read of the first instruction. The first instruction fetch will be discussed in more detail after the following section, which continues to specify the boot Reset configuration serial bits.

SERIAL BOOT MODE PROM — SPECIFIC CASES

The R4000 requires that 256 bits be serially loaded into its initialization logic on its ModeIn input pin for the first 256 ModeClocks. Of the 256 bits, only the first 64 are defined. Although specific systems will have specific values, an example of some "workable" values that can be used as a start for debugging are listed in Table 1 in binary. The rest are reserved to 0. Most of the bits are described by the R4000

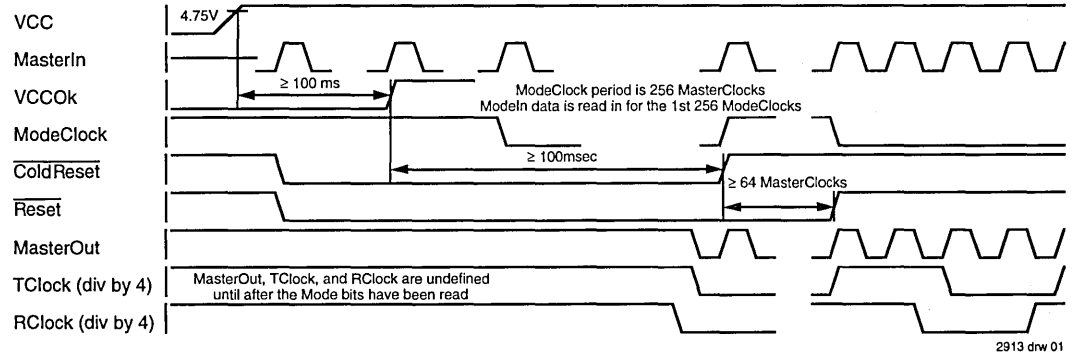


Figure 1. R4000 Reset Timing

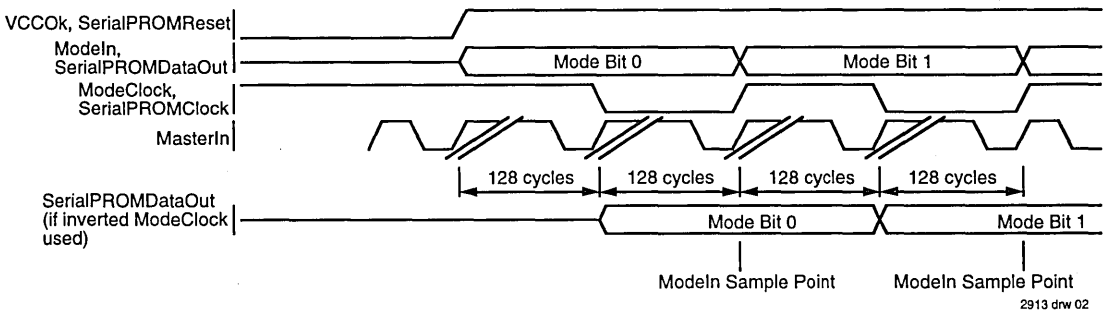


Figure 2. R4000 Serial Initialization Timing

Users Manual [2]. However, the values to choose for some bits can be confusing during initial debug. An example is the PLLOn configuration bit. This bit is intended only for chip testing and should be left asserted. The symptom is that the MasterOut and other clock outputs will not toggle. The implication is that the lowest MasterIn clock speed that can be used is 25MHz (for a 50MHz part). However, the SClock divisor configuration bits called SysCkRatio can be programmed to divide by 2, 3, or 4 which can reduce the System Interface frequency to 6.25MHz. One of the most common and perplexing hindrances in finding problems, at 50MHz, is having a noisy clock line to one of the state machines. This noise can clock a signal twice, or perhaps not at all. Therefore, reducing the System Interface frequency during the initial stages of testing is highly recommended.

TABLE 1. EXAMPLE OF SERIAL BOOT PROM VALUES.

Mode Setting	Value	Comments
BlkOrder	1	1 for sub-block ordering if PC, 0 for sequential ordering if SC/MC
EIBParMode	0	ECC
EndBit	0	Little Endian ordering
DShMdId	0	dirty shared mode enabled
NoSCMode	0	present (depends on package type)
SysPort	00	64 bits
SC64BitMd	0	128 bits
EISpltMd	0	Secondary cache unified
SCBlkSz	11	Secondary block size of 32 words (depends on system)
XmitDatPat	0000	Xmit Data Pattern DD (depends on system)
SysClkRatio	010	system interface bus divided by 4 (see text)
reserved	0	
TimIntDis	0	timer interrupt connection enabled
PotUpdDis	0	potential updates disabled
TWrSUp	0011	(SC write de-assertion delay, depends on SC timing, minimum shown)
TWr2Dly	01	(SC write assertion delay 2, depends on SC timing, minimum shown)
TWr1Dly	01	(SC write assertion delay 1, depends on SC timing, minimum shown)
TWrRc	0	(SC write recovery time, depends on SC timing, minimum shown)
TDis	010	(SC disable time, depends on SC timing, minimum shown)
TRd2Cyc	0011	(SC read cycle time 2, depends on SC timing, minimum shown)
TRd1Cyc	0100	(SC read cycle time 1, depends on SC timing, minimum shown)
reserved	0000	
Pkg179	0	Large Package (depends on package type)
CycDivisor	0011	power down clock divisor
Drv	100	1 clock Drive delay
InitP	0001	pull down di/dt (msb is opposite most fields)
InitN	1000	pull up di/dt
EnblDPLLR	0	disable di/dt mechanism during cold Reset
EnblDPLL	0	disable di/dt mechanism
DsblPLL	0	Enable PLLs (see text)
SRTristate	1	tri-state when Reset or ColdReset is asserted
Bits65:255	0	rest of the bits are reserved

2913 tbl 01

During debug, other serial boot configuration bits that may be of use are the SCBlkSize, which configure the secondary cache line size, if present, to 4, 8, 16, or 32 words. This will control the maximum size of block reads and writes for secondary cache systems. Also, the XmitDatPat bits configure the system interface data rate with various patterns such as D, DDx, DDxx, etc. Another design consideration is if the secondary cache is not used, then sub-block ordering, as programmed with the BlkOrder bit, is mandatory.

BASIC LOGIC ANALYZER CONNECTIONS

After the serial configuration register is read, the majority of the debug effort centers around memory bus cycles on the System Interface. For this reason it is recommended that most of the System Interface be accessible from a Logic Analyzer. This includes the information on the SysAD(63:0) bus.

Two items should be considered when attaching the SysAD bus to a logic analyzer. The first is the latching control circuitry of the SysAD bus as shown in Figure 3. To demultiplex it into separate MemAddr(35:0) and MemData(63:0) busses is usually straightforward, but the multi-level write buffering of SysAD into the MemAddr and MemData is not. Thus, if there are enough pod connections, one should hook up MemData, MemAddr, and SysAD. However, the second consideration is that there usually are not enough pods or probes to do this. Therefore, in a compromise, attaching SysAD is probably more useful than attaching MemAddr, since MemAddr is usually a single level deep register, latch, or buffer. However, it is essential to look at the least significant MemAddr lines to verify that the address can be incremented within a block correctly, especially if sub-block ordering is used. Also, using MemAddr instead of SysAD only requires 36 probes, and possibly less, if not all the physical address lines are used. A

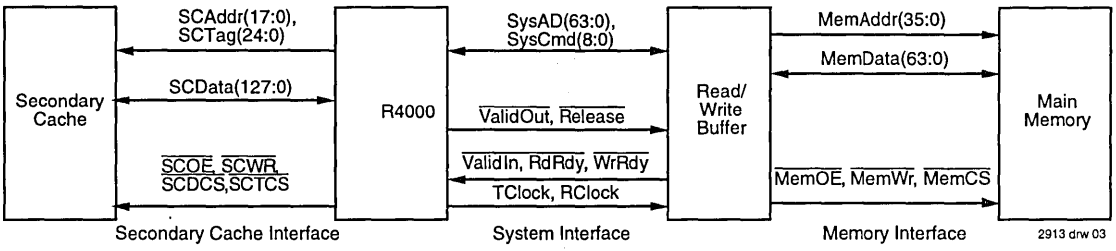


Figure 3. Typical R4000 System

make-shift solution is to hookup only 32 data lines at a time, either MemData(63:32) or MemData(31:0). The upper/lower halves can be swapped as needed, since, during the initial debug, the function of the lines is more important than examining the sequential flow of instructions and data.

It is also essential to bring out the entire SysCmd(8:0) bus. This bus acts as the control and status lines, and determines whether the transaction is a read or write, etc. Along with SysCmd bus, ValidOut, ValidIn, and Release are essential, since they indicate when the SysAD and SysCmd busses are valid, and when they can be driven by the memory system. RdRdy and WrRdy and read/write buffer control lines such as MemOE and MemWr are also sometimes needed.

If a state analyzer is being used, one should consider attaching the RClock output, which leads the TClock, that is usually used by 25% (of the TClock period), as the state clock to trigger the logic analyzer, so sufficient hold time is provided (at the expense of having less setup time). Otherwise one of the other output clocks, either the TClock or the MasterOut, should be attached.

Thus, the minimum number of logic analyzer probes needed is 64+9+3+1=77. A typical number would be 64+32+9+3+1=109 and could be as many as 64+36+64+9+3+1=177. Additional pods will be needed to test for specific cases, such as the control lines during Reset, the ECC bits during fault checking, etc.

If the secondary cache is present, one should be prepared to examine its interface. However, because of the enormous number of lines (128 data, up to 36 address and tag, and 4 control lines) and the relative straightforwardness of the functional design, the secondary cache will probably only need to be on the logic analyzer temporarily. The secondary cache lines may require oscilloscope probing to verify the electrical signal transmission line design. To help follow the processor flow, leaving the control lines SCOE and one of the SCWr lines connected to the logic analyzer at all times can be helpful.

MINIMAL SOFTWARE BOOT CODE

After Reset, the R4000 will be executing instructions out of uncached memory kernel segment 1 space at virtual address 'h bfc0 0000, which is hard mapped to physical address 'h 01fc0 0000. ValidOut will assert LOW, and the SysCmd(8:0) bus will indicate an uncached read of 1 word, 'b 10011011, and, on a little endian machine, will expect data on SysAD(31:0) at the same time ValidIn is asserted. Big endian machines will expect data on SysAD(63:32). During uncached reads of addresses divisible by 8, (number of bytes per double word), SysAD(63:32) will be ignored on little endian machines. Big endian machines will ignore SysAD(31:0). The second instruction fetch will be similar, except it will be at physical address 'h 01fc0 0004, and a little endian machine will expect the data to be put on SysAD(63:32), with ValidIn asserted, while SysAD(31:0) is ignored. Likewise, big endian machines will expect the data to be put on SysAD(31:0), while SysAD(63:32) is ignored. The minimal boot code discussed here will get the part initialized and allow various types of memory accesses to take place. This includes initializing the caches so that block reads and writes can be tested.

One common cause of no system commands being generated (ValidOut never asserts), is the GroupStall input pin (if present for the particular R4000 version/type) has to be deasserted.

The next section will discuss the very first operation software should do, namely, initializing the software configuration registers. After initializing the registers, the software can execute various kinds of reads and writes to uncached memory space in order to test the ROM, I/O and RAM chip selects, byte enables, and wait-state timing.

Configuration Registers $14 and $16

The first operation that the boot code software needs to perform is to initialize the software configurable registers. This includes the Status Register and Configuration Register. Most of the registers do not have default values on Reset, and must be programmed before being used. The Status Register, Configuration Register, and the WatchLo Register have effects on loads, stores, and processor operations that must immediately be programmed into a known state. An example of programming Status Register $14 and Configuration Register $16 settings is shown in Listing 1. The other general purpose and coprocessor registers, including the Timer and Compare registers must be initialized before they are used.

```
set noreorder
li              v0,0x30410000   # load constant with CP1, CP0 usable,
                                #  BEV, DE set, IE (interrupts) disabled
mtc0    v0,$14          # move it to the Status Reg
mtc0    zero,$18    # clear R and W trap enable masks in the WatchLo Reg
nop                             # an operation is needed between a mtc0 and mfc0 instruction
mfc0    v1,$16          # get Configuration Reg
nop                             # delay two operations before v1 can be used
li              a0,0xa0000160   # load address constant
sw              v1,0(a0)        # dump Configuration Reg to external memory
li              v0,0x00000033   # load constant with IB, DB set to 32 byte p-cache line widths
                                #   and Kseg0 to be non-coherent cachable
mtc0    v0,$16          # move it back to the Configuration Reg (only bits 5:0 writable)
```

Listing 1. Software for Reading and Writing the Configuration Registers

Figure 4 shows the register fields. Refer to the User's Manual [2] for more detail.

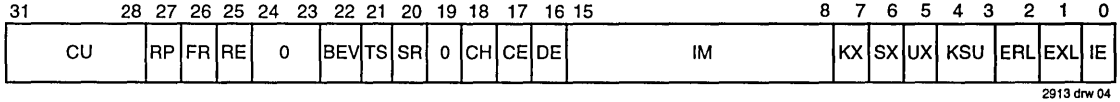


Figure 4. Status Register $14

Two suggestions on programming these fields during initial debugging are to set the BEV bit and the DE bit. Setting BEV, bit 22, the Diagnostic Status Field of the Status Register $14 will send any exceptions to the uncached kernel segment 1 bootstrap exception vector base virtual address 'h bfc0 0200, instead of to the cachable mapped user segment 'h 8000 0000, which requires that the cache and TLB be initialized first. An exception handler for initial diagnostics, such as the (unoptimized) one in Listing 2, can put code at physical address 'h 01fc0 0200 and offsets 'h 0000, 'h 0080, 'h 0080, 'h 0100, and 'h 0180, i.e., physical addresses, 'h 01fc0 0200, 'h 01fc0 0280, 'h 01fc0 0300, 'h 01fc0 0380. The exception handler should at least dump out the cause register $13, the exception vector, $14, and the cache error register $27, and the error exception program counter, $30. If the registers can't be displayed with a UART, they should at least be written out to uncached memory so they can be observed on a logic analyzer. In contrast to the R3000 RFE instruction, the R4000 uses an ERET instruction to return back to the code.

```
li      a0,0xa0000000 # load address constant

mfc0    v1,$13          # get Cause Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x130(a0)    # dump to memory

mfc0    v1,$14          # get EPC
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x140(a0)    # dump to memory

mfc0    v1,$27          # get CacheErr Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x270(a0)    # dump to memory

mfc0    v1,$30          # get ErrorEPC Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x300(a0)    # dump to memory

mfc0    v1,$12          # get Status Reg
nop                     # two non-v1 operations needed
nop                     #
sw      v1,0x120(a0)    # dump to memory

eret                    # return from exception
```

Listing 2. Software for the Exception Handler

The DE bit is bit 16 in the Diagnostic Status Field of the Status Register $14, when set specifies that cache parity and ECC errors don't cause exceptions. This is somewhat necessary when initializing the cache, otherwise a lot of unnecessarily confusing jumps to the exception handler will probably occur as the cache locations are first initialized, since the tag and data parity haven't been initialized yet.

The WatchLo Register must have its trap on a read/load mask (bit 1) and trap on a write/store mask (bit 0) disabled before loads or stores are attempted, so an inadvertent trap from an address match is not taken. Thus, similar to the example in Listing 1, the WatchLo register can be cleared.

Reading the Configuration Register and dumping its contents out to uncached space allows one to see if various bits in the boot serial PROM were programmed correctly. Figure 4 shows the Configuration Register fields. For instance, the System clock ratio and the transmit data pattern can be checked. The only writable bits are the lower 6, which are also uninitialized on Reset, and, therefore, should be written to by software as soon as possible. Of the writable bits, IB and DB are used to program the primary Instruction Cache line size and the primary Data cache line size to either 4 words or 8 words. The primary cache line sizes must be smaller than or equal to the secondary cache line size. Note that if there is no secondary cache, it is possible to program the data and instruction caches to different line sizes, which is the one case where different block sizes will be presented to the system interface.

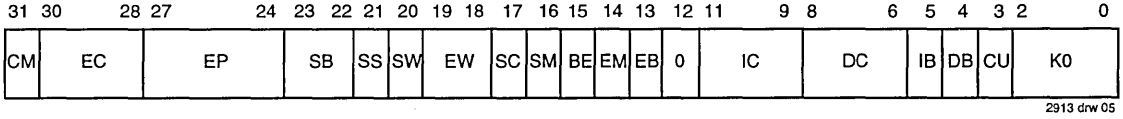


Figure 4. Configuration Register $16

PRIMARY DATA CACHE INITIALIZATION

In general, it is much simpler to test the data cache than it is the instruction cache. Several reasons exist for this. First, if the data cache read fails, the program can still continue, where as an instruction cache failure may or may not continue and could cause the program to get lost. Second, it is simpler to initialize the data cache since it can be written directly with stores. Finally, forcing cache miss writebacks is more straightforward, since it just requires writing to different addresses as opposed to jumping back and forth in code. As shown in Listing 3, when initializing the caches, the Cache opcode is used heavily. The algorithm in Listing 3 is not the most efficient. However, from a debugging point of view, it does not do any unnecessary System Interface block reads or writes. The idea is to, first, invalidate the tags, and then fill the data slots with any data so that ECC/parity can be set correctly. The base virtual address, 'h 8000 0000, is used because it is in the unmapped cachable kernel segment 0, which does not require the TLB. Note that if an R3000 compiler is being used, which can't generate the R4000 Cache opcode, then a data statement using the ".word" directive can be inserted into the program with the data for the hand assembled hex machine instruction.

In a similar manner, by substituting the appropriate cache instructions, and by adjusting for the cache line size, the secondary data cache can be initialized.

```
   set noreorder                          /* turn off assembly rescheduler (no reordering optimization) */
   li          a0,0x80000000              /* primary data cache start pointer */
   li          a1,0x80002000-0x20         /* 8K last location - 32 */
   mtc0  zero,$28                         /* set TagLo CP0 Reg to 0 */
#ifndef R3000asm
1:       cache     2*4+1,0x00(a0)         /* Index Store Tag, invalidate cache line (prevent writebacks) */
         cache     3*4+1,0x00(a0)         /* Create Dirty Exclusive (prevent block reads) */
#else
1:       .word     0xbc890000             /* use if using R3000 assembler */
         .word     0xbc8d0000
#endif
         nop
         nop
         sw        zero,0x00(a0)          /* fill data slots with good ECC/parity (8 word cache lin        e) */
         sw        zero,0x04(a0)
         sw        zero,0x08(a0)
         sw        zero,0x0c(a0)
         sw        zero,0x10(a0)
         sw        zero,0x14(a0)
         sw        zero,0x18(a0)
         sw        zero,0x1c(a0)
         nop
         nop
#ifndef R3000asm
         cache     2*4+1,0x00(a0)         /* Index Store Tag, invalidate cache line */
#else
         .word     0xbc890000
#endif
         blt       a0,a1,1b               /* if count is less than last */
                                          /* then jump Back to last label called "1". */
         addu      a0,0x20                /* branch delay slot, increment addr pointer */
```

Listing 3. Primary Data Cache Initialization Software

PRIMARY INSTRUCTION CACHE INITIALIZATION

As shown in Listing 4, the instruction cache is initialized a little differently than the data cache. First, their data slots need to be filled from main memory, using the Fill Cache operation, so the ECC/parity for the data can be set correctly. Then, their tags are invalidated and tag ECC/parity set. As with the data cache, the base virtual address 'h 8000 0000 is used because it automatically maps to a physical address without requiring the use of the TLB.

The secondary instruction cache can be initialized in a similar manner to the primary cache. The initialization can be accomplished by using the cache fill instruction over the entire secondary cache address space, adjusting for the cache line size, and by substituting the appropriate cache instructions. Note, that if the secondary cache has a unified instruction and data memory, then the cache only needs to initialized once.

```
        .set noreorder                  /* turn off assembly rescheduler */
                                        /*  (no reordering optimization) */

        li      a0,0x80000000           /* primary data cache start pointer */
        li      a1,0x80002000-0x20      /* 8K last location - 32 */
        mtc0    zero,$28                /* set TagLo CP0 Reg to 0 */

1:
#ifndef block_reads_are_being_tested_later
        cache   5*4+0,0x00(a0)          /* fill i-word data slots (8 word cache line size) */
                                        /* 0xbc940000 */
                                        /* note that the fill operation requires that block */
                                        /* reads are working. Thus during initial debug */
#endif                                  /* one may want to delete the fill operation */

        cache   2*4+0,0x00(a0)          /* index store tag */ /* 0xbc880000 */
        blt     a0,a1,1b                /* if count is less than last */
                                        /* then jump Back to last label called "1". */
        addu    a0,0x20                 /* branch delay slot, increment addr pointer */
```

Listing 4. Primary Instruction Cache Initialization Software

MINIMAL TEST CODE (BLOCK READS AND WRITES)

Cachable data loads will read from the internal primary cache or the secondary cache, unless the cache line location is invalid or has a non-matching tag. Such cache misses will generate block reads to the external system interface.

The block reads are tested by doing a cached read, which misses in the cache. It is easier to look at cache locations that are initialized as invalid, so writebacks do not occur.

The data cache uses a writeback protocol. So, when writing to a cached location, the data is stored only to the cache, and a dirty bit is set. Main memory is updated later, when the cache line, where the data was stored, is replaced for a cache miss. Because the cache is direct mapped, a cache miss can be created by writing or reading to locations that are modulo cache block size apart, i.e., every 8K apart.

Code in Listing 5 shows a method that may be needed early-on, which is to test writebacks without doing a block read first.

After block reads and writes are tested individually, data writes to cache block offsets of 8K, as in Listing 6, will force a writeback. On the R4000PC without secondary cache, this will be two separate System Interface transactions. However, on R4000s with secondary cache, the write address and data will be issued immediately following the read address, such that the write address and data will come between the read address and when data is returned by the system. In a typical system, the write address and data is FIFO buffered such that after the read is handled, the system issues the write to main memory.

```
/* assume that cache has just been flushed (invalidated) */

li      a0,0x80000000       /* start addr pointer */
cache   3*4+1, 0x00(a0)     /* Create Dirty Exclusive, otherwise a block read
                               will occur on the first store so that the entire cache
                               line is filled */ /* 0xbc8d0000 */
nop
addiu   a1,a0,0x00          /* store incrementing pattern, i.e., 0x0, 0x4, 0x8, 0xC */
sw      a1,0x00(a0)         /* into cache */
addiu   a1,a0,0x04
sw      a1,0x04(a0)
addiu   a1,a0,0x04
sw      a1,0x08(a0)
addiu   a1,a0,0x04
sw      a1,0x0c(a0)
addiu   a1,a0,0x04
sw      a1,0x10(a0)
addi    a1,a0,0x04
sw      a1,0x14(a0)
addiu   a1,a0,0x04
sw      a1,0x18(a0)
addiu   a1,a0,0x04
sw      a1,0x1c(a0)
nop                         /* 2 operations required between store and cache */
nop
cache   0*0+1,0x00(a0)      /* index write back invalidate */ /* 0xbc810000 */
```

Listing 5. Block Write Code with No Block Read

```
li      a0,0x80000000       /* load start addr pointer */
li      a1, zero            /* load data */
sw      a1,0x0000(a0)       /* read from 0000 and possible writeback to xxxx */
li      a1,0x2000           /* load data */
sw      a1,0x2000(a0)       /* read from 2000 and definite writeback to 0000 */
```

Listing 6. Block Read with Writeback

TESTING ALL THE PHYSICAL ADDRESS LINES

The R4000 has 36 of the physical address lines implemented. Although unspecified, one can customarily expect SysAD(63:36) to be 0 during any address phase. Only the bottom 30 out of 36 physical address bits can be tested within the unmapped fixed kernel space provided with 32-bit virtual addressing. One way to test address bits 35:32 is to go into 64-bit virtual addressing by setting the KX (bit 7) in the Status Register $14 and then using the 64-bit kernel space called xkphys. Virtual addresses 'h 9000 0000 0000 0000 to 'h 97ff ffff ffff ffff are uncached and automatically mapped such that physical address bits 35:0 are the same as virtual address bits 35:0.

A second, but more tedious way to test address bits 35:30, is to use the mapped space via the Translation Lookaside Buffer (TLB) which converts the software program's virtual address into the hardware's physical address. Although initialization of the TLB is beyond the scope of this application note, one tip includes initializing all 48 entries, not just the ones going to be used. This is because the unused entries may happen to power up with a matching virtual address. Should two or more TLB entries match, a TLB shutdown may occur and the CPU does not know which one to choose. In addition, initialize the TLB virtual pages to an unmappable unmapped virtual address space such as 'h 0x8000 0000 as well as setting the entry's Valid bit to invalid. This is because the TLB shutdown logic, when two or more entries match, does not take into account the valid bit. Since 'h 8000 0000 is automatically mapped to a physical address space, and does not go through the TLB, those entries cannot accidently cause a shutdown.

SUMMARY

Bringing up the hardware requires a mixture of hardware and software. The part must be Reset, serial configuration registers loaded and software configuration registers written. A mixture of single doubleword reads, writes, block reads, and block writes can be checked. Reaching this stage is usually sufficient to continue with more intensive diagnostics and operating systems. Continued diagnostics may include interrupt line checks, memory checks, and I/O initialization.

FOR FURTHER INFORMATION

[1] MIPS R4000 Microprocessor Introduction, Integrated Device Technology, Inc., MAN-RISC-10091, Santa Clara, CA, 1991. — Gives a brief general overview of the architecture and features.

[2] MIPS R4000 User's Manual, Integrated Device Technology, Inc., MAN-RISC-00091, Santa Clara, CA, 1991. — Describes the H/W features and functionality of the device as well the bus interface. Also describes the R4000 instruction set architecture from a systems and assembly level programming perspective.

[3] IDT79R4000 Family Data Sheet, Integrated Device Technology, Inc., Oct. 1991. — Contains the Data Sheet with packaging, pinout, AC/DC electrical specifications and thermal parameters.

Integrated Device Technology, Inc.

TIMERS USING SONIC™ AND COUNT REGISTER IN ORION™

APPLICATION NOTE AN-127

By Sugan Subramanian

INTRODUCTION

This is an Application Note on two timer modules, one based on SONIC™ running at 20MHz and another based on COUNT register present in R4600™ running at 50MHz. The timer-modules are broken up into two groups of functions. The first group of functions is specific to a particular timer. These functions are always written in assembly language and do low level timer specific initialization for starting and stopping the timer. The second set of functions have generic functionalities. These functions are written in C and they are used to keep track of the number of ticks in the timer in microseonds between the period at which the timer is started and stopped, to display time, and to set a constant value based on the current speed at which the processor is set to run. First, we are going to discuss how to measure time using SONIC. Secondly, we going to look at how to keep track of time using R4600's count register. Finally, we will discuss how to use the timers.

SONIC TIMER

SONIC is used to measure time in various boards, including IDT's 79S460-board (an R4x00 evaluation board) and 79S381™-board (a R30xx evaluation board). The SONIC has a 32-bit downcounter that is controlled by SONIC's 16-bit registers, watchdog register1 and watchdog register2. In the SONIC running at 20MHz, each timer-tick represents 200-ns. In general,

$$t = (1000/f)*4$$

where t is the time for each timer-cycle in ns and f is the frequency of SONIC in MHz. Moreover, (1000/f) represents time period per cycle. Sonic decrements the counter registers once every four cycles. Mechanisms involved in starting the timer, stopping the timer, and displaying time using SONIC are discussed in the following sections. The following figure describes the header file "sonic_globals.h" used by SONIC-timer's high level functions.

```
/* sonic_globals.h */

/* speed based on user specified
frequency of SONIC */
unsigned int current_speed;
/* speed based on default frequency of
SONIC */
unsigned int def_speed;
/* counter keeps track of the time count
*/
unsigned int counter;
```

Figure 1
sonic_globals.h

Starting SONIC Timer

Starting the timer involves enabling the ST bit in SONIC's control register, and initializing 16-bit watchdog registers 1 and 2 with all their bits set to 1 (0xffff). "TimerStart", a low level function, does the initialization required for starting a timer. "timer_start", a high level function, calls "TimerStart" and keeps track of the number of timer ticks. Figures 2 and 3 describe "TimerStart" and "timer_start" routines for SONIC.

```
/*for 79s460-board */
#include <r4ksonic.h>
/*for 79s381-board
#include <r3ksonic.h>
But, don't include both headers */
#    LOW LEVEL FUNCTION TimerStart

     .globl  TimerStart
     .ent    TimerStart
     .set    noreorder
TimerStart:
     li      v0, SONIC_COMMAND_REG_ST_BIT
     li      t1, SONIC_COMMAND_REG
     nop
     nop
     sw      v0,0(t1)
     nop
     nop
     li      v0, oxffffffff
     sw      v0, SONIC_WATCHDOG1(t1)
     nop
     nop
     sw      v0, SONIC_WATCHDOG2(t1)
     nop
     nop
     j       ra
     nop
     nop
     .end    TimerStart
```

Figure 2
SONIC TimerStart routine

```
     #include <sonic_globals.h>
     /* HIGH LEVEL FUNCTION timer_start */
     unsigned int timer_start()
     {
     if (cur_speed)
/* set cur_speed of the SONIC based on the user
specified frequency */
             def_speed = cur_speed;

     counter = TimerStart();
     return counter;
     }
```

Figure 3
SONIC timer_start routine

Stopping SONIC Timer

The SONIC-timer is stopped by enabling the STP bit in SONIC's control register. By enabling STP bit, the SONIC preserves the previous values of WATCHDOG registers 1 and 2. "TimerStop" returns a 32-bit value that is a concatenation of 16-bit count in watchdog registers 1 and 2. "timer_stop" function calls the low level "TimerStop" to retrieve the current timer value. "timer_stop" returns an unsigned integer value representing the time period between the previous timer initiation and the current instance of execution (in microseconds). Figures 4 and 5 describe "TimerStop" and "timer_stop" routines for SONIC.

```
/*for 79s460-board */
#include <r4ksonic.h>
/*for 79s381-board
#include <r3ksonic.h>
But, don't include both headers */
     .globl  TimerStop
     .ent    TimerStop
     .set    noreorder
TimerStop:
     li      v0, SONIC_COMMAND_REG_STP_BIT
     li      t1, SONIC_COMMAND_REG
     nop
     nop
     sw      v0,0(t1)
     nop
     nop
     lw      v1, SONIC_WATCHDOG1(t1)
     nop
     nop
     lw      v0, SONIC_WATCHDOG2(t1)
     sll     v1,16
     addu    v1,v1,v0
     addu    v0,v1,v1
     j       ra
     nop
     .end    TimerStop
```

Figure 4
SONIC TimerStop routine

```
/* HIGH LEVEL FUNCTION timer_stop */
extern unsigned int counter;
unsigned int timer_stop()
{
     counter -= TimerStop();
/*
     when counter == i,
     i*100 == time in nanosecs
     (i/1000)*100 == time in microsecs
*/
     return counter/10;
}
```

Figure 5
SONIC TimerStop routine

Displaying Time (SONIC)

"disp_time" function displays the time. It is always called after a call to "timer_stop". It takes a parameter which is usually zero, otherwise represents current time-count in microseconds. It displays the timer period in the following format:
" %dS %dmS %duS" where %d represents the number of seconds in the time count; mS - represents the number of milliseconds in the time count; and uS - represents the number of microseconds in the time count. Only the non-zero units are displayed. Figure 6 describes the fuction "disp_time" for SONIC.

```
extern unsigned int counter, def_speed,
     cur_speed;

void disp_time(unsigned int i)
{
     unsigned int temp_counter=counter;

     if (i)
          temp_counter = i;

     printf("elasped time = ");
     if (temp_counter >
          1000000)
     {
          printf("%dS ",
          temp_counter
          /1000000);
          temp_counter %=1000000;
     }

     if (temp_counter > 1000)
     {
          printf("%dmS",
          temp_counter
          /(1000));
          temp_counter %=1000;
     }

     if (temp_counter)
     {
          printf("%duS ",
          temp_counter);
     }

     printf("\n");
}
```

Figure 6
SONIC disp_time routine

Setting Up SONIC timer speed

"set_timer_speed" takes an integer that represents the clock frequency of SONIC in MHz as its parameter and sets a constant that represents the speed of SONIC in ns for the given frequency. This constant is used by the "disp_time" to display the time correctly. The following figure has the C-source for "set_timer_speed".

```
extern unsigned int cur_speed;
void set_timer_speed(int speed)
{
     cur_speed = 1000/speed*2;
}
```

Figure 7
SONIC set_timer_speed routine

Difference between SONIC timer on 79s460-board and 79s381-board

It's recommended to include the header file, "timer_sonic.h" before encorporating the timer routines in his/her code. Only difference between the SONIC timer routine for IDT's 79s460-board (R4xxx evaluation board) and 79s381-board (R30xx evaluation board) is the SONIC chip's base address. The header file "r4ksonic.h" has R4xxx board specific SONIC base address and "r3ksonic.h" has R30xx board specific SONIC base address. The following figures describe these header files.

```
/* include file r4ksonic.h */

#define      SONIC_BASE
     0xbf600000
#define      SONIC_COMMAND_REG
     0xbf600000
#define      SONIC_WATCHDOG1
     0xa4

#define      SONIC_WATCHDOG2
     0xa8
#define      SONIC_COMMAND_REG_ST_BIT
     0x20
#define      SONIC_COMMAND_REG_STP_BIT
     0x10
```

Figure 8
r4ksonic.h

```
/* include file r3ksonic.h */

#define      SONIC_BASE
     0xbfb00000
#define      SONIC_COMMAND_REG

     0xbfb00000
#define      SONIC_WATCHDOG1
     0xa4
#define      SONIC_WATCHDOG2
     0xa8
#define      SONIC_COMMAND_REG_ST_BIT
     0x20
#define      SONIC_COMMAND_REG_STP_BIT
     0x10
```

Figure 9
r3ksonic.h

Constraints on SONIC timer

The SONIC timer module on a SONIC running at 20mHz is capable of counting upto 7 minutes. If the SONIC is running at a different frequency, function "set_timer_speed" should be called before calling "timer_start".

R4600 TIMER

R4600 microprocessor has a COUNT register in CoProcessor 0 (CP0). This COUNT register in CP0 increments its count by one on every timer tick of the R4600 processor. In the R4600 processor running at 50MHz, each timer-tick represents 20-ns. In general,

$$t = (1000/f)$$

where t is the time for each timer-tick in nano-seconds, f is the frequency of R4600 processor in MHz, and (1000/f) represents the time per cycle. The following figure describes the header file used by the high level functions of R4600 timer.

```
/* orion_globals.h */

/* speed based on user specified
frequency of R4600 processor */
unsigned int current_speed;
/* speed based on default frequency of
R4600 processor */
unsigned int def_speed;
/* counter keeps track of the time count
*/
unsigned int counter;
```

Figure 10
orion_globals.h

Starting R4600 Timer

Timer is started by resetting the value of the COUNT register in R4600 microprocessor to zero. The following piece of assembly code and c-code shows how to start the timer.

```
#include "r4kcp0.h"
# Timer is started by assigning zero to
# COUNT register located in CP0

     .globl  TimerStart
     .ent    TimerStart
     .set    noreorder
TimerStart:
     and     v0, $0
     mtc0    v0, CP0_COUNT
     nop
     nop
     j       ra
     nop
     nop
     .end    TimerStart
```

Figure 10
R4600 TimerStart routine

```
/* r4kcp0.h */
#define CP0_CONTEXT $4
#define CP0_BVADDR  $8
#define CP0_COUNT           $9
#define CP0_COMPARE $11

/* r4kcp0.h ---contd. */
#define      v0             $2
#define      v1             $3
#define      ra             $31
Figure 11
R4600 header file r4kcp0.h

#include <orion_globals.h>

unsigned int timer_start()
{
     if (cur_speed)
          def_speed = cur_speed;
```

```
        counter = TimerStart();
        return counter;
}
```

Figure 12
R4600 timer_start routine

Stopping R4600 Timer

Stopping the timer involves simply getting the contents of the COUNT register that represents the most recent timer tick count and converting that timer tick count to microseconds. The following piece of assembly code and c-code shows how to stop the timer.

```
#include "r4kcp0.h"
    .globl  TimerStop
    .ent    TimerStop
TimerStop:
    mfc0    v0, CP0_COUNT
# The timer is stopped by getting
# the most recent value of COUNT
# register
    nop
    nop
    j       ra
    nop
    nop
    .set    reorder
    .end    TimerStop
```

Figure 13
R4600 TimerStop routine

```
extern unsigned int counter;

unsigned int timer_stop()
{
    counter = TimerStop() - counter;
/*
    when counter == i,
    i*20 == time in nanosecs
    (i/1000)*20 == time in microsecs
*/
    return counter/50;
}
```

Figure 14
R4600 timer_stop routine

Displaying Time (R4600)

"disp_time" function is very similar to the one presented previously for the SONIC timer module. Only difference in this case is that the time displayed is based on the frequency of the R4600 in the 79s460-board (50-MHz).

Constraints on R4600 timer

The timer module is capable of counting upto 85 seconds assuming that the R4600 processor is set to run at 50MHz. If the R4600 processor is set to run at a different frequency, function "set_timer_speed" should be called before calling "timer_start" so that "disp_time" displays the correct time. The following piece of c-code describes the "set_timer_speed" function.

```
extern unsigned int cur_speed;
void set_timer_speed(int speed)
{
    cur_speed = 1000/speed;
}
```

Figure 15
R4600 set_timer_speed routine

TIMER MODULE USAGE

The procedures to use SONIC timer for 79s460-board, SONIC timer for 79s381-board, and R4600-timer for 79s460-board are the same. The following figure describes it. Moreover, IDT-C 5.0 is shipped with SONIC-timer/79s460-board and R4600-timer/79s460-board. In IDT-C 5.0, source code for SONIC-timer/79s460-board is located under "/IDTC/timers/SONIC-timer"; and source code for R4600-timer/79s460-board is located under "/IDTC/timers/ORION-timer". Figures 16 and 17 give the general procedure to use the timer routines.

```
#include <timer.h>
main()
{
timer_start();
.
/* main body */
.
timer_stop();
disp_time(0);
}
```

Figure 16
How the timer routine is to be used

```
/* timer.h */
unsigned int timer_start();
unsigned int timer_stop();
void disp_time(unsigned int);
```

Figure 17
timer.h

Integrated Device Technology, Inc.

R4600™ POWER CALCULATIONS

APPLICATION NOTE AN-129

By Robert Napaa

INTRODUCTION

The IDT R4600™ Orion™ RISC microprocessor is a full 64-bit architecture that is fully compatible with numerous 32-bit and 64-bit Operating Systems and applications. It is a highly integrated microprocessor designed to serve embedded applications. It incorporates large on-chip caches (16KBytes for both the instruction and the data caches); both two-way set associative. The R4600 implements a large TLB to map 96 virtual pages (raging from 4KB to 16MB in size) to their corresponding physical addresses. The R4600 has a four deep write buffer to isolate the high speed internal caches from the low speed external memory.

The R4600 uses advanced power management techniques to lower the peak and typical power consumption. The power saving is implemented through an intelligent scheme which turns off the power from the unused sections of the device (e.g. the FPU). A standby mode is also available which shuts down the internal clocks and freezes the pipeline, thus reducing the consumed power substantially. This feature is very desirable for power sensitive applications such as portable systems and notebooks.

This Application Note explains how to compute the R4600 power consumption under different working conditions and capacitive loading.

TYPES OF POWER

The data sheet of the R4600 lists three different modes of power consumption in the Icc table: Standby mode, Active_Typical mode and Active_Max mode. The R4600 operates in any one of these three modes. The mode of operation of the R4600 is under the system control (both S/W and H/W).

Standby Mode

The R4600 implements a Standby mode which is entered through software control using the WAIT instruction. Executing the WAIT instruction enables the interrupts and causes the CPU to enter the Standby mode. The Standby mode is actually entered when the WAIT instruction finishes the W stage of the pipeline. In this mode, the internal clocks are shutdown and the pipeline is frozen. No instruction advances through the pipeline and the external bus activity stops. However, the PLL, internal timer, some of the input pins (~Int[5:0], ~NMI, ~ExtReq, ~Reset, ~ColdReset, SyncIn and the MasterClock) and the output clocks (TClock[1:0], RClock[1:0], SyncOut, ModeClock and MasterOut) continue to run. In this mode, the R4600 consumes very little power which is reflected by the standby Icc values in the data sheet.

Once the CPU is in Standby mode, any unmasked interrupt, including the internally generated timer interrupt, will cause the CPU to exit the Standby mode.

Active_Maximum Mode

In this mode the R4600 is fully functional. The pipeline is continuously running, instructions are advancing through the pipeline and the CPU is accessing the internal caches and the system resources. In this mode, the power to all the internal units may be turned on. This is achieved if the code sequence uses and accesses all the internal units (such as the integer unit, the FPU , etc.) continuously. This mode also represents the worst case power consumption values, with the supply voltage at its max limit (e.g. 5.25V). In this mode, the R4600 consumes its max power and this is reflected by the max Icc values in the data sheet.

Active_Typical Mode

This mode is similar to the Active_Maximum mode with the exception that the instruction sequence doesn't fully exercise the internal resources (like the FPU for example). The R4600 implements advanced power management techniques to take advantage of such code sequences. In this mode, the unused sections of the device are powered down. For example, if the FPU is not used, it will be powered down to reduce the overall power consumption. This amounts to substantial power consumption savings compared to the maximum case. In this mode, the supply voltage is assumed to be at its mid-point nominal value (e.g. 5V or 3.3V). This mode is reflected by the typical Icc values in the data sheet. It represents the average (typical) power the device will consume in a typical application that is not fully utilizing the internal resources. In such typical applications, the CPU is usually executing instructions 75% of the time and stalled the remaining 25%.

COMPONENTS OF POWER CONSUMPTION

The total power consumption of the R4600 in the previous three modes includes two components: the internal power consumption and the output power consumption of the device.

The sum of the internal and the output powers is the total power consumption of the R4600. The system designer must calculate these two values for any mode to obtain the total power consumption of the CPU in that mode.

Note:

In this Application Note, the power examples assume a system with the following attributes:

- 5V power supply for typical measurement
- 5.25V power supply for max measurement
- MasterClock is 50MHz, the pipeline clock is 100MHz and the SysAd bus operates in the divide by 2 mode (50MHz).
- The examples use the values of Icc published in the March 1994 revision of the R4600 Data Sheet. For the most accurate results, the system designer should use the values published in the most recent revision of the data sheet.

Internal Power

The internal power is provided in the data sheet. It represents the power consumed by the internal logic of the device. However, it excludes the power consumed by the output buffers, since that is system dependent. Specifically, it depends on the capacitive loading of the output pins and the write pattern supported. The internal power is available in the data sheet and corresponds to the 0 pF loading condition on the output clocks and no SysAD activity. The internal power consumption (IP) is computed using the following equation:

IP = Icc * Voltage (Watts)

For the system example used in this Application Note with a supply voltage of 5V, the Standby internal power consumption is 920mWatts (175mA * 5.25V). Similarly, the Typical internal power consumption is 4375mWatts (875mA * 5V) while the Maximum internal power consumption is 6565mWatts (1250mA * 5.25V).

Output Power

The output power is the power consumed by the output buffers of the R4600. It is completely system dependent. It is a function of the capacitive loading the output buffer is driving and the frequency of the signal. System designers should use the guideline provided in this Application Note to compute the output power for their particular applications.

The output power per output pin is computed using the following equation:

$OP = C * V^2 * f$ (Watts)

OP is the Output_Power
C is the capacitive loading on the output pin.
V is the supply voltage
f is the frequency (number of low-to-high transitions / sec) of the output pin.

The total output power consumed is the sum of the output power for every individual output pin.

EXAMPLE OF OUTPUT POWER CALCULATIONS

The R4600 has two classes of output signals. The clock output signals and the bus signals (which include the SysAd and the output control signals). This example shows how to compute the output power for each class. Every calculation has to be done twice: to compute the Typical and the Max output power consumption. Remember that for the Typical power consumption, the power supply is assumed to be at its nominal value (5V in this case) and for the Max power consumption it is assumed to be at its max (5.25V in this case).

Clocks Output Power

The R4600 has 6 different output clocks: MasterOut, SyncOut, TClock[1:0] and RClock[1:0]. The output power calculation for each clock should be done separately.

SyncOut. Typically SyncOut is connected to SyncIn or to a single buffer to match the delay on the TClock and RClock. This is about 20pF of loading. The frequency of SyncOut is the same as MasterClock (50MHz). So the typical output power consumed by the SyncOut clock is:

$OP_{SyncOut} = C * V^2 * f$ (Watts)

$OP\_Typical_{SyncOut} = (20*10^{-12}) * (5)^2 * (50*10^6)$ Watts

$OP\_Typical_{SyncOut} = 25$ mWatts

The max output power consumed by the SyncOut clock is:

$OP\_Max_{SyncOut} = (20*10^{-12}) * (5.25)^2 * (50*10^6)$ Watts

$OP\_Max_{SyncOut} = 27.5$ mWatts

MasterOut. Typically MasterOut is connected to a couple of loads (mostly to the reset logic). This is about 30pF. The frequency is the same as the MasterClock (50MHz). So the typical output power consumed by the MasterOut clock is:

$OP\_Typical_{MasterOut} = (30*10^{-12}) * (5)^2 * (50*10^6)$ Watts

$OP\_Typical_{MasterOut} = 37.5$ mWatts

The max output power consumed by the MasterOut clock is:

$OP\_Max_{MasterOut} = (30*10^{-12}) * (5.25)^2 * (50*10^6)$ Watts

$OP\_Max_{MasterOut} = 41.3$ mWatts

TClock[1:0] and RClock[1:0]. Typically TClock[1:0] and RCLock[1:0] are connected to several loads; for this example assume that they add up to about 50pF. The frequency of TClock[1:0] and RClock[1:0] ($f_{TRClock}$) depends on the bus_clock_divisor which is selected at boot time (from 2 to 8). It is calculated using the following equation:

$$f_{TRClock} = \frac{MasterClock * 2}{bus\_clock\_divisor} \text{ (MHz)}$$

The bus_clock_divisor in this example is set to 2. The $f_{TRClock}$ is then:

$$f_{TRClock} = \frac{(50*10^6) * 2}{2} \text{ MHz} = 50 \text{ MHz}$$

There are 4 clocks (two TClocks and two RClocks). So the typical output power consumed by the TClocks and RClocks in this example is:

$OP\_Typical_{TRClock} = 4 * (50*10^{-12}) * (5)^2 * (50*10^6)$ Watts

$OP\_Typical_{TRClock} = 250$ mWatts

The max output power consumed by the TClocks and the RClocks is:

$$OP\_Max_{TRClock} = 4 * (50*10^{-12}) * (5.25)^2 * (50*10^6)$$

$$OP\_Max_{TRClock} = 275.5 \text{ mWatts}$$

The typical total output power consumed by the clocks is the sum of the typical output power consumed by the individual clocks:

$$OP\_Typical_{Clock} = OP\_Typical_{SyncOut} + OP\_Typical_{MasterOut} + OP\_Typical_{TRClock}$$

$$OP\_Typical_{Clock} = 25 + 37.5 + 250 = 312.5 \text{ mWatts}$$

Similarly, the max total output power consumed by the clocks is the sum of the max output power consumed by the individual clocks:

$$OP\_Max_{Clock} = OP\_Max_{SyncOut} + OP\_Max_{MasterOut} + OP\_Max_{TRClock}$$

$$OP\_Max_{Clock} = 27.5 + 41.3 + 275.5 = 344.3 \text{ mWatts}$$

Of course, the system designer should determine the power estimate for any given system, factoring in the loading, the clock frequency and the supply voltage.

Bus Output Power

The R4600 bus transactions consist of main memory accesses (read and write operations). The output power consumed by the bus signals differs from one transaction to the other. Read and block read transactions represent the best case since the R4600 consumes output power only during the address phase of the transaction. During the data phase, the system returns the data to the CPU and the R4600 doesn't consume much output power. The output power consumed in the read transactions can be obtained by computing the power consumed during the address phase of the bus. This case will not be demonstrated in this example; since in a typical system, the power contribution of the read transactions is negligible.

On the other hand, the write transactions tend to consume much more output power because the R4600 is continuously driving the bus with either the address or the data. The worst case output power consumption by the bus unit is when the R4600 does a stream of uncached write transactions or write -through stores when the address is the complement of the data. It also assumes that all the SysAd and the SysCmd bits need to toggle. This case represents the max output power consumed by the bus. The example in this Application Note will concentrate on this situation.

Further, there are two major cases to consider when calculating the bus max output power consumption during write transactions. The first is the R4xxx compatible bus write protocol and the second is the write-reissue or the pipelined write bus protocols.

Before starting the calculations of the bus output power consumption during the write transactions, a generic formula to compute the average SysAd_Data_Frequency (f_{SysAd_Data}) is needed. This is the frequency that is used in the Output_Power equation. The average f_{SysAd_Data} is computed as follows:

$$f_{SysAd\_Data} = \frac{1}{2} * \frac{MasterClock * 2}{bus\_clock\_divisor} * \frac{n}{m} \text{ (MHz)}$$

MasterClock * 2 is the frequency of the output bus_clock_divisor clocks (TClock and RClock).

n is the number of transitions on the SysAd bus

m is the total number of bus clock cycles to complete a write transaction

1/2 The output clock frequency is divided in half because the max transitions on the SysAd bus are at half the output clock frequency.

R4xxx compatible write protocol. In this mode, the R4600 performs an uncached write transaction every 4 SysAD cycles (the actual pattern is ADxx). The number of transitions "n" is 2 and the total number of clock cycles "m" is 4 in this case. The bus frequency in the case of a bus_clock_divisor equals to 2 is:

$$f_{Compatible} = \frac{1}{2} * \frac{(50*10^6) * 2}{2} * \frac{2}{4} = 12.5 \text{ MHz}$$

There is a total of 81 output signals used during the write transactions (64 SysAD outputs, 8 SysADC outputs and 9 SysCmd outputs). There is also ~ValidOut which should toggle once in this case. However, for simplicity reasons it will not be part of the calculations. On the other hand, all the SysCmd bits are assumed to toggle which might not be the case. Assume a 50pF load on each. Then the max output power consumed by the bus in the R4xxx compatible mode is:

$$OP\_Max_{BusCompatible} = 81*(50*10^{-12})*(5.25)^2*(12.5*10^6) \text{ Watts}$$

$$OP\_Max_{BusCompatible} = 1395.5 \text{ mWatts}$$

Write reissue and pipelined write protocols. In these modes, the R4600 performs an uncached write transaction every 2 SysAD cycles (the actual pattern is AD). The number of transitions "n" is 2 and the total number of clock cycles "m" is 2 in this case. The bus frequency in the case of a bus_clock_divisor equals to 2 is:

$$f_{Pipelined} = \frac{1}{2} * \frac{(50*10^6) * 2}{2} * \frac{2}{2} = 25 \text{ MHz}$$

There is a total of 81 output signals used during the write transactions (64 SysAD outputs, 8 SysADC outputs and 9 SysCmd outputs). ~ValidOut will not toggle in this mode and

is not counted. Assume a 50 pF load on each. Then the max output power consumed by the bus in the write reissue or the pipelined write modes is:

$$OP\_Max_{BusPipelined} = 81 * (50*10^{-12}) * (5.25)^2 * (25*10^6) \text{ Watts}$$

$$OP\_Max_{BusPipelined} = 2790 \text{mWatts}$$

Typical-case bus output power. In a more typical system, the bus output power consumption of the R4600 is much less than the worst case numbers. In normal operation, the R4600 performs primarily block write transactions. In this case, the non-block write transactions are a small percentage of the total bus activity and the output power consumed during non-block write transactions is irrelevant. The block write transactions represent the typical output power consumed by the bus.

The statistics from standard benchmarks indicate that a typical application, executing from the internal caches, requires the R4600 to perform a block write transaction every "I" processor cycles on the average. A processor cycle is executing at the speed of the internal pipeline (MasterClock * 2). The value of "I" is independent from the write back pattern in the block write transaction (because it is always 5 transitions no matter what). The total number of clock cycles to complete the transaction "m" is then actually equals to "I" divided by the bus_clock_divisor as stated in the following equation:

$$m = \frac{I}{bus\_clock\_divisor} \text{ (clock cycles)}$$

The number of transitions in a block write transaction "n" is 5 (address and 4 double words of data). In this case the frequency of the bus ($f_{BusTypical}$) in the case of a bus_clock_divisor equals to 2 and a value of "I" equals to 200 (for example) is:

$$f_{BusTypical} = \frac{1}{2} * \frac{(50*10^6) * 2}{2} * \frac{5}{(200/2)} = 1.25 \text{ MHz}$$

There is a total of 81 output signals used during the write transactions (64 SysAD outputs, 8 SysADC outputs and 9 SysCmd outputs). There is also ~ValidOut which might toggle or not depending on the write-back pattern selected. In this case, with a write back pattern of ADDDD, the ~ValidOut signal doesn't toggle and will not be counted. Assume a 50 pF load on each. Then the typical output power consumed by the bus during a typical write back mode (when all outputs switch) is:

$$OP\_Typical_{BusTypical} = 81 * (50*10^{-12}) * (5)^2 * (1.25*10^6) \text{ Watts}$$

$$OP\_Typical_{BusTypical} = 126.5 \text{ mWatts}$$

The typical total output power consumed by the R4600 is the sum of the clocks typical output power and the bus typical output power. Similarly, the max output power consumed is the sum of the max clock output power and the max bus output power consumptions. The max output power consumption depends on the bus write protocol (R4xxx compatible or write reissue or pipelined write transactions). The typical output power consumption doesn't depend on the write protocol or the write back pattern.

TOTAL POWER CONSUMPTION

The total power consumption of the R4600 is then the sum of the internal power and the output power consumptions. It depends on the system design in terms of the loading on the bus as well as on the application S/W and the mode of operation of the R4600. The system designers should compute the output power consumption for their particular application to obtain the total power consumption of the device. The Total Power (TP) is expressed in the following equation:

$$TP = IP + OP \text{ (Watts)}$$

To finish the example started in this Application Note, the total typical power consumed by the R4600 in the system described is:

$$TP\_Typical_{R4600} = IP\_Typical_{R4600} + OP\_Typical_{R4600} \text{ Watts}$$

$$TP\_Typical_{R4600} = 4375 + [126.5 + (25 + 37.5 + 250)] \text{ Watts}$$

$$TP\_Typical_{R4600} = 4814 \text{ mWatts}$$

Similarly, the total max power consumed by the R4600 in the system described using the R4xxx compatible write mode is:

$$TP\_Max_{R4600} = IP\_Max_{R4600} + OP\_Max_{R4600} \text{ Watts}$$

$$TP\_Max_{R4600} = 6565 + [1395.9 + (27.5 + 41.3 + 275.5)] \text{ Watts}$$

$$TP\_Typical_{R4600} \approx 8305 \text{ mWatts}$$

CORRELATION WITH THE DATA SHEET

The power consumption of the R4600 is listed in the data sheet in the Icc Table. There are several columns in the table that correspond to the internal pipeline frequency and to the external bus frequency (100/50MHz column as an example). For every column, the typical and the max current consumption is listed for the Standby mode and for the Active modes. The 0pF loading with no SysAd activity condition represents the internal power consumption of the device.

The 50pF loading condition in the Standby mode corresponds to the max power consumed in this mode with the active clocks loaded with 50pF. Remember that in this mode only a few external clock signals are active.

The 50pF loading condition for the Active mode for both the Typical and the Max case is the total power consumption of the device. These values are derived using the equations intro-

duced in this Application Note. However, the loading on the bus is different. The clocks are assumed to be driving a load of 50pF. This is substantially more than the 20 or 30pF assumed for SyncOut and MasterOut in this Application Note. Similarly, the R4xxx compatible mode and the pipelined or write reissue mode assume the number of output signals toggling to be 81. The ~ValidOut signal is not part of the calculations. The loading on every output pin is assumed to be 50pF. There is also a small added guard band in the published numbers.

System designers can use the values provided in the data sheet as a max upper limit for the possible power consumption of the R4600 under the mentioned conditions. However, it is always recommended for the system designers to compute the exact power consumption of their particular application. The values they obtain will be much more accurate than the upper limit presented in the data sheet, which reflect the worst case assumptions used during device testing.

CONCLUSION

The R4600 is designed from the ground-up to consume as little power as possible while achieving very high performance. It incorporates advanced power management techniques to turn off the power from the unused units of the device. This reduces the typical power consumed compared to other microprocessor in its class. On the other hand, the R4600 doesn't sacrifice performance for the reduction in the power consumed. Several systems have shown the R4600 to outperform the R4400PC by at least 30% at a given frequency.

This Application Note explains how to compute the output power consumed for every situation and how to derive the total power of the CPU under different system conditions. The system designer should use it as a reference and a guideline in computing the power consumption for their particular application. In addition, the system designer can use this information to make power consumption trade-offs during system design.

Integrated Device Technology, Inc.

VISIBLE DIFFERENCES BETWEEN THE R4650 AND THE R4600/R4700 ORION FAMILY MEMBERS

APPLICATION NOTE AN-135

By: Ketan Deshpande

INTRODUCTION:

The IDTR4650 is a low cost member of the IDTR4600 (Orion) family, targeted towards a variety of embedded applications. R4600 features not required in many embedded systems have been removed in the R4650 to lower device cost; others have been added to better suit the processor for its target applications. Given these changes in architecture, software designed to run on the Orion may need to be slightly modified to be able to take full advantage of the features of the R4650.

This Application note discusses the software visible changes integrated within the R4650; this information is required when porting existing low-level software (e.g. compilers, debuggers and other assembly language programs) from the R4600 to the R4650.

Architectural Differences:

While a complete discussion of the architectural differences between the R4650 and the Orion is beyond the scope of this note, the relevant differences will be enumerated and software issues discussed. Some system control registers have been deleted, some new ones have been added, and some have been modified. Also, some exceptions are no longer generated, and some new exceptions can be generated.

1. Integer Execution Unit:

The R4650 uses the same ALU as the Orion, with a few modifications:

a) Faster MULT/DMULT instructions.

As a result of the faster MULT / DMULT instructions, assemblers or assembly language programmers need not wait as many cycles as earlier to retrieve the result from the HI/LO registers.

For MULT instructions (32x32->64 bits) the R4650 detects the actual size of the operands; the execution time of the multiply is thus determined by the actual number of significant bits in the operands. For 16-bit operands, the time taken to perform a MULT instruction is 2 pipeline cycles (PCycles) and for 32-bit operands, the time is 3 PCycles.

The time to perform a DMULT operation(64x64->128 bits) is 5 cycles, irrespective of the size of the operands.

b) New instructions: MUL and MAD.

The MUL instruction can be used to multiply two CPU general purpose registers (GPRs) and store the result in another GPR (32x32->64-bits), bypassing the HI/LO pair, and eliminating the MFHI/MFLO instructions.

The MAD instruction multiplies two (32-bit) GPRs and adds the product to the contents of the HI/LO registers, storing the result in the HI/LO pair.

MUL and MAD are defined only for 32-bit numbers; there are no DMUL / DMAD instructions.

2. Control Processor 0 (CP0):

CP0 has been greatly changed from the original R4600 Orion. Only two modes: user and kernel are supported (selected by setting the UM bit in the STATUS register). All addresses (virtual and physical) are 32 bits. There is no 64-bit virtual address mode. All CP0 registers are now 32-bit, and the DMTC0/DMFC0 instructions are no longer valid. However, these instructions will not generate a trap.

a) PRId Register:

If the same software will be used to support the Orion and the R4650, CPU-specific code can be separated on the basis of the Implementation field of the PRId register in CP0, which is 0x22 for the R4650, and 0x20 for the Orion.

b)STATUS Register:

The STATUS register has a different format in the R4650.

i) It has a bit to lock set A of the I-Cache (the IL bit, bit 23), and one to lock set A of the Dcache (the DL bit, bit 24). Critical sections of the code / data may thus be locked into the cache for fast access. When locked, this set will not be chosen for line refill. However, a line in a locked set will still be chosen for refill if that line is invalid. Thus locked sets may be flushed without having to unlock them first. It takes 5 instructions after setting the IL bit for refills to be disabled, and 3 instructions after setting the DL bit.

ii) The FR bit (bit 26) can be set to select 16 or 32 32-bit floating point registers.

c) CAUSE Register:

The CAUSE register has a slightly different format. It has two new bits that denote whether the exception was due to IWatch or DWatch (bits 24 & 25 respectively, discussed below) and one bit (IV bit, bit 23) to force interrupts to use a different exception vector offset. On reset, Cause.IV is cleared; thus exceptions and interrupts use the same exception vector offset (0x180). When Cause.IV is set, interrupts use a new exception vector offset (0x200). This can be used for faster decoding of interrupts. This new exception vector did not exist in the R4600 Orion; thus, the use of a dedicated interrupt vector is an option, not a mandate, in the R4650. For systems whose performance is highly dependent on interrupts, additional software modifications may be desirable, since there may be code at that location that now needs to be moved, as well as moving the interrupt management code to that location.

d) TLB:
The R4650 does not include the R4600 Orion Memory Management Unit (MMU). The CP0 TLB registers 0-6, 10 and 20 have been removed. The instructions TLBR, TLBWI, TLBWR are no longer defined, but will not generate a trap. TLB exceptions like TLBMiss / XTLBMiss will never be generated. The exception vector offsets 0x000 and 0x080 are no longer used.

The R4650 performs virtual address translation based on Base/Bound register pairs. There are two sets of these pairs: One for Instruction and one for Data. In user mode, when an address is generated, it is compared with the base / bound register pair. If the address is "out of bounds", an exception is generated, with the appropriate ExcCode bits set in the Cause register (0x2 for Instruction, 0x3 for Data). An MTC0 instruction which changes any base / bound register must be done in unmapped space and mapped space cannot be entered for 5 instructions following a change to these registers. In kernel mode, all addresses undergo a fixed virtual to physical address translation, bypassing the base/bound pairs. In kernel mode, the base/bound exception will never be generated.

e) Cache Algorithm Register:
The LLAddr register in the Orion has been replaced with the CAlg register, which defines the Cache Algorithm for each 512 MB region of the virtual address space. On reset, it gets initialized to 0x22233333, which is consistent with the Orion's interpretation of the K0 bits in its own CONFIG register. An MTC0 instruction should not change the field corresponding to the address space currently active. Doing so will cause undefined behavior.

f) Watch Registers:
Two new registers, IWatch and DWatch, greatly facilitate software debug. By setting the contents of the registers to the desired watch point and enabling the Watch Exception, an exception handler can be called every time the watch point is hit. The exception generated is at the general exception vector, with ExcCode = 0x23 in the Cause register. The IW/DW bits in the Cause register are set to denote whether the exception was caused by a Data Watch point or Instruction Watch point. The actual exception will be generated whenever both the ERL & EXL bits in the STATUS register are cleared. When DWatch is enabled, the two instructions immediately following may not be checked for match with the watch value. When IWatch is enabled, the 5 instructions following may not be checked for match with the watch value.

g) CONFIG Register:
The CONFIG register in the R4650 is read-only. The format has been modified: the IC & DC bits are both now 001, denoting the 8KB:8KB cache sizes. The K0 field has been deleted since this function has been expanded and is now performed by the CAlg register.

h) Other Registers:
The BadVAddr, EPC & ErrorEPC registers in the Orion were 64 bits; in the R4650 they are 32 bits wide.

3. Co-Processor 1 (CP1):

This is the Floating point coprocessor on board the R4650. The single biggest departure from the Orion is that the R4650 supports single precision operations only. The R4650 does not support double precision operations, which could be performed by an emulation library, if required. CP1 has a set of general purpose registers (FGRs) that are 32-bit wide, and can be accessed as a group of 16 or a group of 32 registers, by setting the FR bit in the CP0 STATUS register to 0 or 1, respectively. If STATUS.FR = 0, only even numbered FGRs can be accessed, and accessing an odd numbered register generates a trap. Any double precision operation in CP1 causes a trap to occur; thus a trap-based library could be written to emulate double-precision operations. DMFC1/ DMTC1 instructions will generate a trap.

There are two floating-point execution units in the R4650: one multiply unit and one unit for add/convert/divide/SQRT. As a result, multiplies and add/subtracts can be overlapped.

CONCLUSION:

This Application Note discussed the issues involved in porting assembly code from the R4600 Orion to the R4650. Some relevant architectural differences were noted, with implications for software modification.

Integrated Device Technology, Inc.

HIGH END/ LOW POWER R4650 WITH DSP CAPABILITIES

APPLICATION NOTE AN-137

By Robert Napaa

INTRODUCTION

In the next few years, the market share of the portable systems is expected to increase steadily. Furthermore, users will demand that the performance of these systems matches the performance of desktop systems. The portable systems should be able to manipulate data, voice and video in a multimedia environment exactly like their desktop counter parts. Unlike the desktop systems, portable systems face another set of challenges. First, the power consumption of these systems is limited by the battery life. This implies that the components used should consume as little power as possible and have power management capability to reduce the consumed power even more when the system is idle. Secondly, the portable systems are becoming smaller and smaller, lighter and lighter. This implies that more functionality is implemented using a fewer number of devices. The trend is to implement as much functionality in software as possible to reduce the need for dedicate hardware solutions.

MULTIMEDIA

The definition of multimedia is somewhat vague. Multimedia refers to systems capable of manipulating digital voice, digital images and digital data such as speech, video and file transfers. Multimedia systems must be able to manipulate these applications in either a real time environment or in a play back environment. In the real time environment such as cellular telephony, the max delay can be 250 - 350 msec, because delays longer than that will result in a poor quality of sound. The voice will be chopped and hard to understand. Similar constraints apply to real time digital video.

The above applications involve large amounts of data to be manipulated and stored. Usually the data is compressed to minimize the storage requirements and to increase the effective bandwidth of the systems. Similarly, the data could be encrypted to preserve the content of the information. All these different techniques are based on various Digital Signal Processing (DSP) algorithms. As an example, video images are compressed using different algorithms. Motion-JPEG, MPEG1 and MPEG2 are used for motion video, while JPEG is used for still images. Similarly several algorithms have been developed for speech compression such as TrueSpeech™. All these techniques require a very fast real time DSP engine.

Along with the DSP capability, multimedia systems are general purpose systems that implement other tasks as well, such as interfacing to memory, storage devices or other types of I/O devices. These tasks require the use of a general purpose microprocessor tailored more towards these usages.

TRADITIONAL SOLUTIONS

Traditional implementations for portable systems separate the general systems functions from the specific functions such as DSP or graphics. This separation is accomplished both on the hardware level and on the software level. On the hardware level, two or more types of different compute engines are used. At the center of the design is a general purpose microprocessor or microcontroller responsible for various system tasks and overall system management. Other dedicated hardware modules such as DSP microprocessors, graphic accelerators and custom ASICs are used in the system. Each of these modules serves a particular function.

On the software level the same separation takes place. The general purpose tasks are separated from the specific tasks. Procedure calls link various tasks together. In most instances, different operating systems (OS) are executing in parallel on different microprocessors in the system. As an example, a portable system may be implemented using a real time kernel for general purpose and system administration while a DSP specific-OS is used for the DSP tasks.

These traditional solutions are not well positioned to meet the challenges of the future. Specifically, the power consumption of these systems is not in line with the requirements of true portable systems. Similarly, the use of multiple devices places constraints on the form factor, the development time and the resulting system.

The trend is to merge more and more hardware functionality into a smaller number of devices. These devices perform several independent tasks that once required separate hardware modules. In addition, there is more emphasis on a software approach. Several independent software modules are combined together to execute on a single device. The software solutions are much more flexible than the hardware ones. Applications can be easily added, modified and customized at will without the need for a complete redesign of a hardware module. To be as efficient and as fast as the hardware modules they are replacing, these combined software modules require extensive compute power.

THE IDT ORION™ R4650

The IDT Orion R4650 is the latest member of the RISController™ family from IDT. It is a derivative of the IDT Orion R4600 and is based on the MIPS architecture. The Orion R4650 is a highly integrated microprocessor specially designed for the embedded market. It includes 8 KBytes of Instruction Cache and 8 KBytes of Data Cache, both of which are two-way set associative. The Orion R4650 executes at speeds up to 133 MHz. The internal core of the R4650 is a full 64-bit implementation of the MIPS III Instruction Set Architec-

ture (ISA) with 32 internal general purpose registers. It has a built-in floating-point accelerator unit. The raw performance of the R4650 is about 175 Dhrystone MIPS at 133 MHz. With these capabilities, the R4650 is an excellent general purpose microprocessor.

Most of the DSP algorithms rely heavily on a fast Multiply_and_Accumulate operation to perform effectively. To address the needs for multimedia applications, the Orion R4650 has a dedicated unit to perform integer Multiply_And_Accumulate (MAC) operations. This unit performs a multiply and add instruction every two clock cycles.

The Orion R4650 is also designed for low power systems. It consume less than 1.6 watts peak at 100 MHz, even less power at lower frequencies. It also incorporates active power management mode to further reduce the consumed power. This mode is dynamically invoked through the software. Figure 1 illustrates the simplified block diagram of the R4650.

Thus, the Orion R4650 offers the best of both worlds. It is a powerful general purpose compute engine for overall control and management tasks. It also executes DSP algorithms effectively, reducing the need for a dedicated DSP microprocessor. Its power consumption is very limited and can be dynamically adapted to the portable applications.

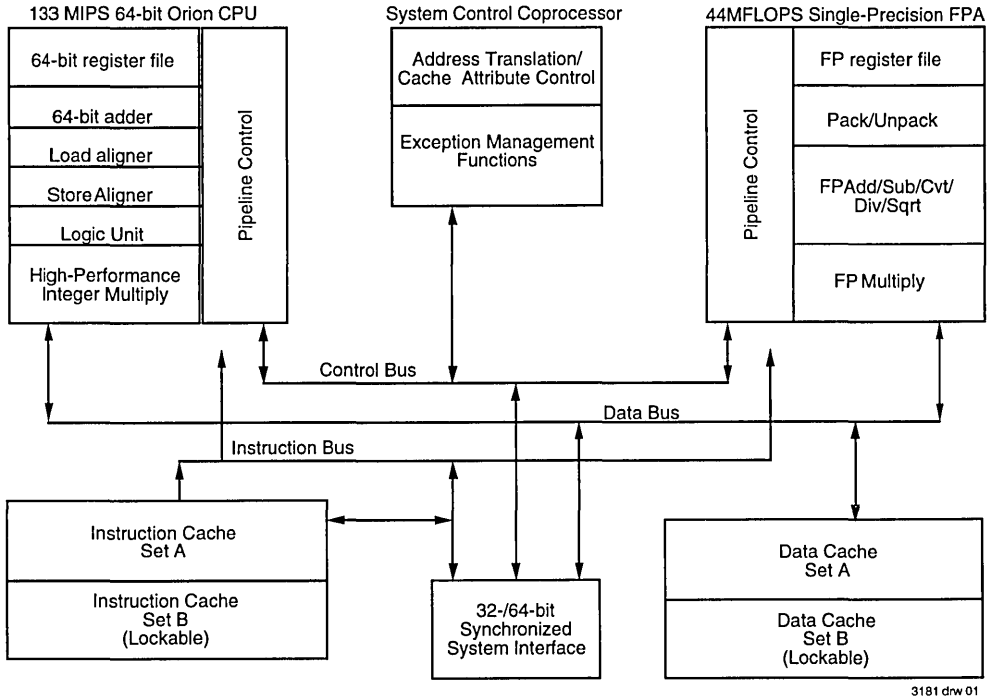


Figure 1. R4650 Block Diagram

SOFTWARE CONTROL

The R4650 is a true RISC compute engine, where the software has control over most functionality of the device. The software can manipulate the internal instruction and data cache to optimize the performance of the system. By using the "CACHE" instructions, the software can control the contents of any cache line. This fine control over the contents of the caches enables the OS to ensure that the data is always available for the different tasks it is scheduling.

The contents of the caches can also be locked. This means that a particular section of the instruction cache will never be replaced. This ensures that a time sensitive routine such as the interrupt service routine or a dedicated task such as a DSP algorithm is always in the cache. This minimizes the time interval between procedure calls. Similarly, a section of the data cache also can be locked. This ensures that the data is available for real time DSP algorithm for example. This minimizes the need to access the main memory and thus makes the response of the system more predictable, since the instruction and the data are local to the internal caches.

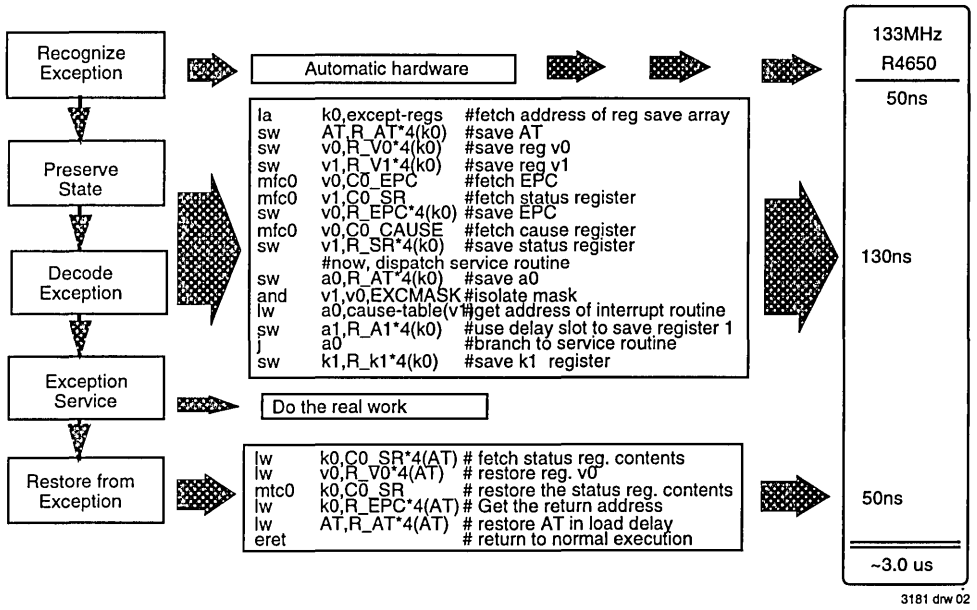


Figure 2. R4650 Interrupt Response Time
Real-Time Interrupt Response

To combine several independent software modules, such as a general purpose OS and a DSP algorithm, onto a single execution engine requires extensive use of interrupts. Usually, the task swapping is triggered by an external event. The interrupt driven approach is much more efficient and dynamic than a polled system. For real time applications, such as multimedia with large amounts of data to be serviced at high bandwidth, polling might not even be an option.

Usually an interrupt is asserted to request the R4650 to swap the tasks. To meet the real time requirements of the application or external event the task swapping must be accomplished as fast as possible.

The R4650 at 133 MHz can respond to interrupts in less than 250 nsec. This time includes recognizing the exception, preserving the state, decoding the exception and restoring the state at the end of the exception. Sample code to accomplish these steps is illustrated in Figure 2.

POWER CONSUMPTION

The components used in a portable system must consume as little power as possible. The R4650 is designed with this goal in mind. It is available in both 3.3V and 5V versions. The 3.3V has a peak power consumption of about 1.6 watts at 100 MHz. Furthermore, the R4650 uses an advanced power management scheme to further reduce the average power consumed. In this mode, the unused sections of the device are powered down. This mode is entered automatically when the internal logic determines that are no activity involving some section of the device. Thus average active power consumption in reduced to about 1 watt.

Finally, the R4650 provides a "Stand-By" mode, which is invoked by the software. In this mode, all of the internal clocks and the pipeline are frozen and the bus activity is stoped. This mode reduces the consumed power to less than 200 mwatts.

The OS can take advantage of all these power saving features on the R4650 by entering the "Stand-By" mode, when there is no system activity to reduce the average power consumption. If on the average, a portable system is active 25% of the time and idle the remaining 75%. The average power consumed by the R4650 will be in the order of 400 mwatts. It is important to note that the R4650 replaces several dedicated hardware modules in the system. This means that the average power consumed is substantially less than the traditional solutions. In addition, I/O power is alos reduced because the interface to the system is stopped.

DSP CAPABILITIES

The DSP algorithms are designed to manipulate large amounts of data effectively. At the heart of any DSP algorithm is a Multiply_And_Accumulate instruction. The R4650 is designed to execute DSP algorithms efficiently. It has a dedicated integer Multiply_And_Accumulate unit that executes at 133 MHz. A new multiply-accumulate instruction can be started every two cycles. As a result, the R4650 can perform 66.7 M multiply-accumulate instructions per sec. This integer DSP performance of the R4650 exceeds the performance of any other DSP microprocessor available on the market today. Table 1 illustrates the peak integer DSP performance of several architectures.

This type of DSP performance allows the R4650 to imple-

ment all the major DSP algorithms effectively. As an example the speech compression algorithm TrueSpeech™ from the DSP Group requires about 10 MIPS or less than 8% of the R4650 compute power to execute.

TABLE 1. COMPARISON OF VARIOUS DSP ARCHITECTURES

PRODUCT	FIXED POINT MACs (In Millions)
TI - TMS320C25	12.50
TI - TMS320C50	40
ATT - DSP16	54
MOT - 56K	40
IDT - R4650	66.7

3181 tbl 01

This example illustrates that the R4650 can mix general purpose tasks along with dedicated DSP algorithms in an efficient way. This powerful DSP engine reduces the need for dedicated external DSP microprocessors.

CONCLUSION

The R4650 is a general purpose microprocessor geared towards the portable applications. It implements a fast multiply-accumulate unit to speed up the different DSP algorithms. It can mix general purpose control tasks with DSP specific applications in an efficient way. These capabilities reduce the need for external dedicated DSP hardware modules. These features, combined with the average low power consumption, makes it ideal for portable applications.

Integrated Device Technology, Inc.

ADAPTING AN R4600 DESIGN TO THE R4650

APPLICATION NOTE AN-139

by Robert Napaa

INTRODUCTION

The IDT R4650 RISC microprocessor is the third generation 64-bit architecture targeted for a variety of performance-hungry embedded applications. It is the second derivative of the Orion family (also referred to as the R4600). The R4650 removes some of the functional units not frequently used in embedded applications, such as double precision floating point arithmetic and a TLB. The R4650 adds new features that are in line with its target applications, such as a dedicated integer DSP core, cache locking, improved real-time support and an optional 32-bit bus interface.

The R4650 is bus- and upwardly software compatible with the Orion family. It maintains the same bus protocol as the R4600 in both the 64-bit and the 32-bit external bus options. The external bus protocol refers to the handshaking between the CPU and the external logic as well as the timing for the various bus transactions. This insures that ASICs and system logic designed to interface to the R4600 will work with the R4650 without modifications. However, the external clock structure of the R4650 is different from that of the R4600 which provides greater flexibility to the system designer.

This Application Note explains: 1) the differences the internal and external clock distribution tree of the R4600 and the R4650, and 2) how to convert the R4650 clocks into R4600 compatible clocks to interface to existing ASICs and external logic.

DIFFERENT CLOCK STRUCTURE

The R4600 and the R4650 have different input and output clock structures but maintaining the same bus protocol.

R4600 Clocks

The R4600 implements the same clock structure as the first generation 64-bit devices such as the R4000 and the R4400. The R4600 uses a single input clock (MasterClock) that is doubled internally by one PLL to generate the pipeline clock (PClk). A second PLL doubles MasterClock, then divides it by a constant number (from 2 to 8 as programmed during reset) to generate the output clocks (RClock, TClock, MasterOut and SyncOut). These clocks are used by the system logic to interface to the R4600 during read and write operations. Figure 1 illustrates the architecture of the R4600 internal clock distribution tree. A more detailed explanation of the usage of these clocks is presented in the *"IDT79R4600 Hardware User's Manual"*.

R4650 Clocks

The R4650 uses a completely different architecture for the internal clock distribution tree. The R4650 uses a single input clock (MasterClock). MasterClock is multiplied internally using a single PLL by a constant number (from 2 to 8 as programmed during reset) to generate the pipeline clock

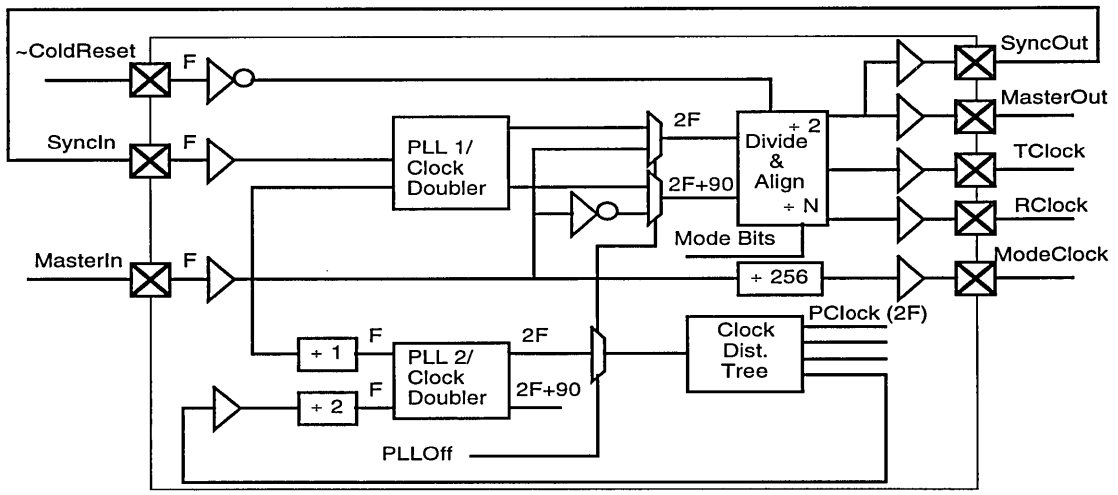


Figure 1. R4600 Clock Distribution Tree and PLLs

(PClk). The R4650 does not generate any output clock. The MasterClock should be used as the system control logic clock. The R4650 guarantees that the interface signals with the external system logic will be sampled using the rising edge of MasterClock. Figure 2 illustrates the internal clock tree of the R4650.

An advantage of the R4650 is that the MasterClock frequency may be kept small. Similarly, the absence of output clocks from the R4650 reduces the power consumption of the device. This architecture allows several systems to synchronize using a single input clock at any frequency without being locked by the clocks provided by the CPU. This is particularly advantageous for backplane applications where the input clock is provided from the backplane to several plugged-in cards.

GENERATING R4600-COMPATIBLE CLOCKS

Systems using the R4650 can reuse the logic and ASICs already developed to work with the R4600. This mechanism requires the generation of MasterOut, SyncOut, RClock and TClock, or alternatively, a subset of these according to the system requirements. The functionality of the different clocks is explained more in detail in the *"IDT79R4600 Hardware User's Manual"*.

The clock distribution tree has to be implemented at the input of the R4650. The R4600 clock generation is illustrated in Figure 3. In this case, a buffer is used to delay the input clock to the R4650. The output of the buffer is equivalent to TClock, MasterOut and SyncOut. The input of the buffer is equivalent to RClock. For a tight delay between RClock and TClock, it is better to use a buffer that has a very narrow window for the min and the max input to output delays. An example of such a clock buffer is the Motorola MC10H645 buffer, which guarantees a single nanosec difference between the min and the max delays.

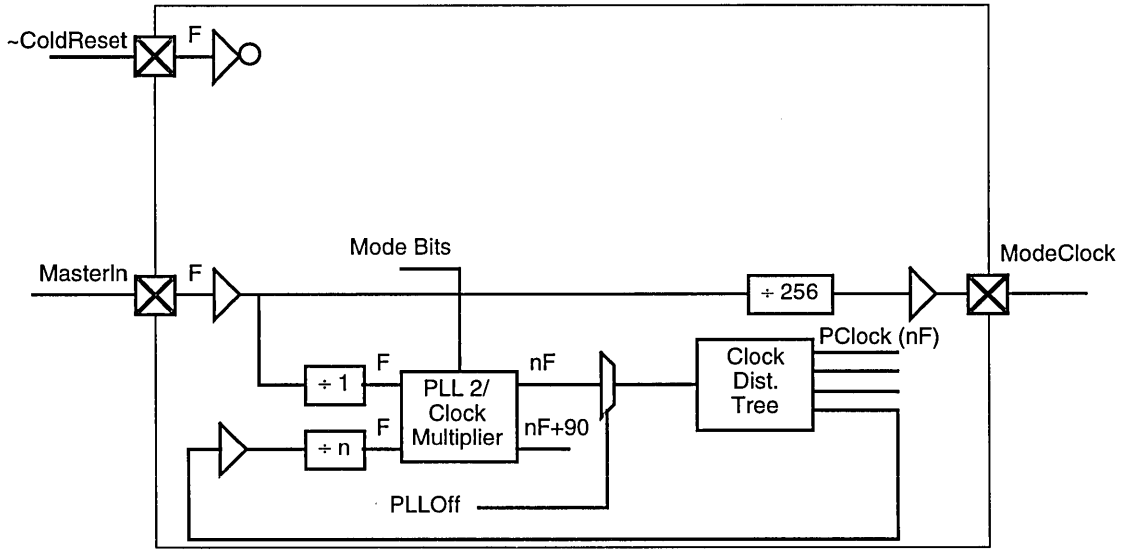


Figure 2. R4650 Clock Distribution Tree and PLL

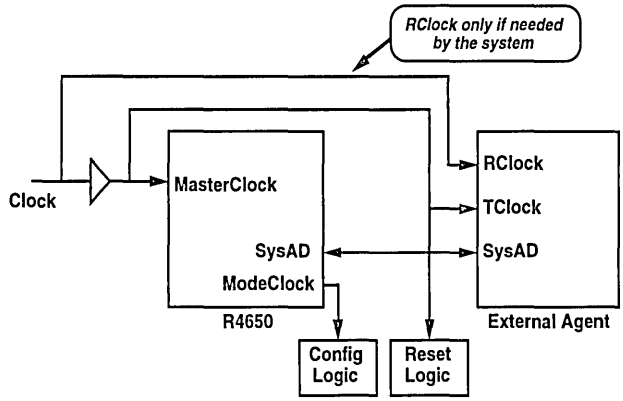


Figure 3. Generating R4600-Compatible Clocks

It is also important to note that in systems using only the R4650, the SyncOut to SyncIn path is irrelevant, since neither the R4650 nor the system logic use these clocks. The MasterOut could be relevant, depending on the system architecture.

DESIGNING A SYSTEM THAT SUPPORTS BOTH CPUs

It is possible to design a single system to support either the R4600 or the R4650 on a single PCB board. The same design allows using the R4600 for high performance applications, while using the R4650 to serve the medium performance segment of the market. This approach preserves the investment in the ASIC development, the system logic, the system

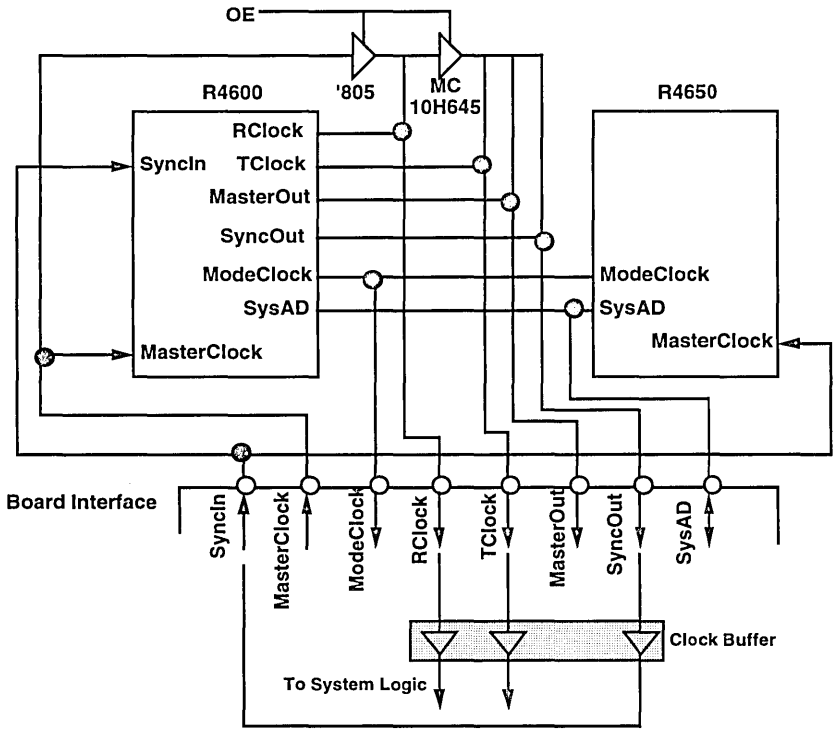


Figure 4. Single System With R4600 and R4650

software and so on. Figure 4 illustrates the block diagram for a system that can support both CPUs on the single PCB.

In this implementation, the clock from the external oscillator, MasterClock, is fed as the input clock to the R4600. It is also fed to the clock distribution circuitry that generates RClock, TClock, MasterOut and SyncOut to be used when the R4650 is used. It is important to note that only one CPU should be plugged-in at any onetime. The clock distribution circuitry is tri-statable when the R4600 is used, since it produces these clocks.

The SyncOut clock is routed on the PCB and returned as SyncIn. The SyncIn clock is fed to the R4600 to align the internal clocks used to sample the system interface, with the RClock and TClock seen by the system logic. In the case of the R4650 the SyncIn clock is used as the input MasterClock to the CPU. This ensures that the input clock to the R4650 is aligned to the system clocks (RClock and TClock) that are generated by the clock distribution circuitry.

THE 79S461

The 79S461 is a small module that supports both the R4600 and the R4650 on a single PCB. It plugs into the PGA socket of the R4600 on any design. It allows the system designer to evaluate the performance of either CPU in the system without modifications to the existing design. Figure 5 illustrates the block diagram of the 79S461. In addition, the schematics of the S461 board are attached to the end of this App Note to provide a better understanding in converting from one clock architecture to the other.

CONCLUSION

It is relatively easy to adapt a design that is based on the R4600 to support the R4650 on a single PCB. This approach offers a great flexibility in selecting the appropriate CPU for the level of performance needed without redesigning the system. The same design allows using the R4600 for high performance applications, while using the R4650 to serve the medium performance segment of the market.

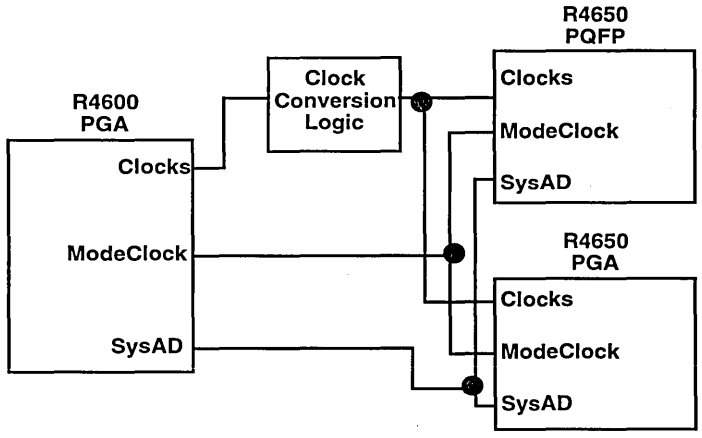
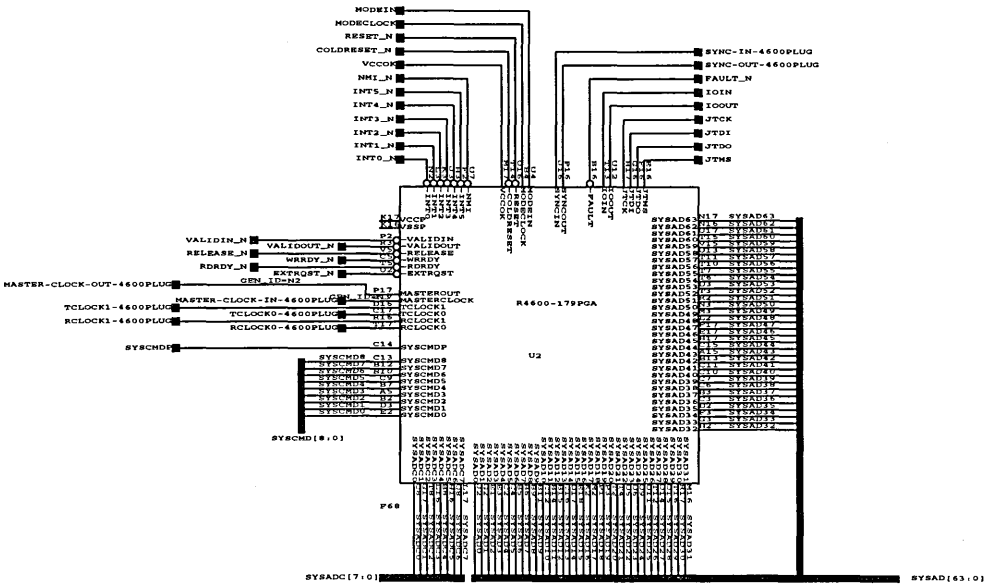
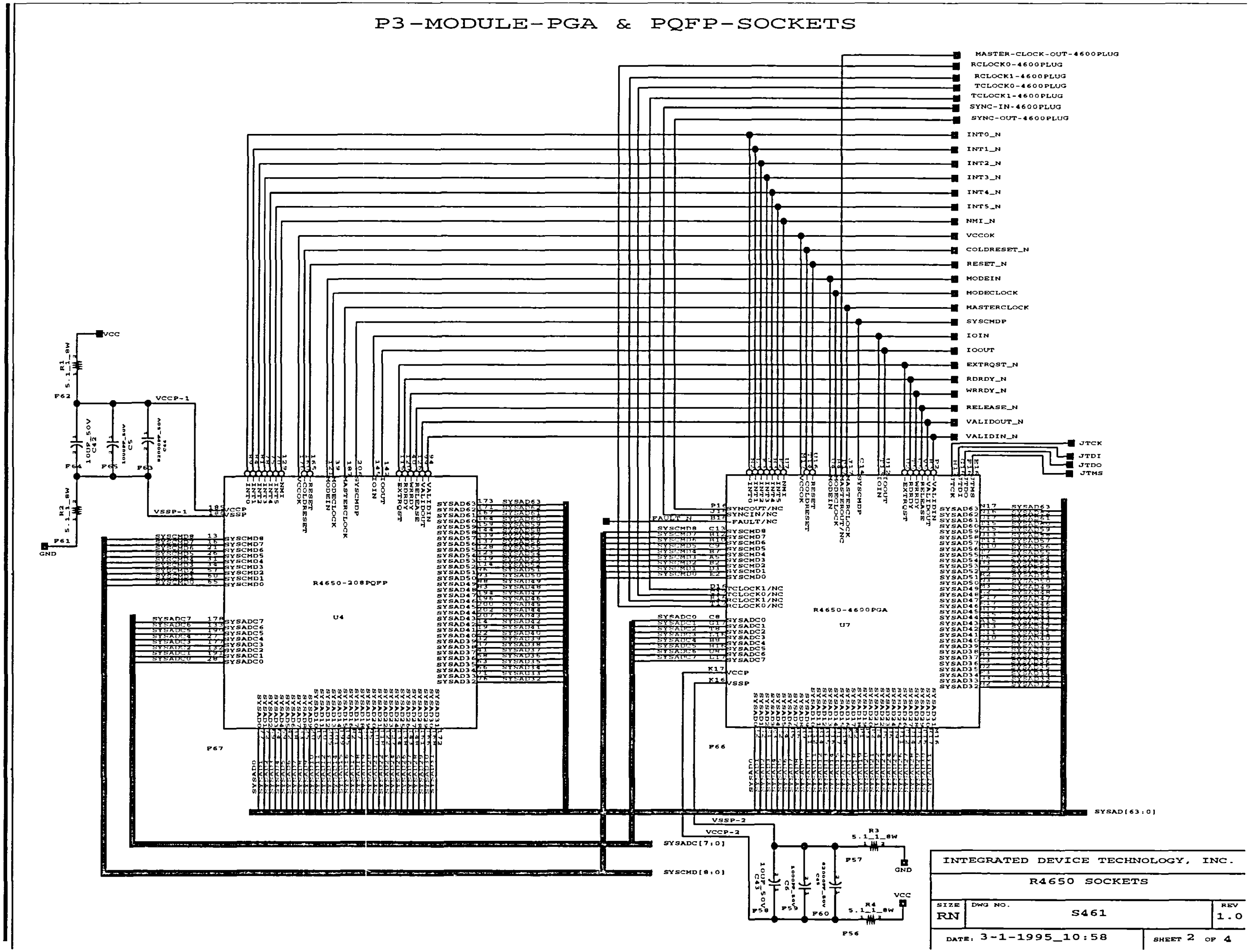


Figure 5. 79S461 Block Diagram

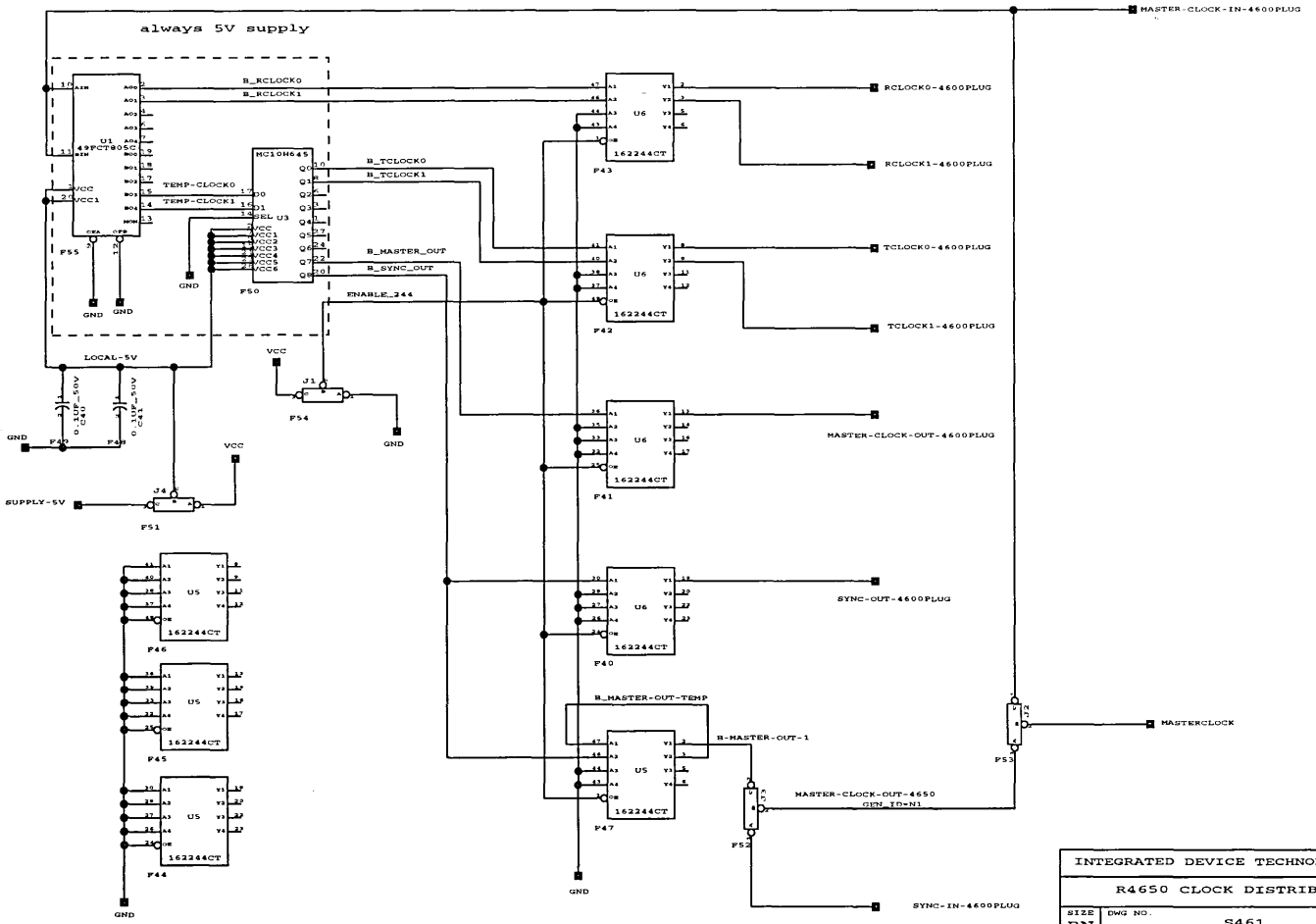
P3-MODULE PGA PLUG



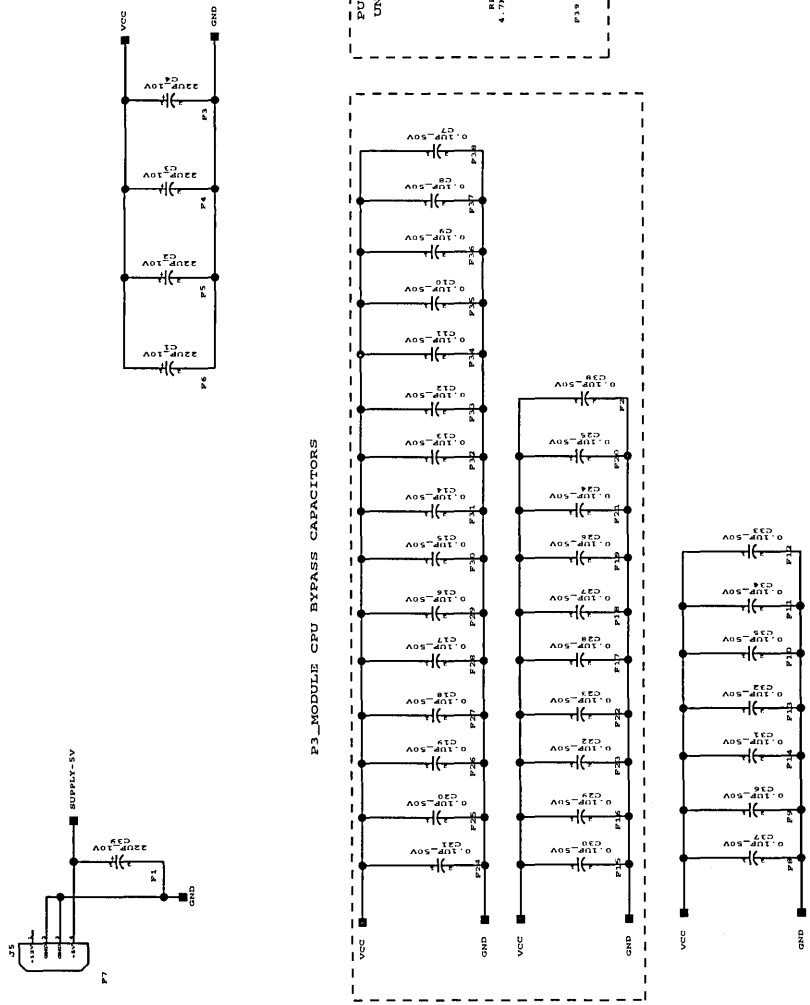
INTEGRATED DEVICE TECHNOLOGY, INC.		
179 PGA R4600 PLUG		
SIZE CR	DWG NO. S461	REV 1.0
DATE: 3-1-1995_10:57	SHEET 1 OF 4	


P3-MODULE-PGA & PQFP-SOCKETS
R4650-208PQFP
U4
R4650-4600PGA
U7
SYSAD[63:0]
SYSADC[7:0]
SYSCMD[8:0]
INTEGRATED DEVICE TECHNOLOGY, INC.
R4650 SOCKETS
SIZE RN
DWG NO. S461
REV 1.0
DATE: 3-1-1995_10:58
SHEET 2 OF 4

P3-MODULE-CLOCK-DISTRIBUTION & BUFFERING



INTEGRATED DEVICE TECHNOLOGY, INC.		
R4650 CLOCK DISTRIBUTION		
SIZE RN	DWG NO. S461	REV 1.0
DATE: 3-1-1995_10:59	SHEET 3 OF 4	


INTEGRATED DEVICE TECHNOLOGY, INC.
CAPACITORS
SIZE CR
DWG NO. S461
REV 1.0
DATE: 3-1-1995_10:42
SHEET 4 OF 4
PULLUPS FOR 4600-PLUG
UNUSED PINS
VCC
SYSCHDP
FAULT_N
IOIN
JTMS
JTDO
JTDI
JTCK
RP1
4.7K(8)
P39
P3_MODULE CPU BYPASS CAPACITORS
VCC
GND
0.1UF_50V
330UF_10V
SUPPLY-5V
P7
J5

Integrated Device Technology, Inc.

DESIGN OF A RISC-BASED PC

Conference Paper
CP-11

By Phil Bourekas, Integrated Device Technology, Inc.
and Blaise Fanning, Deskstation Technology, Inc.

INTRODUCTION

Modern personal computers can take advantage of the processing power inherent in today's high-performance microprocessors. The emergence of Windows™ NT as a 3rd generation operating system for PCs enables the power user to access a wide variety of sophisticated applications simultaneously, and provides a user-friendly windowing interface. This combination means that the modern PC needs to utilize the highest-performance processors available, and take advantage of modern memory system techniques, to offer the performance required for this software environment.

On the other hand, the market place desires that these PCs remain low cost. The market has built an entire infra-structure to support low-cost PCs, including system chip-sets, add-in cards, and peripherals.

This paper describes the implementation of a high-performance, low-cost RISC-based PC. The system is implemented using standard PC-style components and techniques, but uses the high-performance R4000PC RISC microprocessor to achieve ultra-high performance. The resulting system achieves the performance desirable in a Windows NT environment, while meeting the cost constraints of the PC marketplace.

PROJECT GOALS

The goals for the original DeskStation PC were to implement a high-performance EISA-based PC, at low system cost. Along with these primary goals, a few secondary goals helped to shape the actual implementation.

- Maintain PC flexibility. EISA allows a wide variety of add-in functions, including low-cost ISA cards through high-performance EISA master cards.
- Design upgradeability. The PC market place both requires and allows that designs be periodically modified to address different price-performance points. Thus, the initial implementation was designed to insure that these degrees of freedom were maintained.
- Ease of design. Again, given the rapid rate of advancement of the PC marketplace, it made sense to target an implementation that could rapidly be brought to market. In addition, ease of design typically reflects on system cost, as more complex designs typically require more expensive system logic to implement.

Thus, the implementation chose the following:

- Windows NT as the operating system. This brings PC flexibility and compatibility, while offering a robust software environment for sophisticated applications.
- The R4000 RISC microprocessor family. This 3rd generation RISC processor offers ultra-high performance, a painless upgrade path in the future, and compatibility with Windows NT. Figure 1 shows a block diagram of the R4400PC microprocessor.

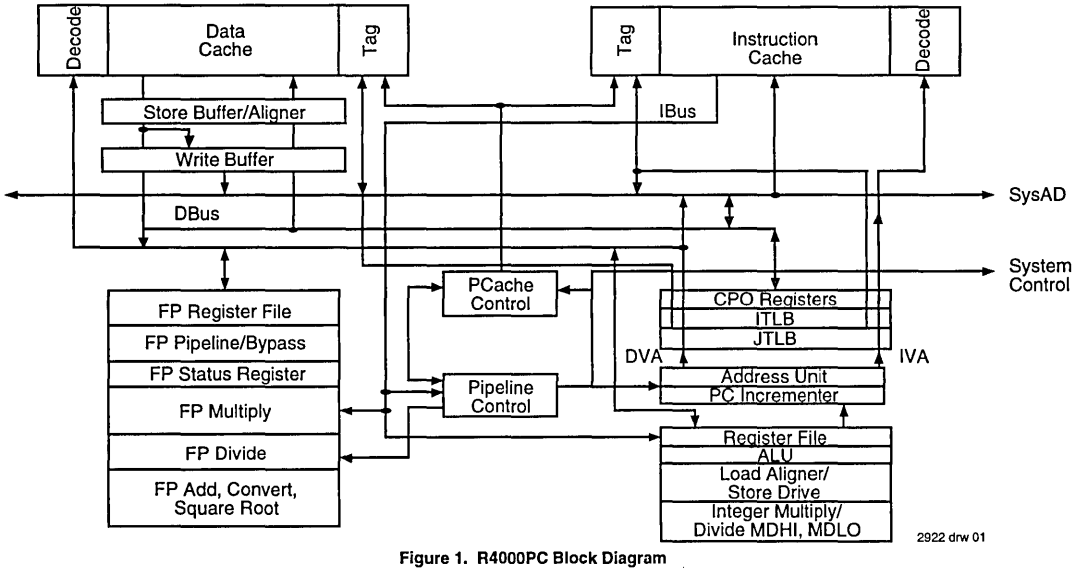


Figure 1. R4000PC Block Diagram

The R4000 features a high-speed pipeline (100 MHz or greater), while preserving the ability to keep the system bus at 50MHz or less. High-speed execution is supported by large on-chip caches: 16KB each of instruction and data cache for the R4400. Further, the processor is multiply sourced, and readily available.

Although the R4000 family offers a device with a dedicated secondary cache port (the R4000SC), this system is built around the lower cost R4000PC. The processor performance is aided by the use of a discretely-built secondary cache which resides on the main memory bus, analogous to the secondary caches constructed for i486™ processors.

- A traditional PC system architecture, to take advantage of the low-cost and ease of design infrastructure of the PC marketplace. Thus, the design targeted the use of a standard PC chip set, and standard PC peripherals and add-in cards.

  Since the system design target was for an EISA PC, the Opti EISA chipset was selected to implement the main memory and I/O systems. Thus, the primary design burden was to interface between the R4000PC/R4400PC, and the Opti EISA chip set.

INITIAL DESIGN

In order to minimize time-to-market, and to prove that the high-performance inherent in the R4000 architecture could be readily obtained from a PC system architecture, the initial design utilized discrete parts to interface between the R4000PC and the Opti PC chip set.

The design uses standard PALs, data buffers, and SRAMs to construct the processor secondary cache, and to provide the interface to the Opti chip set. Although the Opti chip set does feature secondary cache control for an i486 processor, the design chose to implement a higher bandwidth secondary cache for the R4000; thus, the secondary cache controller in the Opti chip set is not utilized. Instead, a discrete secondary cache controller, using PALs to control standard 32K x 8 SRAMs, is implemented.

The design was partitioned in such a way to to be easily modified to different chip sets, cache algorithms, and to simplify debug. In addition, the overall design can be readily cost reduced, by re-implementing the control and/or data path functions into low-cost, low-complexity ASIC devices.

System Overview

The system overview is shown in figure 2 below. The system functions are broken down as follows:

- R4000PC and Secondary Cache
- Opti EISA chip set and interface.
- Main memory and EISA expansion bus.

R4000PC and Secondary Cache

The secondary cache on the CPU main memory bus is designed to provide a balance between ease of design/low cost and high-performance, high-bandwidth.

In order to achieve high-bandwidth, the cache implements a 128-bit wide memory array. Thus, a single cache read can provide four 32-bit words, which are returned to the R4000PC as two 64-bit pieces of data. The two 64-bit datums are available in adjacent cycles, minimizing processor latency on cache hits.

In order to minimize the cost and complexity of the secondary cache, a few tradeoffs were made:

- The cache is implemented as write-through, rather than write-back cache. This greatly simplifies both the control and data path logic. The cache logic is further simplified by the R4000 "write-behind" operation; that is, on a cache miss that requires a memory writeback, the cache miss read is processed prior to the cache line writeback; thus, memory write performance is not a first order control of system and processor performance.
- The cache contains 7 tag bits + 1 valid bit. Thus, the total amount of main memory is relatively limited (relative to the 64GB address space of the R4000). Nonetheless, the cache architecture allows a main memory of 64MB, which is much larger than what is found in non-server PC's.

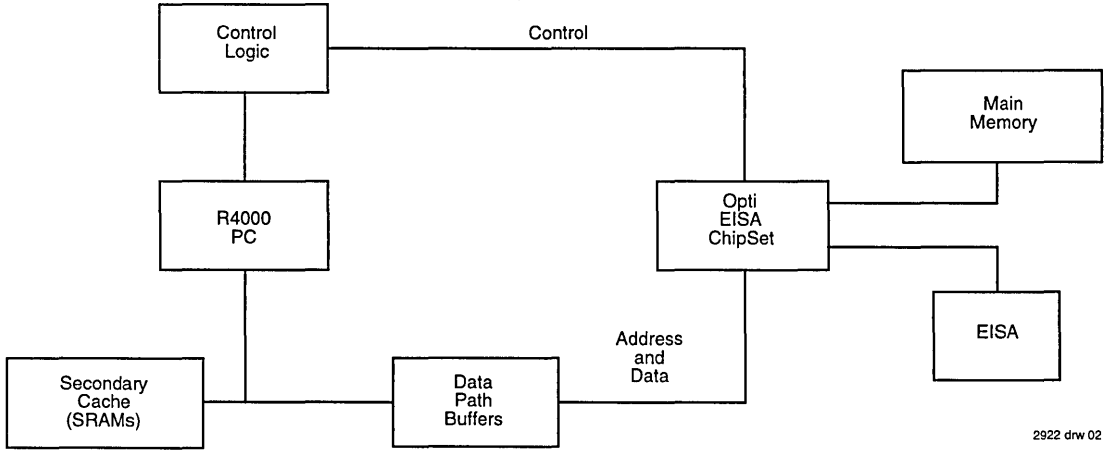


Figure 2. System Block Diagram

- The cache is implemented using 25ns 32K x 8 SRAM, with a separate but similar SRAM for the cache tag. Thus, relatively modest speed SRAMs are used, and no specialty TAG rams are required.

The cache glue logic is implemented using standard PALs and data buffer chips. The data buffers sequence the data between the R4000PC, its cache, and the host bus interface to the Opti chipset.

Table 1 shows the Cache Logic control signals, broken down by group. As can be seen from this relatively short table, this cache control logic can easily be replaced by a low-cost ASIC device, to further reduce cost.

TABLE 1. SYSTEM CACHE CONTROL SIGNALS

Host Buffer Control	Latch Control	Cache Control	Tag Bits
HdClk(1:0)	AClkEn	CacheEn	SADT(9:0)
HWW(1:0)	AClkEn	CBusEn	Tag(9:0)
HRD	DClkEn	CB0WE	TagOE
PDClk		CB1WE	TagWE
		CB0En	Valid
		CB1En	
		CB0OE	
		CB1OE	
		CWrite0	
		CWrite1	

i486 Host Bus Interface

The other primary design consideration had to do with implementing a reasonable i486 host bus interface between the R4000PC/secondary cache subsystem, and the Opti chip set.

Fortunately, the relatively large cache resources available to the processor (both on-chip and in the secondary cache) serve to largely decouple system performance from the main memory bandwidth. Thus, a relatively simple host bus interface could be constructed, minimizing both design time and system cost. The host bus interface in the initial system requires 8 buffer devices.

The main memory is directly controlled by the Opti chip set, and provides a relatively modest 40MB/sec of peak bandwidth. The main memory system is only 32-bit wide DRAM; thus, R4000 cache line refill requires four page mode accesses. Similarly, the memory system does not support a burst write protocol; cache line writeback is processed as four separate write transactions.

The i486 host bus interface contains 32 address bits, 32 data bits, and 10 control bits. In this system, it is derived from the R4000PC interface, which uses a 64-bit data bus which is time multiplexed to include 36 address bits, and which uses 18 bits of control data (actually, the R4000 uses a bi-directional command bus, and other control signals to coordinate transfers on the bi-directional control and data busses).

Table 2 shows the signals derived from the host bus control logic interface. Again, the number and types of signals are small and simple enough to be easily integrated into the same ASIC which implements the cache control.

The 32-bit host bus address is derived from the processor 36-bit address by registering the SysAD bus from the R4000PC. Since the i486 bus is only 32-bits, the upper four address bits from the processor are dropped.

The host bus data interface is constructed from a pair of 32-bit data transceivers. By sequencing the output enables of the transceivers, the 32-bits host bus is created by effectively multiplexing the halves of the 64-bit processor bus. Since a

TABLE 2. HOST BUS INTERFACE SIGNALS

Address	Control	Arbitration/Cycle Control
HA(3:2)	HClk	Rdy
BE(3:0)	HMem	BRdy
	HWr	Hold
	HData	HoldA
	HLock	HAdS
		BLast
		BusReq

cache line write back involves two 64-bit chunks, and since a cache line read involves four 32-bit datums, the control logic insures proper data staging occurs between the R4000 SysAD bus and the host bus 32-bit data bus.

The control bus of the host bus is constructed by simply converting the various processor requests into the appropriate sequences of host bus requests. This conversion process obviously also utilizes the datastages, so atomic R4000PC requests can be broken down into the appropriate series of host bus transfers, and the data kept consistently timed with the request.

EISA Bus Interface

The EISA bus interface requires no modification from the Opti design recommendations. EISA is chosen because it provides fairly good I/O performance (compatible with a high-performance RISC processor), yet also allows low-cost ISA add-in cards to be used.

The system relies on the EISA bus to provide functions such as networking and SCSI. By relegating these to the system bus, maximum flexibility but minimal cost is obtained.

SOFTWARE CONSIDERATIONS

Obviously, the introduction of the R4000 as the system microprocessor does change the software requirements from those of a typical PC to that of an ARC (Advanced RISC Computing) system. Fortunately, Windows NT is architected to allow this kind of flexibility.

Windows NT directly supports the R4000 processor. That is, Windows NT runs native on the R4000. In addition to allowing new R4000 applications to be run, Windows NT allows existing 16-bit DOS and Windows applications to be run on the ARC system. Thus, Windows NT insures compatibility with older software.

Further, Windows NT allows a wide variety of underlying hardware implementations to be built, by segregating system specific functionality into a lower-level of software, called the Hardware Abstraction Layer (HAL). HAL code is responsible for machine management functions, such as cache management.

Thus, the task of software development for the Deskstation ARC system was minimized to providing a layer for Windows NT.

ARCS BIOS Firmware

The BIOS firmware is responsible for certain basic aspects of system software, including configuration management, OS installation support, and providing a uniform boot environment for Windows NT.

The BIOS firmware provides the low-level system I/O functions and the processor boot-up software. The firmware can be shadowed in the main memory, to provide higher performance than from EPROM accesses. The BIOS required approximately 40K lines of source code, and is approximately 200KB of compiled binary.

Hardware Abstraction Layer

The hardware abstraction layer then resides on the hard disk, along with the operating system itself. Whereas the BIOS provides very-low-level, almost OS independent system functions, the HAL is designed to provide the various system dependent runtime support functions for Windows NT.

The HAL was developed beginning with the HAL kit provided by Microsoft, and ported to the specifics required by the underlying hardware implementation.

SUMMARY

The combination of the R4000 microprocessor, the Windows NT operating system, a standard PC chip set from Opti, and some clever design work using low-cost discrete components enabled Deskstation to implement an extremely high-performance PC without incurring substantial system cost.

The system retains maximal flexibility, based on its design objectives. Future options include:

- Higher frequency versions of the R4000 family. Higher frequency parts do not necessarily raise the bus interface frequency, thus raising system performance without raising system cost.
- Lower cost versions of the R4000 family, including the forthcoming IDT Orion™.
- Other PC standard architectures. Once the problem of mating the R4000 to an i486 host bus interface is solved, other standard architectures, including ISA and various Local Bus standards, can be easily implemented.
- Cost reduction via "ASIC-ization."
- Performance improvement via cache expansion.

Thus, this system represents a technology baseline for the rapid adoption of IDT/MIPS RISC into the Windows NT desktop marketplace.

This paper was presented at the 1993 Windows Hardware Engineering Conference in Santa Clara, California.

Integrated Device Technology, Inc.

THE IDT R4600 POWERS INTER-NETWORKING APPLICATIONS

CONFERENCE PAPER CP-14

By Philip Bourekas
Manager, RISC Product Definition & Applications Engineering, Integrated Device Technology, Inc.

INTRODUCTION

In recent times, there has been significant expansion in the number of applications using embedded RISC processors. Inter-networking equipment is one of the most visible applications to embrace the price-performance available from embedded RISC processors.

The IDT R4600 (Orion) dramatically increases the performance available to this application class, by tripling (or more) the performance available to the embedded system designer, while achieving the cost and power goals of an embedded system. To fully appreciate what the Orion brings to this application, one must look at what the application requires, and then examine how the Orion addresses those needs.

INTER-NETWORKING SYSTEMS

Inter-networking applications emphasize different architectural capabilities than do laser printer or desktop computing applications. As this market continues to advance, it is expected to place higher demands on embedded processors, as database sizes increase, transmission rates go up, and additional protocols and media become supported.

It is clear that there are a few processor capabilities that will continue to be valued most highly:

- *Packet movement* will emphasize the available bandwidth of the processor. What will be especially important is the ability of the processor to move the kind of data found in the packet header into and out of the CPU, for packet processing. In some systems, it may be important for the processor to perform the movement of the entire packet as well, although this is often done by DMA.
- *Packet processing*, including routing and protocol conversion, will continue to require rapid completion of relatively simple calculations. The single cycle nature of RISC boolean, ALU, and load/store operations serve this need well; higher frequencies and larger caches enable more of the peak performance to be actually achieved.
- *Interrupt response and task switching* times will be key metrics for the processor. Relatively low interrupt and task management overhead allows more of the processor performance to be directed to the packet processing operation, rather than processor state management.

The Orion speeds each of these key metrics, resulting in more value (more packets/second and/or more channels) from the resulting system.

ORION OVERVIEW

The IDT R4600, also well known as the Orion, is the latest and highest-performance member of the IDT RISController family.

The Orion is derived from the R4000 architecture, and shares many traits in common with the original R4000 devices. These traits include: 64-bit architecture, high-speed pipeline, and large on-chip caches. However, the Orion represents an independent design effort, targeting lower cost, and lower power consumption, than the R4400.

Key attributes of the Orion include:

- 64-bit integer CPU
- 64-bit FPA
- 16kB, 2-set associative instruction cache
- 16kB, 2-set associative data cache. The data cache can be managed with a mix of write back and write through protocols
- 5 stage traditional RISC pipeline operating currently at 133MHz, scalable to 200MHz.
- 64-bit burst interface bus
- Flexbus™ allows the bus interface to be run at 1/2 to 1/8 the pipeline clock rate.

Figure 1 gives a block diagram representation of the Orion.

ORION'S PACKET MOVEMENT CAPABILITY

Although RISC processors are typically known for their computational performance, inter-networking performance is typically more dependent on the processor's ability to move data rapidly through the system.

Note that what is required here is more than just an efficient block copy: in processing the packet header information, the processor must be able to rapidly process unaligned Big-Endian data, and must be able to efficiently handle data structure accesses. These areas are key strengths of the IDT Orion.

Most systems utilize external DMA engines to actually move the packet data between channels. However, other systems may employ the processor for this task, under software control.

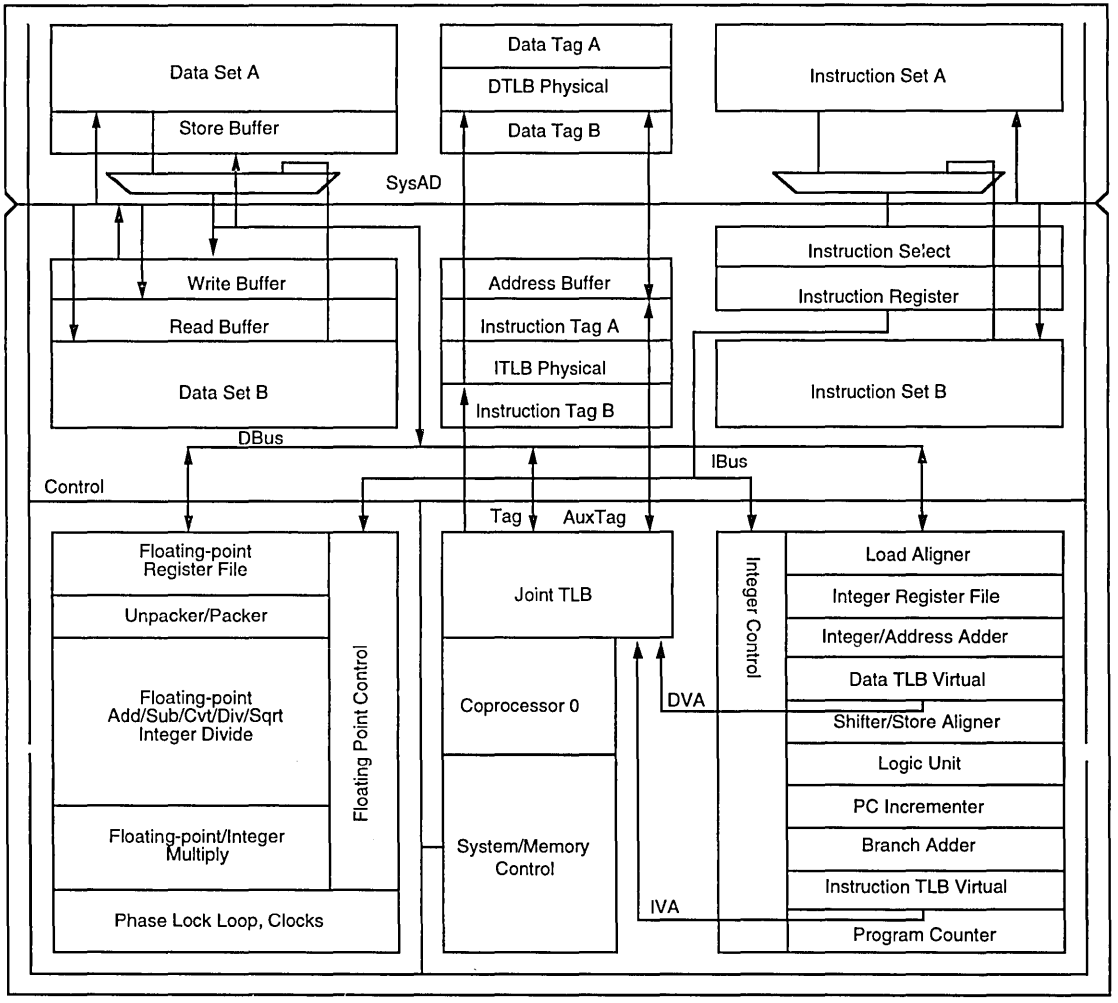
If the system approach is to utilize external DMA, then the processor must be able to rapidly process the packet header information, perform the routing, and then perform the DMA channel pointer management. In addition, the processor needs to allow the external DMA to have significant amounts of bandwidth left for its data movement.

To support these goals, the Orion implements high-bandwidth, both internally and on the bus, to allow the packet header information to be moved rapidly; large on-chip caches to speed the routing (including a large data cache, which can contain significant amounts of routing information), and a high-speed pipeline. In addition, the large caches (which can be managed using a writeback protocol) insure that the

processor operates most frequently out of the on-chip memory, leaving significant amounts of bandwidth available for external DMA.

In the case of software controlled data movement, the processor needs the attributes described above, but also needs to be able to move data in efficient bursts. To support this need, the Orion implements high-bandwidth (described below), and a set of cache operations to allow the programmer to explicitly access this bandwidth (useful if the data movement is designed to flow-through the CPU). Alternately, the system can utilize a fly-by technique; when the processor reads data from a certain address range, external system logic can sample the data simultaneous with the CPU. This avoids the need for the processor to later utilize write cycles, at the cost of some system logic. For these systems, the large address space of the Orion enables "aliasing" of system memory, simplifying the design of these fly-by techniques.

The Orion strategy for bandwidth is to implement high-bandwidth between the on-chip register file/functional units and the on-chip caches, and separately to implement high-bandwidth between the on-chip caches and the external main memory. Rather than consume valuable chip real estate (and slow context switch performance) with a windowing register file, the MIPS architecture uses a large orthogonal register file, with the cache feeding the register file at over 1GB/second. In addition, the cache is able to hold relatively complex data structures (as is found in complex systems programmed in high-level languages), rather than being limited to "word" and sub-word data (as is typically found in a register windowing system).



3106 drw 01

Figure 1. IDT Orion Block Diagram

In addition to raw bandwidth from the data cache, the Orion implements other techniques to speed packet movement (especially for key header and routing data) in an inter-networking system.

Big- or little- endian memory system support. The Orion directly implements big- or little- endian systems, as well as systems of mixed byte ordering. Most inter-networking applications will implement Orion using a Big-Endian system, which is compatible with the byte-ordering conventions of networking protocols.

Unaligned datum support. Since packet data is not guaranteed to be aligned to word addresses, unaligned data support is important in achieving the desired bandwidth. The Orion implements the MIPS "load-word-left/right" instructions, and complementary store instructions. These instructions are designed to support 32-bit 64-bit datums which may not be aligned on the proper modulo byte address. These instructions eliminate the exception/emulation method required in other processors, and also eliminate the need to process the unaligned word through a series of byte-load/shift/OR operations. Instead, a single pair of instructions can be used to load or store a 32- (or 64-) bit quantity between a single CPU register and the memory system. This mechanism allows these "split" datums to be loaded in just 2 clock cycles.

These operations can be invoked either through assembly level programming or from C. For example, the Gnu C compiler uses unaligned operations to move data marked with the "packed" attribute.

Ultra-high on-chip bandwidth. Inter-networking requires high data-bandwidth, so that packet data can be brought rapidly in and out of the processor registers. To support this need, the Orion implements large on-chip caches. With dual 2-set associative, 16kB caches on chip, the Orion delivers average performance very close to its 133-MIPS peak performance, by allowing most instruction accesses to be served from the on-chip cache. In addition, when packet data is accessed cacheably, data can be brought into the cache at over 500MB/sec, and subsequent cached accesses to additional portions of the packet will occur at over 1GB/second. Concurrently, instruction fetch bandwidth exceeds 500MB/s.

Many embedded processors provide special mechanisms for high data bandwidth. However, the effectiveness of these mechanisms is dramatically reduced if the execution engine is "starved" of key instruction and data information by insufficient on-chip caches. The Orion, on the other hand, provides large on-chip caches, to keep the engine running at full speed.

There is yet another advantage from these large caches; systems which employ external DMA engines to move packet data through the memory and I/O systems will find more of the bus bandwidth available to them. Since the CPU will be able to execute for long periods of time from the internal caches, and since the data cache can be managed with write-back protocols, the processor will require the bus only infrequently.

Early restart of execution. To facilitate real-time processing of packet data, the Orion restarts execution as soon as the requested datum is brought on-chip from memory or I/O devices, even if additional data is being brought in to fill a cache line. The rest of the cache line (which usually contains additional useful packet data) fills the on-chip cache simultaneous with the processing of the first datum. This parallelism allows the bus bandwidth to proceed in parallel with the execution bandwidth. With instruction cache fetches occurring at over 500MB/sec, and data accesses at greater than 1000MB/sec, and the bus moving data at 500MB/sec, the Orion represents over 2GBytes/sec of packet movement horsepower.

64-bit datum support. Bulk data movement, such as fetching of packet headers, or block copies, can take advantage of the 64-bit operations of the Orion. This elevates peak bandwidth, and allows more data to be processed in a single operation. Processing power is also increased, since more of the packet header or data is processed in a single operation.

Varying cache management protocols. Inter-networking applications manage diverse types of data, including relatively "static" data such as the program stack, task queue, and routing table entries, as well as the more "dynamic" packet data. To speed both types of data, the on-chip caches support both write-through and write-back operation. In an inter-networking application, general processing data (such as the runtime stack) benefit from the cache write-back algorithm, while packet data, which may later be DMA'd out on another network channel, are managed using the write-through protocol to insure cache and memory coherency. The on-chip write-buffer allows the execution core to continue processing additional data, as write-through or write-back data gets processed out to the memory system. The addressing modes of the Orion allow subfields from data structures to be rapidly accessed, using base-address plus sub-field offset addressing directly in the load or store instruction (and thus eliminating explicit address calculation instructions).

The varying write protocols, coupled with the large address space, also enable the system designer to implement "aliased" physical memory. Either through the use of the on-chip MMU, or through address decode logic, multiple virtual address spaces can be mapped to a single physical address space. By assigning differing write protocols to the various virtual spaces, the programmer can then choose to reference data as uncacheable, cacheable with writeback, or cacheable with write-through, merely by the choice of the virtual address used. When the MMU is used, the programmer can further use the multi-tasking capability of the MMU to insure that code is "well-behaved", by limiting access to certain virtual address regions to certain tasks.

Explicit cache management support. To allow the system to directly control the available bandwidth, Orion provides a set of "cache operations". Cache operations can be used to pre-load the caches with desired data and/or instructions, and can also be used to initiate the write-back of data to insure cache and memory coherency before DMA activity occurs.

The Orion cache ops allow the assembly programmer to explicitly manage the bandwidth between the cache and the external main memory; they can be used to initiate burst reads and/or writes of main memory, for example. This facility, coupled with the fact that the Orion executes multiple instructions per bus clock cycle, enables the system to achieve average bandwidth close to the peak bandwidth of the interface.

PACKET PROCESSING

The Orion really shines in packet processing. The Orion features 133-MIPS execution, which is typically sustained by the large on-chip caches. Thus, the Orion can quickly determine the appropriate routing or conversion for a packet, perform the operation, and use its high-bandwidth to dispatch the packet. By reducing the amount of time to obtain the packet, to process the packet, and to dispatch the packet, the Orion supports higher "packets-per-second" rates, as well as higher numbers of channels under the control of one processor.

Instruction throughput. Other processors may claim high peak MIPS rates; however, on closer examination, it becomes obvious that these rates are rarely achieved. The effects of data dependencies and issue restrictions on pipeline throughput, coupled with the low hit rates associated with small caches to feed the engine, dramatically degrade actual system performance.

To avoid this problem, the Orion architects made certain fundamental decisions: the pipeline would be a traditional RISC pipeline, avoiding the issue restrictions found in most superscalar machines and the pipeline bubbles found in super-pipelined machines; the pipeline would be high-frequency; and the pipeline would be sustained by large, high-bandwidth, efficient on chip caches. Thus, the performance ratio between the "133-MIPS Orion" and other embedded RISC processors targeted to inter-networking is actually significantly larger than the ratios of their peak MIPS rates. Figure 2 shows the Orion pipeline structure.

High-level language programming. This traditional RISC micro-architecture has other benefits as well. For example, the Orion architecture, as with the MIPS architecture in general, is "high-level-language friendly". The optimization rules for it are easily supported by modern compilers. Thus, the system programmer can achieve the performance potential of the Orion without having to program in assembly, resulting in code that is more portable and easy to maintain. The Orion is designed to allow efficient translation of "C" programs to its object code. The large caches allow the programmer plenty of "elbow room" for system software, without requiring assembly level tuning.

64-bit data support. Inter-networking applications can also find advantage in using the 64-bit ALU and boolean operations of the Orion, as well as the high-bandwidth from its on-chip 64-bit wide registers to memory. These operations enable the Orion to process more data in a single chunk.

Although the Orion is a true 64-bit architecture, it is equally adept with smaller data. The system can be constructed to use 32-bit addresses (reducing the size of pointers, and thus using less memory) and to use 64-bit operations only for "long" data. This is accomplished merely by the selection of appropriate compiler switches, along with the types declared for datums.

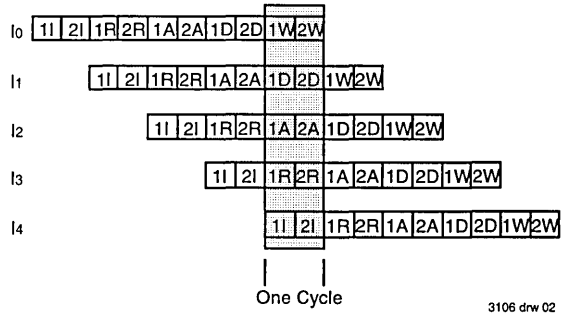


Figure 2. Orion Pipeline

INTERRUPT RESPONSE AND TASK SWITCHING TIME

The Orion also excels at minimizing the overhead for exception processing and task switching. The Orion does not require explicit pipeline state management, cache flushing, TLB or MMU management, or register spill management. Exceptions feature very low latency, and special registers in the Orion facilitate exception decoding and interrupt service dispatch. Thus, very little overhead is required in the interrupt and task switch model, leaving more processing power for packet movement and processing.

Although many vendors attempt to use the time required for exception recognition as a measure for real-time efficiency, this dramatically understates the requirements of a real system. True exception latency is a function of exception recognition, exception decode, state preservation, state restoration, service dispatch overhead, and instruction throughput. A number of these factors are operating system specific (for example, the amount of state preservation/restoration, and the overhead for prioritization and task selection).

Exception recognition. In general, the amount of time required to detect an exception is less than the pipeline length. In the particular case of interrupts, latency is 5-6 cycles. Of course, the longest CPU stall cycle (e.g. due to a main memory access) can lengthen the amount of time.

Once an exception is detected, the Orion will (automatically):

- enter kernel mode
- disable interrupts
- encode state information in on-chip registers designed for exception management
- branch to an exception vector location

These activities are automatic, and occur in the few cycles of exception latency mentioned above. Figure 3 illustrates the exception latency.

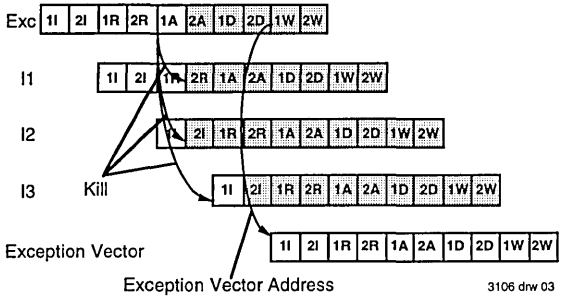


Figure 3. Orion Exception Latency

Exception service dispatch. Once the Orion has branched to the exception vector, software is responsible for decoding the cause of the exception, performing whatever state preservation is appropriate, and dispatching the service routine.

The Orion contains registers designed to speed exception decode and to simplify return to normal execution at the end of exception processing. These registers show the cause of the exception, the return address, and other bits of information for decode. Because of these registers, exception decode, minimal state preservation, and service dispatch can be performed in as few as 15 instructions (less than 120ns). Figure 4 shows the code typically executed at the general exception vector.

```
.set      noreorder          #tell assembler not to fill delay slots
la        k0,except-regs     #fetch address of reg save array
sw        AT,R_AT*4(k0)      #save a few general registers
sw        v0,R_V0*4(k0)
sw        v1,R_V1*4(k0)
mfc0      v0,C0_EPC          #fetch return address
mfc0      v1,C0_SR           #fetch status register
sw        v0,R_EPC*4(k0)     #save return address
mfc0      v0,C0_CAUSE        #fetch exception cause register
sw        v1,R_SR*4(k0)      #save status register
sw        a0,R_AT*4(k0)      #save another general register
and       v1,v0,EXCMASK      #get at the actual "cause index"
lw        a0,cause-table(v1) #get address of service routine
sw        a1,R_A1*4(k0)      #use delay slot to save another reg.
j         a0                 #branch to service routine
sw        k1,R_k1*4(k0)      #save one more general register
.set      reorder            #re-enable pipeline scheduling
```

Figure 4. Exception Service Dispatch code

Note that the advantages of the Orion register, cache, and MMU architecture serve to minimize the amount of state software needs to preserve. Specifically, the use of unique process ID's avoid the need to flush the MMU at context switch; physically tagged caches eliminate the need to flush the on-chip caches; and the orthogonal, non-windowed register file eliminates the need to manage window overflow.

Return from Exception. Returning from exception is equally simple and quick. Basically, the software needs to restore the original machine state registers and any preserved context, and execute a return to the saved return address. One additional instruction, the *eret* instruction, restores the bits of internal state designed to be hidden from the programmer. A minimal return/restore from exception sequence can be implemented in as few as 7 instructions (less than 60ns). Figure 5 shows this typical code.

```
.set      noreorder
                             # by the time we have gotten here
                             # all general registers have been
                             # restored (except k0 and v0)
                             # reg. AT points to the reg save array
lw        k0,C0_SR*4(AT)     # fetch status reg. contents
lw        v0,R_V0*4(AT)      # restore reg. v0
mtc0      k0,C0_SR           # restore the status reg. contents
lw        k0,R_EPC*4(AT)     # Get the return address
lw        AT,R_AT*4(AT)      # restore AT in load delay
j         k0                 # return from int. via jump reg.
          eret               # the eret instr. is executed in the
                             # branch delay slot
.set      reorder
```

Figure 5. Exception Return code

Instruction Throughput. As illustrated above, service dispatch and return from exception are performed via simple software functions requiring very few instructions. Thus, the key to minimal exception service latency is to keep instruction throughput high.

As discussed above, the Orion is able to sustain extremely high instruction throughput rates, based on its 133MHz pipeline fed by its large internal caches. Various techniques, including cache locking, fast local memory, and appropriate data cache protocols, can also be used to sustain the high-rate of instruction throughput.

Special techniques. Note that it is possible to use the on-chip registers reserved for the on-chip FPA as a high-bandwidth backup store for processor state in certain exceptions. Software can be written to use these registers as a small stack for key machine state. Since the Orion implements sophisticated dynamic power management on chip, these registers can be used without incurring a large increase in CPU power consumption.

SUMMARY

The Orion serves as an excellent device for inter-networking applications. With over 2 GB/sec total bandwidth, efficient management of packet data (including unaligned data), and 133 MIPS processing power, the Orion provides the CPU resources necessary to support the increasing requirements of inter-networking, imaging, printing, multi-media, and desktop computing applications, while maintaining the cost and power goals required by these applications. As the Orion frequency continues to increase, the performance gap with other architectures will widen further.

Integrated Device Technology, Inc.

SYSTEM DESIGN ISSUES WITH THE R4600/R4400 PROCESSORS

CONFERENCE PAPER CP-15

By: Russell Cummings

INTRODUCTION

This article describes the basic issues related with the design of a system using either the IDT79R4400 or the IDT79R4600 64-bit CPU. It will cover the concepts of the system interface between the CPU and the rest of the system and give an example of a zero-wait state SRAM based memory system. The major focus will be on the system interface, how it relates to the rest of the system and the new features of the R4600 to improve performance. To end the article, I will discuss some of the issues of a zero-wait state memory system for the R4600.

THE R4X00 SYSTEM INTERFACE

The system interface connects the R4x00 CPU to external memory and other peripherals. This section will discuss the various aspects of the system interface including the signals used.

The system interface consists of three main elements:

1) The 64-bit multiplexed address and data bus; SysAD[63:0]
2) The 9-bit command bus; SysCmd[8:0]
3) The 6 handshake signals to control issue rates and validate requests; $\overline{\text{RdRdy}}$, $\overline{\text{WrRdy}}$, $\overline{\text{ValidOut}}$, $\overline{\text{ValidIn}}$, $\overline{\text{ExtRqst}}$ and $\overline{\text{Release}}$

THE SYSAD BUS

The SysAD bus is shared for both addresses and data cycles. It will present addresses during cycles where a valid interface command is present on the SysCmd bus. Data will be presented during cycles which have a valid data identifier on the SysCmd bus. During the address cycles, only the 36-bit physical address will be driven on SysAD[35:0] allowing up to 64GB to be accessed. For the R4400, the unused address bits will be driven as zeros. The R4600 will drive zero on SysAD[55:36] and will drive virtual address bits 19..12 on SysAD[63:56].

THE SYSCMD BUS

The 9-bit SysCmd bus is used to encode the type of transaction that is present on the SysAD bus. SysCmd[8] will indicate whether the current driven cycle is a command (SysCmd[8] = 0) or data (SysCmd[8] = 1). During the address cycles, the other bits encode the type of cycle (read, write or null) along with the amount of information to be transferred. For the data cycles, the remaining bit determine if the current datum is the last of the transfer, if the data is for a read request, if there is an error and if the CPU should check the parity.

THE SYSTEM INTERFACE CONTROL SIGNALS

The system interface control signals are used to communicate when buses have valid data and when the external system is ready to accept a command. The output, $\overline{\text{ValidOut}}$ and the input, $\overline{\text{ValidIn}}$ are used by the CPU to indicate when there is valid information driven on the bus either by the CPU ($\overline{\text{ValidOut}}$) or the external system ($\overline{\text{ValidIn}}$). Two input signals, $\overline{\text{RdRdy}}$ and $\overline{\text{WrRdy}}$, are used by the external system to indicate to the CPU that it can accept a command. The CPU output, $\overline{\text{Release}}$ indicates to the external system that the CPU has releasing the SysAD and SysCmd buses and it can start driving these buses after a bus turn-around cycle. The external system will indicate to the CPU that it need the system buses by asserting the $\overline{\text{ExtRqst}}$ signal to the CPU.

SYSTEM INTERFACE PURPOSES

The major purpose of the system interface is to handle requests that arise from system events. The system events include; Load Misses, Store Misses, Store Hit on a write-through page (R4600 Only), an uncached Load/Store or CACHE operations. These system events will translate into one or more requests from the processor, Processor Requests, or the external agent, External Requests. There are two Processor Requests; a Read Request and a Write Request. There are three External Requests; a Read Request (although there are no readable CPU resources), a Write Request (to write the interrupt register) and a System Interface Null Release Request.

PROCESSOR REQUESTS

Processor requests are used to transfer data between the processor and the external system. The processor will issue a read request either for a cache line sized block (a Block Read Request) or for an uncached datum (a Word Read Request - can be either a doubleword, word or partial word access). A processor write request can also be of a Block or Word type. The Block Write will be of a cache line that is possibly being replaced. A Word Write can result from either an uncached store or from a store hit to a cache line whose page attribute is write-through (this is for the R4600 only). Each processor request will have an "Issue" cycle after which the CPU will finish the rest of that transaction.

PROCESSOR REQUEST ISSUE

The processor samples the $\overline{\text{RdRdy}}$ and $\overline{\text{WrRdy}}$ signals to determine when a read or write request has issued. These signals are sampled at the rising edge of the S Clock (the system interface clock). The actual issue cycle will be the

cycle two S Clocks after the respective control signal ($\overline{\text{RdRdy}}$ for read and $\overline{\text{WrRdy}}$ for write) was sampled as asserted. Figure 1 shows this for a read request issue cycle.

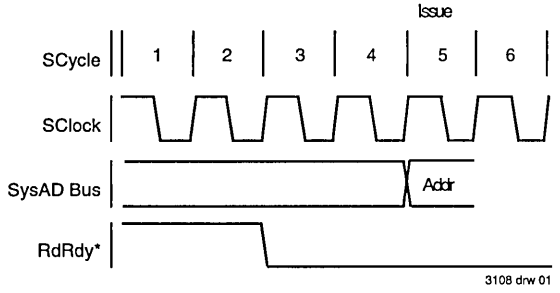


Figure 1: Read Request Issue Cycle

PIPELINE RESTART FOR READS

When a cache miss occurs, the pipeline will stall until some or all the data for the miss is returned. For the both the Instruction and Data caches of the R4400 and the Instruction cache of the R4600, the pipeline will stall until after the entire cache line is returned. The first returned doubleword will contain the missed instruction or data. In the R4600, when a Data cache is serviced, the pipeline will restart after the first doubleword is returned. This first doubleword contains the missed data. The rest of the missed cache line is returned in parallel. This can result in significant performance increase due to the data streaming and also results in more efficient use of the CPUs resources.

PROCESSOR READ REQUESTS

When the processor needs some information from the external system or need the next instruction(s) to execute, it will use a read request to get it. A read request begins by driving the address on the SysAD bus and the appropriate read command on the SysCmd bus. For a block read, the command will indicate that a block is needed and the size of the block; this can be 4 or 8 words for the R4400 while the block size is fixed at 8 words for the R4600. For a word read, the command will indicate this along with the number of bytes it expects returned. With the valid address and command driven, the CPU will assert the $\overline{\text{ValidOut}}$ signal to let the external system know that there is valid information on the buses and that the external system is expected to service the request. If the $\overline{\text{RdRdy}}$ signal has meet the requirements for an issue, the valid address and data is driven for that cycle only, otherwise the CPU will continue to drive the same information until the issue requirement is meet. For the R4600, the CPU will further indicate an issue cycle by asserting the $\overline{\text{Release}}$ signal in the issue cycle. After the $\overline{\text{Release}}$ signal is asserted for the one cycle, the CPU will 3-state the buses to allow the external system to start driving them. The R4600 guarantees that the $\overline{\text{Release}}$ will assert in the issue cycle. For the R4400, the cycle for the $\overline{\text{Release}}$ to be asserted can be the issue cycle but may be delayed by several cycles based on internal activity.

After the release cycle, the buses will "turn-around" to allow the external system to drive the read response data back to the CPU. The external system will drive valid data on the SysAD bus along with a command on the SysCmd bus to indicate the this is read response data. The command will also tell the CPU other information about the returned data such as: if the data is erroneous, if the CPU should check the data and check bits and, for a block read, if this is the last data element for the block. Once the external system has the valid information on the buses, it will assert the $\overline{\text{ValidIn}}$ signal, this indicates to the CPU that it can now sample the buses for the requested information. After the external system returns the last data element, it will 3-state its drivers and turn the buses around for the CPU to start driving after a one cycle delay.

BLOCK READ REQUESTS, MORE DETAILS

When the R4x00 issues a block read request, it expect the data to be returned in "sub-block ordering". The idea behind sub-block ordering is to start with the doubleword at the miss address, the address driven by the CPU to start the read request. The external system will determine the next doubleword address to return by the XOR of the start address with the value of a binary counter. The number of bits in the binary counter are determined by the line size the CPU uses. For example, with 8 word lines, one needs a 2-bit counter and will XOR the count with address bits 4..3 to determine which double word to return next. This scheme works well with interleaved memory systems. The overall, the algorithm is:

- Get the doubleword which missed first;
- Next, get the doubleword which will fill out the quadword containing the missed data;
- Then get the quad word filling the octalword, in the same order as the previous quadword.

PROCESSOR WRITE REQUESTS

As with the read request, a write request can be either a block or a word write. For a block write, the CPU will first drive the start address on the SysAD bus and the block write command on the SysCmd bus with the $\overline{\text{ValidOut}}$ signal asserted. For the R4600, the CPU will start sending the data out in the cycle immediately following the address issue cycle. The address issue cycle follows the rules stated before with respect to the assertion of the $\overline{\text{WrRdy}}$ signal. The number of cycles between the doublewords is programmable at boot time. This is the data write-back pattern and can be as fast as a doubleword every cycle to a doubleword every 4 cycles. The setting used is determined by the speed at which the external system can handle the write data. For the R4400, there can be a delay between the address issue cycle and the first doubleword out but the remainder of the write-back will occur at the programmed rate. One thing to note is that once the write has issued, there is no way to stop the write-back and there is no way to dynamically throttle the number of wait cycles between the doublewords.

For single writes, the address issue cycle is the same as that of the block write. The R4400 can then have an unused cycle after which it will drive the data out and follow this by another unused cycle. This results in a 4 cycle minimum for a

single write and is due to the internal state machine implementation.

The R4600 can perform single writes in the same manner or it can use one of the two new write modes. These write modes are programmed at reset. In the R4x00 compatible mode, the R4600 will issue the write address, immediately follow this with the write data and finish the write cycle with 2 unused cycles during which it will continue to drive the valid data on the SysAD bus but will only have the $\overline{\text{ValidOut}}$ signal asserted for the first data cycle. This again results in 4 cycle writes. The new write modes are Write Re-issue and Pipelined Writes, both of which result in 2 cycle writes.

For the Write Re-issue protocol, the CPU will first look at the WrRdy signal as with other writes to see that it is asserted two cycles previously but in addition, the WrRdy signal must still be asserted in the issue cycle for the CPU to consider the write to have issued. If the WrRdy was deasserted in the issue cycle, the CPU will retain the address/data pair in the write buffer and re-issue the write once the WrRdy is again asserted at the appropriate times. The Write Re-issue is shown in Figure 2.

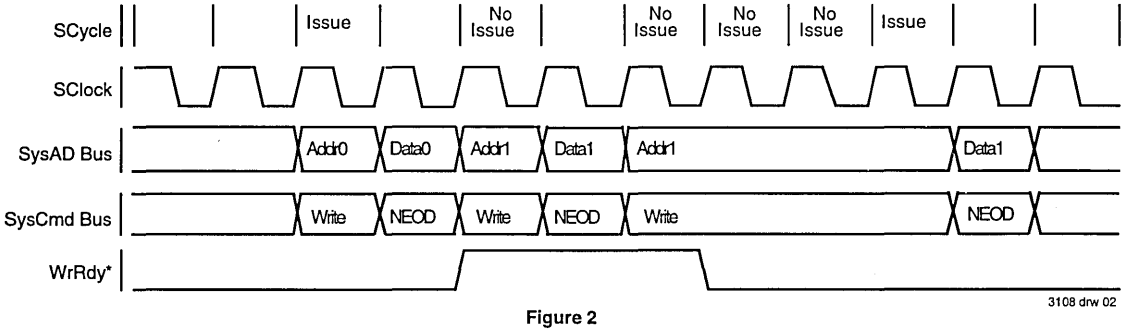


Figure 2

The Pipelined write protocol maintain the issue rules of the R4x00 compatible writes but eliminates the two unused cycles between back-to-back writes. The external system is therefore required to accept one additional write after the $\overline{\text{WrRdy}}$ is deasserted for a stream of back-to-back writes. The Pipelined Write is shown in figure 3.

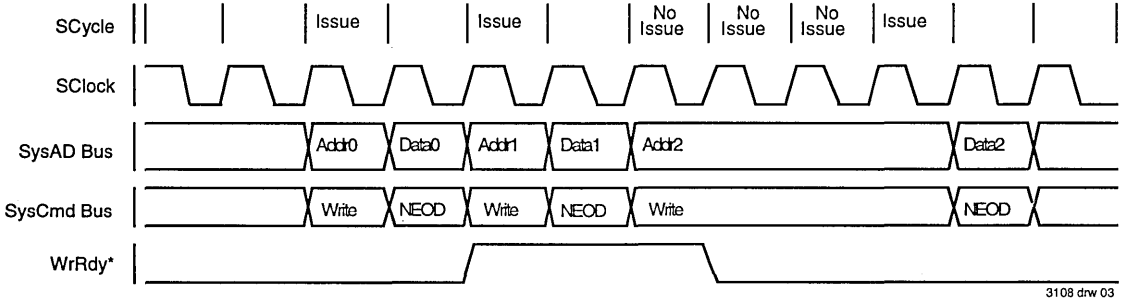


Figure 3

EXTERNAL REQUESTS

External requests are used by the external system to transfer information to the CPU. There are four possible external request for the R4x00: Read Response, Read, Write and Null. The Read Response is used for the return of data requested by a processor read request. The external Read request is intended to allow the external system to read data from internal CPU resources but there is currently nothing to read from. The external Write request is used to directly write to internal CPU resources with the Interrupt register the only currently implemented write-able resource. The external Null request is used by the external system to return the system buses to the CPU when the external system is finished with them. The Null request is only required if the external system is not sending data to the CPU, i.e., it is not needed for the read response or external write request, only if the external system has requested the system buses for some other use that the CPU is not involved with such as a DMA.

The external system must arbitrate with the CPU for the control of the system buses. This is initiated by the external system by first asserting the $\overline{\text{ExtRqst}}$ signal to the CPU. Some

time after the ExtRqst is asserted, the CPU will signal to the external system that it is giving up mastership of the system buses to the external system by asserting the Release for one cycle. One cycle after the CPU asserted the Release, the external system can start issuing its request(s). For a read response, external read and external write request, the CPU will regain mastership of the system buses once the external request is completed. If the external request does not involve the CPU, the external system will need to drive a Null request to the CPU in order to return mastership of the system buses to the CPU.

R4X00 CLOCKING

The R4x00 CPUs have several clocks that are involved in various aspects of operations. The input clock is MasterClock. The MasterClock is used by the internal PLLs to generate the other clocks. The MasterClock frequency is 1/2 the pipeline frequency. The MasterOut clock signal is aligned and at the same frequency as the MasterClock. This clock is used for the synchronous assertion and deassertion of the reset control signals. There are two internal only clocks, PClock and SClock. The PClock is the actual pipeline clock and is 2x the MasterClock frequency, i.e., if MasterClock is 50MHz, the PClock is 100MHz. The SClock is the system interface clock and it is used to clock all the into or out of the system interface. The frequency of the SClock is determined at reset through the programmable divisor which can be from 2 - 8. All the external timing, drive-out, setup and hold, are with respect to the SClock.

There are 4 more externally accessible clocks. The SyncOut-SyncIn pair are used as the feedback path one of the internal PLLs and is used to model the delays and loading of the external system. If the other two external clocks, TClock and RClock, are buffered, then an identical buffer is placed in the SyncOut to SyncIn path to allow the PLL to align the external clocks with the internal SClock so the user will know when the CPU will sample inputs and drive outputs. The other external clocks are used to sample outputs from the CPU or to clock signals to the CPU. The RClock is the receive clock and can be used by the external system to register the driven outputs from the CPU. The RClock is at the same frequency as the internal SClock but its phase leads the SClock by 25%. The TClock is the transmit clock are can be used to clock signals to the CPU. The TClock is also at the same frequency as the SClock and is aligned to the SClock.

R4X00 MEMORY INTERFACE EXAMPLE

As an example memory system, I will discuss a 0-wait state SRAM based memory. By 0-wait state I mean that the CPU requests run at the maximum speed. This means that the CPU requests complete in following number of cycles:

Block Read	3-1-1-1
Single Read	4
Block Write	2-1-1-1
Single Write	4 (2 for the R4600 new write modes)

The memory system will be 2-way interleaved with each back 64-bits wide. The address path will consist of a first level register that registers the address on the rising edge of the RClock, followed by a latch to provide a one-level address buffer to allow for the read following a write case of back-to-back CPU requests. For the registers, we use the IDT74FCT162823ET 18-bit registers and the latches are the IDT74FCT162841ET 20-bit transparent latches.

The data path consists of IDT74FCT162501 18-bit registered transceivers acting as registers for data to the memory and latches for data from the memory, and the IDT74FCT162260 12-bit tri-port bus exchangers to control the data from the 162501 to the even and odd banks of the SRAM memory.

Because the memory is interleaved, the read access is the limiting time, writes can overlap to some extent with the next transaction. To determine the read access time, we need to return the data to the CPU in at most 3 clock cycles (60ns for a 50MHz system interface bus). From this maximum time we first subtract the propagation delay times for the clock-to-out of the register and the address latch. Finally, we must subtract the propagation delay for the data to get to the CPU and the setup time required by the CPU. The result is the maximum read access time for the SRAMs.

For the given components, we get the following for a 50MHz system:

Access Time = 60ns - (address time) - (data time)
= 60 - (4.4 + 7.5) - (3.5 + 7.4) = 19ns

For this design we will use 15ns SRAM SIMMs to allow for a margin.

CONCLUSION

The R4x00 system interface has many features that can make a design challenging but with a little common sense and some careful planning, one can take advantage of the system interface and design and build a high-performance system.

Integrated Device Technology, Inc.

PORTING R3000 CODE TO AN R4400/ R4600 PLATFORM

CONFERENCE PAPER CP-16

By: Sami Khan

INTRODUCTION

The IDT79R4x00 family supports a wide variety of processor based applications including 32-bit Windows™ NT desktop or notebook systems. It is also suited for a variety of embedded applications, such as laser printers and data communications. R4x00 provides complete upward application software compatibility with the IDT79R3000 family of microprocessors.

IDT79R4x00 family extends performance range for embedded applications performing greater than 68 SPECint92 and 60 SPECfp92 at 100Mhz. Migrating earlier generation designs to R4x00 family of microprocessors is of great importance.

This paper describes various aspects of porting software from R3000 embedded system to a R4x00-based system. The discussion will start with changes in the software model from R3000 to R4x00 and associative changes in the kernel model. Next, various software modules that needs modification will be discussed. The IDT's System Integration Manager (IDT/sim™) software will be used as an example to emphasize major changes in the kernel model.

Finally, some of the compatibility issues between R4600 and R4400 will be explained to provide the better understanding as how to apply the modifications to R4600 system.

MIPS R4X00 FAMILY

IDT's R4x00 family is the extension of IDT's RISC road map. It targets various segments of embedded market such as disk arrays, color printers and routers. The architecture integrates full 64-bit integer and floating point units which are supplemented by larger caches. It is fully upward compatible with R3000 instruction set.

The changes in the software model depicts that only kernel model needs to be modified when earlier generation code (R3000) is migrated to the R4x00 architecture. User applications do not need to be modified even though some performance improvement by taking full advantage of MIPS III ISA. Changes in the software model include:

- System control coprocessor (CP0)
- Exception processing
- Memory management
- Instruction set
- Cache organization

SYSTEM CONTROL COPROCESSOR

System control coprocessor (CP0) has been has been completely changed. Existing R3000 status register has been modified along with the introduction of some additional registers. Changes in the status register reflects changes in the exception, memory management and cache organization from R3000 architecture. The new registers are:

Exception handling:	Cache error register
	Xcontext register
	Error EPC register
Memory Management:	Page Mask register
	EntryHi and EntryLo0,
	EntryLo1 register
	Index register
	Random and wired registers
Cache management:	Cache Tag registers

EXCEPTION PROCESSING

Besides changes in the exception model, newer exceptions were defined in R4x00 architecture. New exceptions are:

- Trap exception
- Floating point exception
- Reference to WatchHI / WatchLo address
- XTLB refill
- Cache error exception

Floating point errors are no longer mapped to Interrupt. Floating points errors are reported by an exception. Also exception vector locations has been changed. The R3000 exception vector base location for non-cache access has been changed from 0xbfc00000 to 0xbfc00200. Along with new exception vector base address, the location of exception vectors with respect to the base address has been changed. They are:

- TLB refill 0x000
- XTLB refill 0x080
- Cache error 0x100
- General 0x180

The 'rfe' (return from exception) instruction which is executed as the last instruction of exception handler has been replaced with 'eret' (exception return) instruction. These changes require major modification to the exception handling code. Again, this does not affect the user application as this code runs in kernel mode only.

MEMORY MANAGEMENT

In addition to 32-bit addressing mode, 64-bit addressing mode has also been introduced in R4x00 architecture. Moreover, three levels of security have been implemented. They are:

- User mode, for user applications
- Supervisor mode

- Kernel mode

Each mode's memory map can be configured as 64-bit virtual addressing space, or 32-bit virtual addressing space. Memory management logic maps this 32-bit or 64-bit virtual address space to 36-bit physical address space.

R4x00 has integrated 'fully associative' TLB. Unlike R3000 which has 64 entries, R4x00 has 48 entries, each mapping a set of odd and even pages. This effectively allows mapping of 96 pages at a time. Moreover, 256 process IDs allows multiple processes share the TLB without the need of flushing it at context switch.

R3000 has fixed page size of 4kB, whereas R4x00 allows variable page sizes, varying from 4kB to 16MB. Coherency attributes can be set for a page which allows the selection of cacheablity of the memory on a page by page basis.

INSTRUCTION SET ENHANCEMENT.

New instructions have been added to fully use 64-bit architecture. Additional instructions have been added for cache management, exception handling, 64-bit data movement, and data manipulation. These additions include:

- New CACHE operation instruction which allows cache management functions described later
- Double word load and store operations (LDL, LDR...)
- TRAP instruction for software trap exceptions.
- Double word arithmetic operations (DADDI, DADDIU...)
- Double word load and store operations to and from floating point coprocessor.
- Exception handling 'rfe' instruction has been replaced by 'eret' instruction.

CACHE ORGANIZATION.

Two level of caches are supported. On R4x00PC and R4600, only primary caches are supported. The caches are integrated on-chip and are separated as instruction and data caches. The cache size has been increased. R4x00 architecture can support maximum of 32kbytes. In case of IDT79R4400 and IDT79R4600, size is fixed to 16kbytes, both for instruction and data caches. Line size of the caches can be configured as 4-wide or 8-word wide. In case of R4600, line size is fixed as 32-byte.

Unlike the R3000 caches which has write-through update policy, the R4x00 has write-back caches. Caches are Direct map except for R4600, which has 2-way associative caches.

MODIFYING CODE

The porting of the R3000 code to R4x00 environment needs major modification to low-level kernel code. The user application can remain the same even though some performance gain can be achieved by taking advantage of MIPS III ISA.

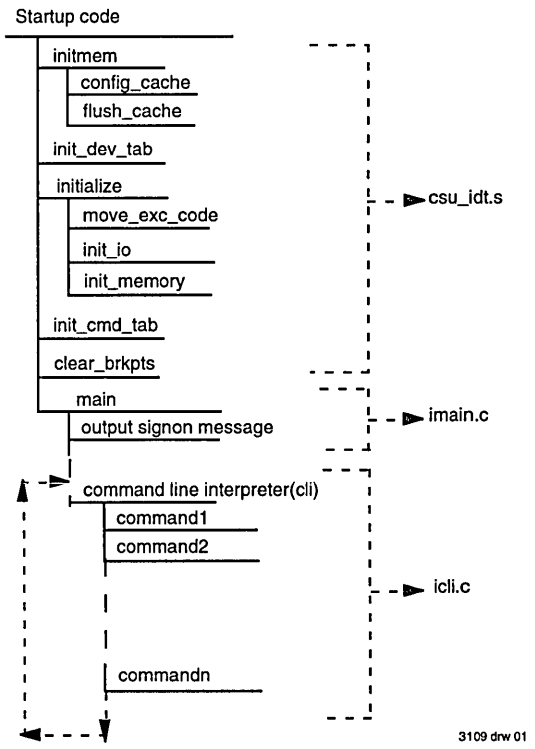
IDTsim can be used as an example to explain the modification required. IDT's System Integration Manager is a ROMable software product that permits convenient control and debugging of RISC systems built around R3000 ISA CPUs. Facilities are included to operate the CPU under controlled conditions, examining and altering the contents of memory, manipulating and controlling CPU resources. IDT/sim runs in kernel mode and was developed originally for the R3000 environment. In order to port the code to R4x00 platform, minimal modification is required to the kernel and will be used as an example.

In order to discuss IDT/sim, it is important to understand the execution flow of IDT/sim and see which modules need modification.

IDT'S SYSTEM INTEGRATION MANAGER (IDT/SIM)

IDT/sim starts by executing startup code and then jumps to the main program which runs command line interpreter. Commands can be entered on-line to execute functions of the monitor program.

Startup code performs several functions and then passes the execution control to the main program. This includes initializing caches, TLB, memory and configuring and initializing internal registers of the processors. The following diagram graphically explains execution flow of the startup code.



IDT/sim's Global Execution Flow

The startup code begins by initializing CPU registers. This includes writing to the status register which disables the interrupts globally.

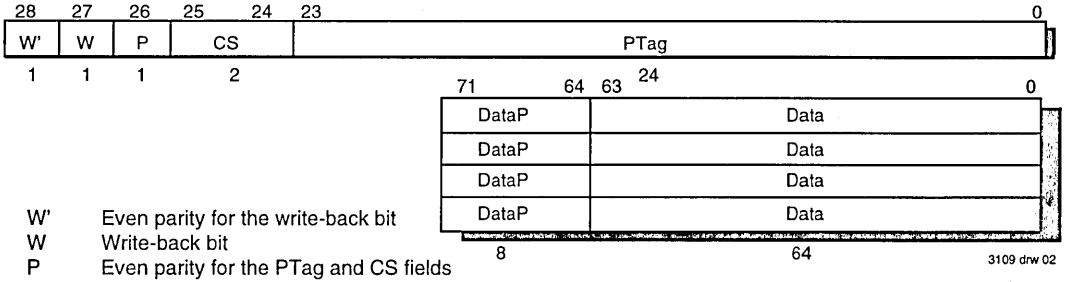
The next step is 'initmem' routine which does cache manipulation operations. Precisely, it configures and sizes the caches, then caches are flushed. This part of the code needs major modification as cache architecture has been completely changed for R4x00.

STARTUP CODE OF IDT/SIM

After initializing the status register, IDT/sim determines the size of the caches. In case of R3000, cache sizes are determined by isolating and then writing and reading different data patterns to the caches.

In case of R4x00, cache size and line size can be determined by reading the IC and DC bits and IB and DB bits of the status register, respectively.

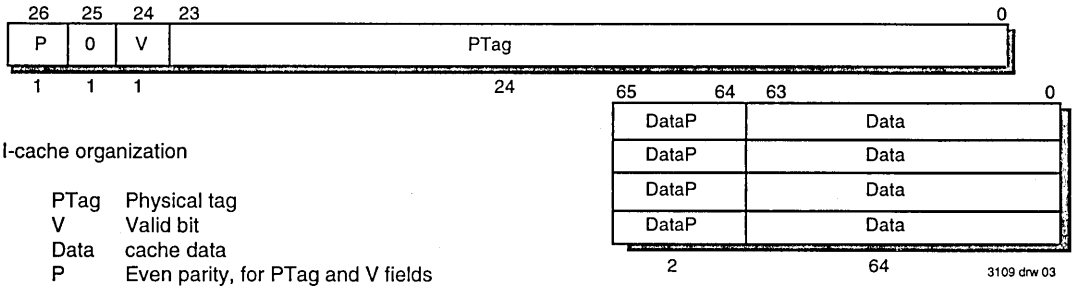
D-cache organization



W' Even parity for the write-back bit
W Write-back bit
P Even parity for the PTag and CS fields
CS Primary cache state: 0 = Invalid, 1 = Shared, 2 = Clean Exclusive, 3 = Dirty Exclusive

PTag Physical tag
DataP Even parity for the data

Figure 1



I-cache organization

PTag Physical tag
V Valid bit
Data cache data
P Even parity, for PTag and V fields
DataP Even parity; 1 parity bit per byte of data

Figure 2

Caches are invalidated in R3000 architecture by isolating and writing partial words to the them. In case of R4x00, cache operation instructions are provided which perform operations on caches, and when used in certain sequence, perform cache invalidation.

Figures 1 and 2 explain the organization of R4x00 instruction and data caches. Note that both data and tag parts of the caches are parity protected. At power-up, the states of the parity bits and the data plus tag fields are unknown. If caches are accessed in this state, we may get parity error and, unlike the R3000 architecture where parity errors are treated as cache miss, the R4x00 processor will take cache error exception. Therefore, invalidating the caches in the R4x00 architecture also involves forcing good parity in both the data and tag fields.

The invalidation involves following steps:

a) First, tags are initialized. The software first disables the cache error exceptions by writing to the DE bit of the status register. This is to make sure that cache error exception does not occur while the caches are being accessed. Then the value of zero is loaded into TagHi and TagLo registers and is transferred to all the entries of

the tags (both I and D caches). This is done by issuing 'Index store tag' command. This forces good parity into the tags and also clears the V and CS fields of instruction and data cache tags, respectively. The index values are determined by the size of the caches and the line size. However, this operation does not initialize the W' and W field of data cache tags.

b) Once tags have been initialized and invalidated, the next step is to force good parity into the data part of the caches. In the case of Instruction cache, 'Fill_I' command is available which allows moving data directly from the main memory into the caches on line by line basis. This operation brings data into the cache, and writes corresponding valid parity.

c) For the case of data cache, where corresponding 'Fill' command is not available, different means are used to force good parity. First, the state of the cache is changed from 'invalid' to 'dirty exclusive' by issuing 'Create Dirty Exclusive' command. Once all entries in the data cache have been validated, known value (say zero) is stored to all locations of the cache. This stores valid data and forces good parity. This also sets the W' and W bits of the tag.

Both caches are in a valid state at this point. In order to invalidate these caches, 'Index invalidate' command for Instruction cache and 'Hit invalidate' command for data cache are issued. The index values are determined by cache size and cache line size of respective caches.

The next software module of the startup code which needs attention is 'initialize' routine, which is responsible for moving exception code to the DRAM and initializing TLB. This part of the code needs modification because, for the R4x00, some new exceptions have been defined and the vector locations have been changed. The code is moved using processor's block write mode. The code is read into the internal registers from EPROM as an uncached read, and then is stored into the cache using the store operation. Once the code is written to the D-cache, DRAMs are written by issuing 'Hit Writeback invalidate' cache commands.

The next important function of the module is initializing TLB. For the R3000, known values are stored in EntryHi and EntryLo registers. Then the index value of the entry is written to index register which is then shifted by 6 bits. At the end, the 'tlbwi' instruction is issued which writes the values in the EntryHi and EntryLo registers into the TLB entry.

TLB entries for R4x00 are different than R3000 entries. Values are written to EntryHi, EntryLo0 and EntryLo1 registers. Page sizes are set by writing to the page mask register. The index value is written into index register, and then the 'tlbwi' instruction is issued. Note that in the case of the R4x00, shifting the contents of the index register by 6 bits is not required.

COMPATIBILITY WITH THE R4600 ORION

At the conclusion of this paper, it is important to discuss the compatibility-related issues affiliated with the R4600. The processor is software-compatible with earlier generation R4x00 architecture processors, such as R4000 and R4400. The processor provides some additional features which enhance the performance of existing R4x00 designs. The points to note are:

Unlike the R4400, the R4600 has 2-way associative caches with line size fixed to 32-bytes. Operations which uses index values need to access set 0 by setting virtual address bit 13 equal to zero and set 1 with virtual address bit 13 equal to 1.

In addition to new cache organization, a WAIT instruction has been added for power management.

CONCLUSIONS

This paper has presented the modifications required when an R3000 code is ported to an R4x00 platform. Specifically, only the kernel mode code needs modification, whereas the user application can remain the same. The source code of IDT/sim has been used as an example to explain minimal modifications. This allows the design upgrade within a minimal amount of time.

Integrated Device Technology, Inc.

R4600 BUS ERROR HANDLING

TECHNICAL NOTE TN-15

By Peter N. Glaskowsky

INTRODUCTION

The Fast Restart feature of the IDT R4600 RISC CPU has changed the way that databus error detection is handled. This tech note will explain the change and describe ways to implement robust error detection in R4600-based systems.

"FAST RESTART" IN THE R4600

When the R4600 experiences a data cache miss in the data fetch stage of the execution pipeline, a stall condition stops the pipeline until the required data is supplied from the external system interface. Since the R4600 uses only sub-block ordering for block reads, the data requested by the fetch will always be returned in the first doubleword in the read response from the external agent.

As discussed in the R4600 Hardware User's Manual on page 3-7 under "Stall Conditions", the R4600 will resume processing as soon as this first doubleword is returned. This behavior is new to the R4600. The R4000 and R4400 wait until the entire cache line is refilled before the pipeline is restarted.

BUS ERROR DETECTION IN THE R4600

The R4600 takes a bus error exception if the active-low Good Data Indication bit, SysCmd(5), is set to 1 when the first data element is returned in a block read response. If no error is reported, the execution pipeline is restarted and the remainder of the cache line is loaded while processing continues.

If the external agent were to report an error with these later data elements, the error could not be correctly associated with the instruction which caused it, and a proper Bus Error exception could not be generated. For this reason, the R4600 does not evaluate the Good Data Indication bit on data elements after the first data element in block read responses.

ENSURING DATA INTEGRITY

Systems which implement parity on the SysADC bus

Block read response data is loaded directly into a cache line, along with the parity from the SysADC bus. If the external agent detects an error in the first data element in the block, it may optionally set the Good Data Indication bit to 1 to generate a Bus Error exception.

If the Good Data Indication bit is set to 0, the Data Checking Enable bit (SysCmd(4)) is also set to 0, and the first data element contains a parity error (i.e., the external agent does not signal the error to the processor), the processor will take a Cache Error exception and indicate that the error came from the SysAD bus by setting bit 26 of the CacheErr register.

If any of the subsequent data elements in the block read response contain parity errors, the bad parity will be stored in the cache, and later accesses to them will generate a Cache Error exception. The exception handler can examine the CacheErr and ErrorEPC registers to determine where the error occurred.

Systems without parity on the SysADC bus

If external logic is used to test for data errors and parity is not passed through to the R4600, the Data Checking Enable bit must be set to 1 during read responses. Parity will be generated internally by the R4600, stored in the cache along with the read response data and checked normally during processor operation. This allows the R4600 to use parity on the cache even if parity is not used on the external interface.

An error in the first data element in a block read response may be signalled to the R4600 by setting the Good Data Indication bit to 1. Errors in subsequent data elements must be signalled using a different mechanism. The Non-Maskable Interrupt (NMI) exception is recommended, since it cannot be masked. However, it is not normally possible to continue program execution after servicing an NMI.

It would also be possible for an external agent to retry a main memory read if a parity error is detected, and submit the data along with the ValidIn* control signal only if the data is error-free. This would add significant complexity to the state machine. It will generally be easier to connect external parity to the R4600 SysADC bus.

Systems without parity on main memory

In systems where parity is not provided on main memory, the Data Checking Enable bit must be set to 1 and the Good Data Indication bit must be set to 0 during read responses. The R4600 will generate parity internally and store the parity and data in the primary cache. Parity will be tested each time the cache is read, ensuring cache integrity even though the integrity of externally supplied data is not testable.

COMMAND BUS PARITY

The R4600 does not check parity on the SysCmd bus. The SysCmdP signal is not checked when the system interface is in the slave state (for example, during read responses).

Integrated Device Technology, Inc.

32- AND 64-BIT OPERATION OF THE IDT79R4600™

TECHNICAL NOTE TN-21

By Phil Bourekas

INTRODUCTION

The IDT79 R4600™ (Orion™) is a true 64-bit microprocessor; the internal registers, data paths, and arithmetic units are all 64-bits, and the processor directly implements various 64-bit operations (such as arithmetic operations) as single cycle operations.

The R4600 allows great flexibility in how this capability is used, insuring both compatibility with 32-bit applications, and insuring that new applications can easily take advantage of the higher bandwidth and throughput available from 64-bit operations. This technical note is intended to provide an introductory look at how the Orion, and its support tools, accomplish these objectives.

THE R4600 64-BIT ARCHITECTURE

In all respects, the R4600 is a 64-bit microprocessor. Consider:

- The register file of the processor contains 32 64-bit wide registers, for the most part used orthogonally by the instruction set.
- The functional units, including logical and arithmetic functions, multiply and divide, and memory management, operate directly on 64-bit datums.
- The R4600 MMU manages full 64-bit virtual addresses.
- The R4600 directly moves 64-bit datums between its internal caches and its internal execution core in a single cycle; that is, the cache data path is 64-bits wide.

Although not strictly a pre-requisite for a "64-bit processor", the system interface is a 64-bit wide multiplexed address/data bus. However, to facilitate migration of existing software, the R4600 64-bit architecture is directly compatible with 32-bit operations. Also note that for the R4600, memory remains "byte addressable". Load and store operations can specify the operand size as 8-, 16-, 32-, or 64-bit in size.

NUMERIC COMPATIBILITY BETWEEN 32-BIT AND 64-BIT OPERATION

The R4600 operates seamlessly with various 32-bit applications. The MIPS architecture insures this interoperability by defining that 32-bit operations will sign extend their results to fill 64-bit registers. Thus, using a 2's complement numeric representation, the value of the results appears identical when viewed as either a 64-bit or a 32-bit value.

VIRTUAL ADDRESS MODE

In addition to being able to directly utilize 32- and/or 64-bit numeric values, the R4600 can directly support 32-bit or 64-bit addressing. The mechanisms provided allow varying OS and applications strategies, including 32- or 64-bit applications running on a 64-bit operating system. To facilitate this operation, the R4600 offers two virtual addressing modes:

- 32-bit virtual addressing mode. In this case, all virtual addresses are considered to be 32-bit values. This affects the operation of the address translation unit (the MMU), and also affects the selection of the TLB exception vectors. Specifically, virtual addresses whose upper 32-bits are not equal to all "0" or all "1" are considered invalid addresses and will cause an address error.
- 32-bit virtual addressing mode is only available to user and supervisor tasks: the kernel always executes in 64-bit virtual addressing mode. This mode is selected via the UX (user mode) and SX (supervisor mode) bits of the CP0 status register.

When operating in 32-bit virtual addressing mode:

- 64-bit operations (the MIPS-3 instruction set) are invalid. This prevents software from generating pointer values larger than 32-bits.
- the "Regular" TLB refill exception vector is used. By separating the 32-bit and 64-bit vector locations, the OS is able to quickly perform software TLB refill without worrying about the operating mode of the task.
- physical addresses remain 36-bit.

As noted above, the kernel always operates in 64-bit virtual addressing mode. However, the R4600 does support an operating bit (KX) which enables the kernel to use the "regular", rather than extended, TLB refill exception vector. This was originally provided to enable existing R3000 OS and compiler support to be migrated cleanly to the R4xxx architecture; this bit may be used to implement operating systems which only provide mapping support for 32-bit virtual addresses.

- 64-bit virtual addressing mode. In this case, all virtual addresses are considered to be 64-bit values (Note, however, that the Orion only maps 40 bits of the 64-bit address space. Mappable virtual addresses whose upper 24-bits are not all "0" or all "1" are considered invalid). In 64-bit virtual addressing mode:
- 64-bit operations (the MIPS-3 instruction set) are available.
- the "Extended" TLB refill exception vector is used.
- Physical addresses remain 36-bit.

COMPILER SWITCHES

Most R4600 compilers offer similar flexibility in their treatment of 64-bit vs. 32-bit data. For example, IDT/C™ offers switches to selectively manage:

- whether the MIPS-3 instruction set extensions will be generated by the compiler (compatible with the choice of UX/SX described above).
- the treatment of datum size by the compiler. For ex-

ample, depending on switches selected, integers could be 32-bit or 64-bit values, and types such as "long" can be defined to be the appropriate width.

The IDT/c compiler supports various switches, including the following:

- MIPS-3 switch. If this switch is activated, the compiler will generate MIPS-3 instructions where appropriate. In this case, the full register width of the integer and CPU units is used (64-bits); types "int" and "long" are 32-bits; addresses or pointers are 32-bits; and the type "long long" specifies a 64-bit datum.
- mlong64. This switch (used with "-mips3") makes all "long" variables and pointers 64-bits, but integers remain 32-bits.
- mint64. This switch makes all variables of type "int", "long", and "long long" to be 64-bits, and activates the "-mlong64" switch automatically.

With this flexibility, the programmer can implement a variety of schemes, including applications which use 32-bit pointers referencing 64-bit datums; 64-bit pointers and datums; 32-bit pointers and datums, etc. This capability enables systems to take advantage of the bandwidth available from the 64-bit processor without rewriting the entire application (and using up memory resources).

SUMMARY

The R4600 is a true 64-bit microprocessor. However, the processor architects have implemented it in a fashion that allows 32-bit applications to readily take advantage of the 100+ MIPS capability of the device, without forcing complicated changes to the application or the operating system. Although the Orion does NOT implement a "32-bit mode", the use of 32-bit *virtual addressing mode*, along with the technique of sign-extending 32-bit values so that they are compatible with 64-bit operation, allows software to readily take advantage of the performance available in the Orion.

Integrated Device Technology, Inc.

R4600™ CACHE INITIALIZATION

TECHNICAL NOTE TN-22

By Russell Cummings

INTRODUCTION

The IDT79R4600™ Orion™ RISC microprocessor is a full 64-bit architecture that brings desktop-like performance at a fraction of the price. It is pin-compatible with its predecessor the R4400PC™ and uses the same instruction set. It provides complete upward application-software compatibility with the IDT RISController™ family. The R4600 maximizes the performance by implementing large on-chip two-way set associative caches, a five stage pipeline with fewer stalls and an early restart mechanism for cache refills during data cache misses.

This Technical Note addresses the initialization of both the instruction and data caches. It includes the basic assembly code needed to do this task and also addresses any issues pertaining to cache initialization.

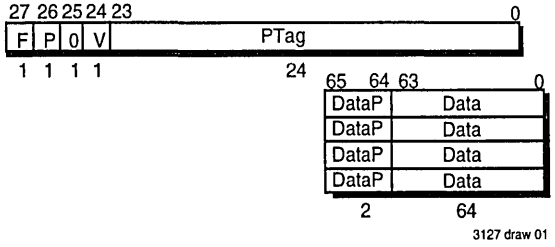
THE R4600 CACHES

The R4600 contains two 16KByte caches, one for instructions and the other for data. Both caches are two-way set associative with 8 word (32-byte) line sizes. The caches are virtually indexed (part of the virtual address is used to index into the cache array) and physically tagged (the tag in the array is compared with the physical address to determine a hit or miss).

ORGANIZATION OF THE PRIMARY INSTRUCTION CACHE (I-CACHE)

Each line of primary I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 28-bit tag that contains a 24-bit physical address, a single valid bit, a reserved bit, a single parity bit and the FIFO replacement bit. Word parity is used on I-cache data.

Figure 1 shows the format of a primary I-cache line.



PTag	Physical tag (bits 35:12 of the physical address)
V	Valid bit
F	FIFO Replacement Bit. Complemented on refill.
P	Even parity for the PTag and V fields
DataP	Even parity; 1 parity bit per 32-bit word of data
Data	Cache data

Figure 1: Instruction Cache Line Format

ORGANIZATION OF THE PRIMARY DATA CACHE (D-CACHE)

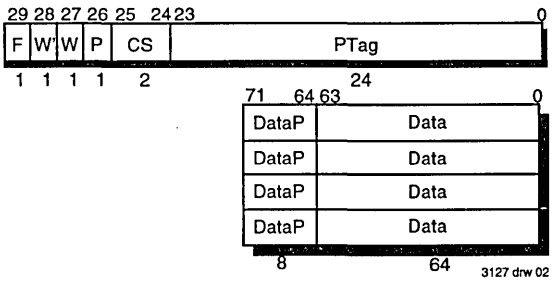


Figure 2: Data Cache Line Format

F	FIFO Replacement Bit
W'	Even parity for the write-back bit
W	Write-back bit (set if cache line has been written)
P	Even parity for the PTag and CS fields
CS	Primary cache state: 0 = Invalid, 1 = Shared, 2 = Clean Exclusive, 3 = Dirty Exclusive
PTag	Physical tag (bits 35:12 of the physical address)
DataP	Even parity for the data; 1-bit per byte
Data	Cache data

Each line of primary D-cache data has an associated 30-bit tag that contains a 24-bit physical address, 2-bit cache line state, a write-back bit, a parity bit for the physical address and cache state fields, a parity bit for the write-back bit and the FIFO replacement bit.

Figure 2 shows the format of a primary D-cache line.

CACHE INITIALIZATION

CACHE STATE DURING RESET

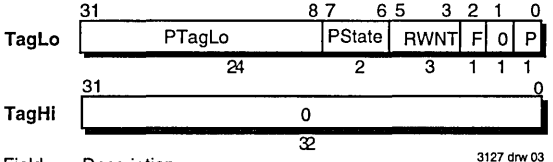
The contents of the primary caches are undefined at the end of the reset sequence. Not only are the tags in an undefined state but the data arrays are also undefined. It is therefore necessary to properly initialize both the TAG and Data arrays before the caches are used. Properly initialized means: 1) there is valid parity, tag data and array data and 2) the tag is in the invalid state. This task is further complicated by the 2-way set associativity of the caches in that the system designer must make sure that both sets are initialized correctly.

This was the reason that the CACHE instruction was defined. It allows the kernel to perform the tasks of cache initialization and maintenance. There are two basic types of CACHE instructions: indexed and hit. The indexed operations use part of the virtual address to specify a particular cache block (VA [12:5]) and VA[13] to specify a particular set. The "hit" operation accesses the specified cache as normal data

references and performs the specified operation if the cache block contains valid data with the specified physical address (a hit). If both sets are invalid or contain different addresses(a miss), no operation is performed.

PRIMARY DATA CACHE INITIALIZATION

It is in general much simpler to test the data cache than the instruction cache. One reason for this is that if the data cache read fails, the program can still continue where as an instruction cache failure can result in the program getting lost and not finding its way back to the correct code. The TagHi and TagLo registers are used for managing the tags. The format for the TagLo and TagHi registers is shown in figure 3.



Field	Description
PTagLo	Specifies the physical address bits 35:12
PState	Specifies the primary cache state
P	Specifies the primary tag even parity bit
F	The FIFO bit used to implement FIFO refill of the cache
RWNT	Read/Write bits required for Windows NT
0	Reserved. Must be written as 0; returns 0 when read

Figure 3: TagHi and TagLo Format

The basic procedure to initialize the data cache is to first turn off error checking (if not already off). Next, use the TagLo and TagHi registers along with the CACHE instruction to place the known good tag into the array (remember to get both sets). Set the TagLo and TagHi both to zero. Next, the base address and loop counter are setup. The base address is where known good (initialized) data is loaded from and the loop counter is an indicator of when the procedure is finished. Now, invalidate all the tags using the Index_Store_Tag CACHE instruction (again on both sets). Next, place known good data into the data array using a load word instruction (because both sets are now invalid, the result is block reads from memory for each load). Finally, re-invalidate the tags.

Example code is shown in Table 1.

```
1:
    li      r2, 0x8000_0000     /* Setup base address*/
    li      r25, 255            /* Setup loop counter to # lines*/
    mtc0    r0, C0_TAGLO        /*Setup TagLo to invalidate tags */
2:
    cache   0x9, 0x0(a0)        /* Index Store Tag - Set0 */
    cache   0x9, 0x2000(a0)     /* Index Store Tag - Set1 */
    lw      r0, 0x0(r2)         /* clear dirty bits and set data and parity */
    lw      r0, 0x2000(r2)      /* to known good values */
    cache   0x9, 0x0(a0)        /* Index Store Tag - Set0 */
    cache   0x9, 0x2000(a0)     /* Index Store Tag - Set1 */
    addu    r2, 0x20            /* increment address pointer */
    bgtz    r25, 1b             /* see if loop done */
    addi    r25, -1             /* decrement  loop counter */
```

Table 1: Data Cache Initialization

PRIMARY INSTRUCTION CACHE INITIALIZATION

The primary Instruction cache is initialized in a similar manner. Because there is no way for a load to place data into the I-cache data array directly, a CACHE operation is provided ("FILL_I") to allow for data to be placed in the I-cache data array from memory.

The basic procedure to initialize the I-cache is to first turn off error checking (if not already off). Next, use the TagLo and TagHi registers along with the CACHE instruction to place the known good tags into the array (remember to get both sets). Set the TagLo and TagHi both to zero. Next, the base address and loop counter are initialized. The base address is where known good (initialized) data will be loaded from and the loop counter indicates when the procedure is done. Now, invalidate all the tags using the Index_Store_Tag CACHE instruction (again on both sets). Next, place known good data into the data array using FILL_I CACHE instruction (because both sets are now invalid, the result is block reads from memory for each FILL_I instruction). Finally, re-invalidate the tags.

Example code is shown in Table 2.

```
1:
    mtc0    r0, C0_TAGLO        /* Setup TagLo to invalidate */
    li      r2, 0x8000_0000     /* Setup base address */
    li      r25, 255            /* setup loop counter to # lines*/
2:
    cache   0x8, 0x00(r2)       /* Index Store Tag, Set0 */
    cache   0x8, 0x2000(r2)     /* Index Store Tag, Set1 */
    cache   0x14, 0x00(r2)      /* fill Icache data from memory */
    cache   0x14, 0x2000(r2)    /* fill Icache data from memory */
    cache   0x8, 0x0(a0)        /* Index Store Tag - Set0 */
    cache   0x8, 0x2000(a0)     /* Index Store Tag - Set1 */
    addu    r2, 0x20            /* increment address pointer */
    bgtz    r25, 1b             /* see if loop done */
    addi    r25, -1             /* decrement  loop counter */
```

Table 2: Instruction Cache Initialization

CONCLUSION

The R4600 is a high-performance CPU and achieves this through several methods, one is using 2-way set associative caches. The use of set associative cache adds some steps to the initialization of the cache but help is provided with the CACHE instructions. Also, to avoid problems, one must make sure that the caches are correctly initialized in both the tag and the data arrays with valid data in the tag, parity and data array and that the state of the line is set to invalid.

Integrated Device Technology, Inc.

IDT79R4600™/R4400™ "OUTSIDE-SPECS" DIFFERENCES

TECHNICAL NOTE TN-23

By Robert Napaa

INTRODUCTION

The IDT79R4600™ Orion™ RISC microprocessor is a full 64-bit architecture that brings non-desktop performance at a fraction of the price. It incorporates advanced power management techniques to lower the peak and typical power consumptions. It features an impressive performance at a relatively low power with about 35 SPECint92/Watt. With its low power and high performance, the R4600 supports a large base of processor applications, including 32-bit Windows™ NT desktop or notebook systems. It is well suited for a multitude of embedded applications including laser printers, color printers, color X-terminals, routers, data communications, disk arrays and set-top cable boxes.

The R4600 is pin compatible with its predecessor the R4400PC™ and uses the same instruction set. It provides complete upward application-software compatibility with the IDT RISController™ family. The R4600 maximizes the performance by implementing large on-chip two-way set associative caches, a five stage pipeline with fewer stalls and an early restart mechanism for cache refills during data cache misses. The power saving is implemented through an intelligent power management scheme which turns-off the power from the currently unused sections of the part. A standby mode is also available through software control which shuts down the internal clocks and freezes the pipeline, thus reducing the consumed power drastically.

This Technical Note addresses the differences between the R4600 and the R4400PC. It mainly highlights the subtle differences that might cause system incompatibilities when swapping the two parts.

NOT AN R4400PC CLONE

The IDT R4600 is a independent design that implements the MIPS-III Instruction Set Architecture (ISA). It is not an R4400PC clone; it doesn't use the R4400PC design data base or internal architecture. It is a complete new design that implements major internal architectural differences to achieve higher performance over the R4400PC with a smaller die area and a lower power consumption. It also implements additional bus interface protocols to speed the main memory interface and improve the overall performance of the system. The core of the R4600 is fully static and implements several power management techniques to reduce the overall system power consumption. The integer and the floating-point execution units of the R4600 share some of the internal resources (such as the multiplier and the divider) to reduce the die size and the power requirements.

The R4600 is designed to maintain full compatibility (hardware and software) with the R4400PC but with a better execution engine and bus interface protocol to enhance the overall system performance.

COMPATIBILITY WITH THE R4400PC

The R4600 is plug, pin and software compatible with the R4400PC. This compatibility is guaranteed for systems that are designed to the specifications of the R4400PC data sheet ("Within-Specs"). For such systems, the R4600 is a one-to-one replacement of the R4400PC. Software applications should execute without modifications and the hardware platform should be used as is. However, even for systems designed "Within-Specs", there are some differences between the R4600 and the R4400PC that the system designer and/or the code developer must be aware off. These differences are well documented in the data sheet of the R4600. These differences are mainly due to the different microarchitecture of the two devices, their implementation and their behavior.

In systems that violate the R4400PC or the R4600 data sheet specifications ("Outside-Specs"), the behavior of the two parts might be (and most of the time will be) completely different. For example, systems which rely on empirical observations of the R4400PC behavior might not run properly with an R4600. These situations create serious systems incompatibilities for the system designers and/or the code developer.

DIFFERENCES "WITHIN-SPECS"

The differences "Within-Specs" between the two parts are primarily due to the different microarchitectures. All these "Within-Specs" differences are well documented in the data sheet of the R4600. These differences are not considered "bugs" and will not be modified. Systems that are designed to support both parts interchangeably must take these differences into account. The hardware platform has to be designed in a way to take advantage of the additional bus capabilities of the R4600 for example. Similarly, the software applications have to be able to take advantage of the two-way set associative primary caches and a shorter pipeline. With these differences in mind, it is possible to design systems that support both parts seamlessly.

Architecture

There are several architectural differences between the R4600 and the R4400PC. These architectural differences enable the R4600 to improve the overall system performance by 20% to 30% compared to R4400PC based systems. The internal architectural differences include different implementation of the primary caches, the pipeline, the Co-Processor 0 and the Co-Processor 1. These implementations allow the R4600 to achieve a higher performance on the same applica-

tions. Furthermore, the R4600 bus interface unit is designed to maximize the bus utilization through the added bus write protocols. These new protocols increase the overall system performance without relying on re-compilation.

Implementation

There are very few differences (between the R4600 and the R4400PC) in the implementation of the MIPS-III ISA set. Mainly, the R4600 conforms to the MIPS specifications regarding the timing hazards when accessing some of the Co-processor 0 registers.

Behavior

The R4600 matches the behavior of the R4400PC even when this behavior is different from the published MIPS architectural specifications.

DIFFERENCES "OUTSIDE-SPECS"

The R4600 is not an R4400PC clone. The internal logic of the R4600 is completely different from that of the R4400PC. This is mainly due to the differences in the microarchitecture between the two parts and the added bus protocols on the R4600. This means that outside the specifications of the data sheet ("Outside_Specs") of both parts, the behavior of the R4600 can be completely different from that of the R4400PC. This different behavior might cause serious systems incompatibilities when swapping the two parts. Systems designers must be very careful not to violate the data sheet specifications of either part to ensure total compatibility and avoid unpleasant surprises.

Definition of "Outside-Specs"

The definition of "Outside-Specs" refers mostly to the violations of the data sheet specifications of the R4600 or the R4400PC. The most common cases are the violation of the setup and/or hold time of the data or control signals. Another common one is the misinterpretation of the timing diagrams. There are other more subtle violations that might be harder to detect. For example asserting the control signals to the CPU (such as ~ValidIn) for more than the required time can cause the state machine of the bus interface unit to lose synchronization. Further, any "between-the-lines" interpretation of the R4400PC or the R4600 data sheets can also become a violation of the specs. This list is not all inclusive and should be used a guideline to possible violations or misinterpretations of the data sheets of either part.

Why different behavior?

The internal logic of the two parts is completely different because of the architectural differences between them. Such architectural differences include different primary caches, pipeline, Co-Processor 0 and Co-Processor 1. Mainly, the two-way set-associative primary caches, support for data streaming, and the five stage pipeline on the R4600 require a total different set of internal logic and state machine. Further, the additional bus write protocols, such as pipeline write and write re-issue require more internal logic than the bus interface unit of the R4400PC.

The two parts will, most probably, react differently to erroneous stimulus and to violations of the data sheet specifications. The scope of the reaction of each part is not guaranteed and depends on the internal state of the logic.

The internal logic architecture of the R4600 is designed to be compatible with the specifications of the R4400PC data sheet but will respond differently to deviations from these specifications.

First Symptoms

The very first symptoms appear when the two parts are swapped in a system and the system doesn't work. This usually indicates a "Within-Specs" incompatibility rather than an "Outside_Specs" one. Such problems are usually easily traced to improper S/W initialization of the internal register or caches. Similarly, incorrect reset vectors or the wrong usage of any part can cause this type of problems. These problems are easily fixed when using the proper initialization sequence or the proper reset vector and so on.

On the other hand, "Outside-Specs" problems are much harder to trace, to determine and to solve. They are usually very time consuming and very frustrating. The usual scenario is that the system works fine with either CPU. However, the system might crash from time to time when using one part and not the other. Most of the time this is a clear indication that there is a data sheet violation somewhere in the system. It could be hardware or software. This behavior of the system is in line with the expectation that two parts will respond differently to erroneous stimulus. One part could cope perfectly with violation of the specs while the internal state machine of the other is being driven to an unknown, undefined or undesired state. Such problems are the hardest to find and usually require additional searching and experimenting to be resolved.

CONCLUSION

The R4600 is plug and software compatible with the R4400PC "Within-Specs". However, even "Within-Specs", there are some differences that are well documented in the data sheet of the R4600 and the errata of the different revisions of the device. These differences are mainly in the microarchitecture of the part. System designers and software developers must be aware off these differences and take them into considerations when designing systems that will support both parts. These differences are not considered "bugs" and will not be modified.

On the other hand, if the specifications of the data sheet are violated, the behavior of the two parts will be different, creating incompatibilities when swapping the two parts. System designers and software developers must avoid violating the specs to ensure a proper design and minimize the time discovering the incompatible modes between the two parts.

Integrated Device Technology, Inc.

HEATSINK ISSUES FOR MICROPROCESSOR PRODUCTS WITH INTEGRAL SLUG

TECHNICAL NOTE TN-25

SUMMARY:

Designing in highly integrated, high clock rate microprocessors such as the R4600 requires careful consideration of thermal management. To ease this burden IDT utilizes an integral heat slug technology in its pin grid array packages. This heat slug is made of thermally conductive material such as CuW which is embedded into the ceramic base of the package. In the process of selecting a heat sink, designers should be aware of some of the constraints associated with industry standard PGA's to avoid potential mechanical problems while affixing heat sinks to the integral slug.

Heat slug is at Vcc potential:

The Integral heat sink used in 179, 161, and 447 PGA packages allows maximum heat transfer (minimum theta jc) from the back of the die to the external surface of the heat sink. This puts the heat slug at Vcc potential which must be taken into consideration when selecting external finned heat sinks and EMI shields.

Clips and EMI shields can damage package edge:

Industry standard ceramic package construction techniques used by leading package vendors for electrolytic nickel and gold plating of internal traces leaves microscopically fine pattern of electrically active metallic contacts on all four edges of the ceramic package body. Sufficient abrasion of the package edges can result in unintentional electrical connection between the internal traces of the package and any metallic material touching the package edge; thus, the use of a metallic clip to attach an external heat sink to the package or a heat sink design that contacts the edges of the package is not recommended and must be taken into consideration for heat sink selection and attachment method.

Recommendation:

Avoid use of heatsink clips or EMI shields which contact the PGA package edge. Consult your heatsink vendor about the proper heat sink for you application.

Integrated Device Technology, Inc.

ORION™ SYSAD OUTPUT TIMING ISSUES

TECHNICAL NOTE TN-26

by Robert Napaa

INTRODUCTION

The IDT Orion™ Family of 64-bit microprocessors supports a wide variety of processor-based applications, including 32-bit Windows NT desktop or notebook systems and embedded systems. The Orion™ family includes several members such as the R4600, the R4700, and the R4650. New products are continuously under development and introduced regularly.

This Technical Note focuses on the SysAD output timing parameters for the Orion™ family which appear in the "AC Electrical Characteristics System Interface Parameters" tables in the data sheets for these products.

BACKGROUND

The data sheets for the different devices in the Orion™ Family list the system interface AC parameters in the "System Interface Parameters" table. This table lists the AC parameters that specify the output timing for the data movement from the CPU to the external memory (t_{DM} and t_{DO}). It also lists the AC parameters that specify the input timing for the data movement from the main memory to the CPU (t_{DS} and t_{DH}). The t_{DS} parameter specifies the minimum data setup time that the system must guarantee before the rising edge of SClock for the CPU to sample the input data properly. Similarly, the t_{DH} parameter specifies the minimum data hold time that the system must guarantee after the rising edge of SClock to ensure that the CPU sampled the input data properly.

DATA OUTPUT TIMING PARAMETERS

There are two parameters, t_{DM} and t_{DO} that specify the AC parameters for the output signals (address and data) provided by the CPU.

Definition

The output signals (address and data) from the CPU become stable a minimum of t_{DM} ns and a maximum of t_{DO} ns after the rising edge of the Clock (the SClock in the case of the R4600/R4700 and the MasterClock in the case of the R4650). This drive-time is the sum of the maximum delay through the processor output drivers together with the maximum clock-to-Q delay of the processor output registers.

t_{DO}

t_{DO} specifies the maximum time it takes for the data issued from the CPU to reach valid signal levels (1.5V) after the rising edge of the Clock (SClock in the R4600/R4700 case and MasterClock in the R4650 case). During that time frame, the external system should not sample these lines because the voltage levels might chane before the final levels are reached. The t_{DO} parameter is specified by the maximum values in the "System Interface Parameters" table. The two values listed provide a range that corresponds to the levels of the output drivers strength programmed during the boot sequence of the CPU. Figure 1 illustrates the t_{DO} parameter.

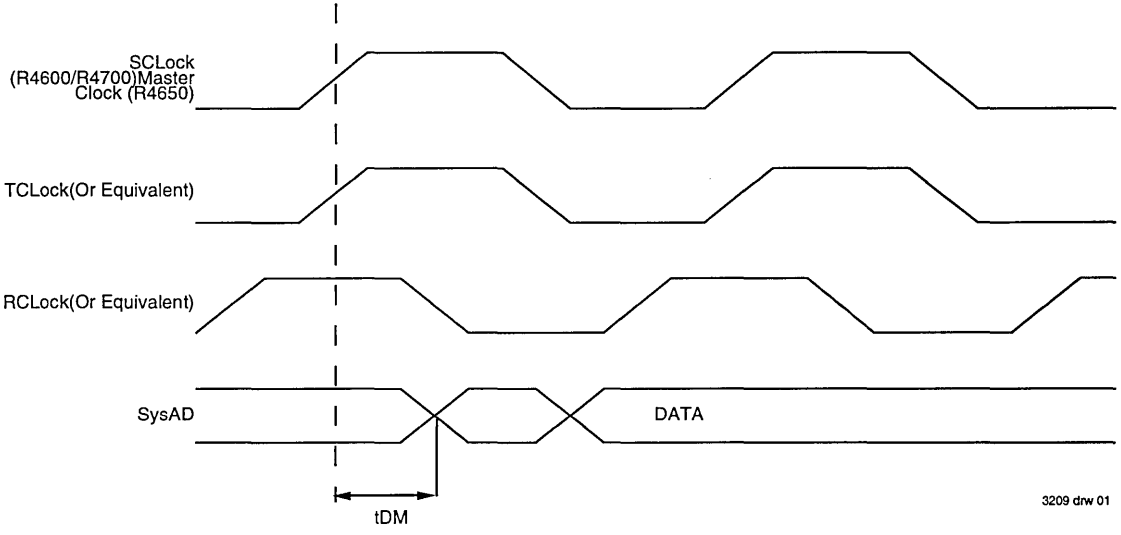


Figure 1. The t_{DO} Parameter

t_{DM}

t_{DM} specifies the minimum time it takes for the data issued from the CPU to reach valid voltage levels (1.5V) after the rising edge of the Clock (SClock in the R4600/R4700 case and MasterClock in the R4650 case). During that time frame, the voltage levels on these data lines might change before the final levels are reached. The t_{DM} parameter is specified by the minimum values in the "System Interface Parameters" table. The two values listed provide a range that corresponds to the levels of the output drivers strength programmed during the boot sequence of the CPU. Figure 2 illustrates the t_{DM} parameter.

The t_{DM} parameter should not be considered as data hold time from the CPU (there is no parameter that specifies the data hold time from the CPU to the system). The t_{DM} parameter specifies when the voltage levels have stabilized not when the CPU starts changing the data. This actually implies that the CPU could start changing the data earlier than the t_{DM} value.

Data Sheet Testing

The devices are tested to data sheet specifications prior to shipment. On the tester, only the t_{DO} parameter is measured and characterized for both the minimum and maximum values. The t_{DM} parameter is not measured and is only guaranteed by design.

Practical Considerations

For system designers, the parameter that should be taken into consideration is t_{DO} which specifies the maximum time before the data is valid from the CPU. The t_{DM} parameter in reality should not be used, since it specifies only the minimum time it might take the data to become valid. The t_{DM} parameter should not be used as the data hold time from the CPU.

When using the R4600/R4700 processors, the RClock should be used to sample the output signals from the processor. The RClock is leading SClock by 25% and thus offers the neccessary hold time for the external logic. When using the R4650 a similar clock to the RClock should be generated form the input clock distribution tree to sample the processor output signals. A detail explanation on this topic is available in the Application Note titled "Adapting an R4600 design to the R4650".

CONCLUSION

The "System Interface Parameters" tables in the data sheets for the Orion™ Family of microprocessors provide the necessary AC parameters for the interface with the CPU. The t_{DS} and t_{DH} parameters specify the timing for the data movement from the system into the CPU. The t_{DM} and t_{DO} parameters specify the data movement from the CPU to the system. The t_{DO} parameter specifies the maximum time for the data to become valid while the t_{DM} parameter specifies the minimum time. The t_{DM} parameter should not be treated as the data hold time from the CPU.

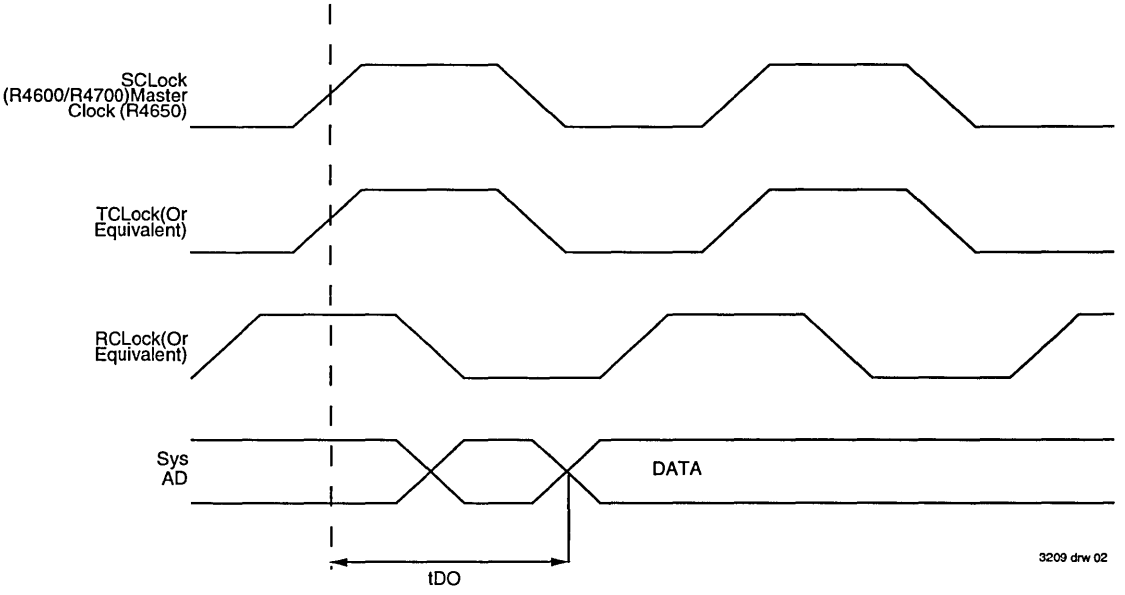


Figure 2. The t_{DM} Parameter

Integrated Device Technology, Inc.

USING THE IDT79R3051™ WITH THE HP16500 LOGIC ANALYZER

APPLICATION NOTE AN-93

by Andrew Ng

INTRODUCTION

The IDT79R3051™ RISController™ is a highly integrated, high-performance MIPS™ R3000™ instruction set compatible CPU that minimizes system cost and power consumption across a wide variety of embedded applications. The R3051 includes 4kB - 8kB of instruction cache, 2kB of data cache, 4-deep read and write buffers, on-chip DMA arbitration, a simple external bus interface, as well as the core R3000A execution engine — all in a single chip 84-pin package. However, in today's marketplace, the technical features of a microprocessor are not enough to guarantee a successful product. A new CPU such as the R3051 must also have a large base of software applications, and very importantly, adequate hardware and software development and debug tools. The R3051 family already has a large base of software applications and a large set of development tools because of its R3000A instruction set compatibility and also because of its widespread market acceptance. The use of just one of these tools, the IDT7RS364 Disassembler for the HP16500 Logic Analyzer will be explained here.

THE IDT7RS364 DISASSEMBLER AND THE HP16500 LOGIC ANALYZER

The IDT7RS364 Disassembler for the HP16500 Logic Analyzer is a useful tool meant to ease the task of debugging software run on R3000-based Target System Boards. Logic analyzers are inexpensive, general purpose debug tools which do not have the power of in-circuit emulators to actively control and simulate target system CPU and memory behavior. However, logic analyzers do provide a useful subset of in-circuit emulator debug capabilities by allowing an engineer to observe and analyze the digital circuit behavior of the target system.

The IDT7RS364 Disassembler consists of a software package that when loaded into the HP16500, pre-processes and formats the state trace listings of the Logic Analyzer. As shown in Figure 1, the HP16500 allows the engineer to capture the CPU's executed hex/binary machine opcodes in a typical Logic Analyzer State Trace Listing format. The user can set multilevel trace traps to capture the area of interest. As shown in Figure 2, with the addition of the IDT7RS364 Disassembler, the hex machine opcodes are automatically decoded and displayed in R3000 assembly code level mnemonic format. Thus the readability and usefulness of the state trace list display screen of the Logic Analyzer are greatly improved.

State/Timing E | Listing 1 | Invasm | Print | Run

Markers Off

Label> / Base>	ADDR Hex	DATA Hex	STAT Hex	Time Absolute
-6	1FC00000	0BF00088	0010	0 s
-5	1FC00004	00000000	0010	760 ns
-4	1FC00220	3C020010	0010	1.52 us
-3	1FC00224	40826000	0010	2.24 us
-2	1FC00228	40806800	0010	3.00 us
-1	1FC0022C	3C02A000	0010	3.76 us
0	1FC00230	3C08AAAA	0010	4.52 us
1	1FC00234	35085555	0010	5.24 us
2	1FC00238	AC480000	0010	6.00 us
3	1FC0023C	AC400004	0010	6.76 us
4	00000000	AAAA5555	0000	7.40 us
5	1FC00240	8C490000	0010	7.88 us
6	00000004	00000000	0000	8.52 us
7	1FC00244	00000000	0010	9.00 us
8	00000000	AAAA5555	0010	9.64 us
9	1FC00248	11280003	0010	10.32 us

2883 drw 01

Figure 1. R3051 Address/Data Trace List on a Logic Analyzer

State/Timing E | Listing 1 | Invasm | Print | Run

Markers Off

Label> / Base>	ADDR Hex	R3000 Mnemonic hex	STAT Hex	Time Absolute
-6	1FC00000	J 0x1FC00220	0010	0 s
-5	1FC00004	NOP	0010	760 ns
-4	1FC00220	LUI v0,0x0010	0010	1.52 us
-3	1FC00224	MTC0 v0,$12	0010	2.24 us
-2	1FC00228	MTC0 zero,$13	0010	3.00 us
-1	1FC0022C	LUI v0,0xA000	0010	3.76 us
0	1FC00230	LUI t0,0xAAAA	0010	4.52 us
1	1FC00234	ORI t0,t0,0x5555	0010	5.24 us
2	1FC00238	SW t0,0x0000(v0)	0010	6.00 us
3	1FC0023C	SW zero,0x0004(v0)	0010	6.76 us
4	00000000	STORE DATA 0xAAAA5555	0000	7.40 us
5	1FC00240	LW t1,0x0000(v0)	0010	7.88 us
6	00000004	STORE DATA 0x00000000	0000	8.52 us
7	1FC00244	NOP	0010	9.00 us
8	00000000	LOAD DATA 0xAAAA5555	0010	9.64 us
9	1FC00248	B 0x1FC00258	0010	10.32 us

2883 drw 02

Figure 2. R3051 Instruction Disassembly on the HP16500 Logic Analyzer

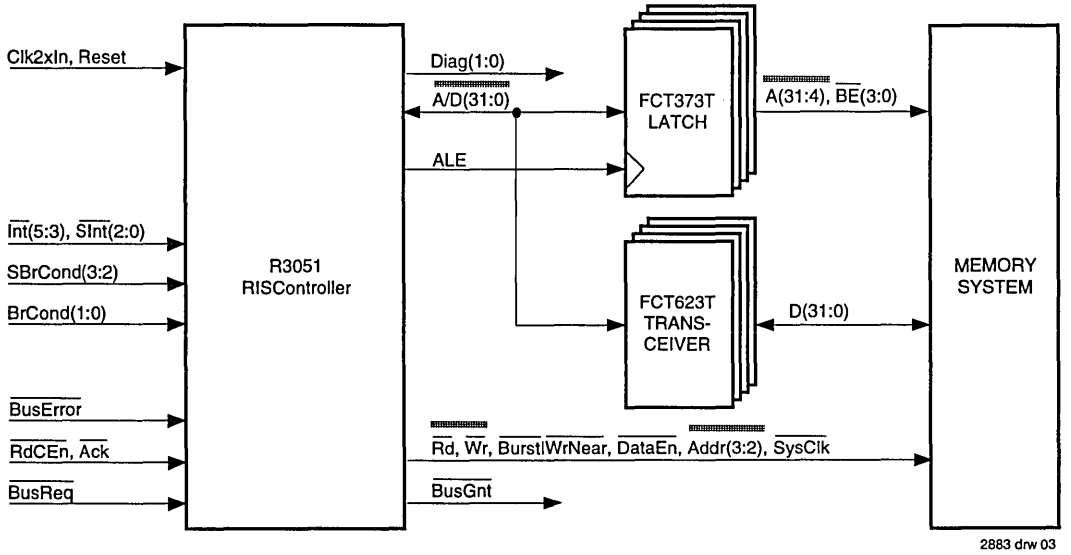


Figure 3. Typical R3051 System

Connecting the R3051 to the HP16500 Pod Sets

Before the Disassembler can be used, the correct connections between the R3051 and the HP16500 must be made. The Disassembler requires five 16-channel probe pod sets. The Disassembler expects that the Pod Probe connections follow its interface protocol so that the pre-processing can correctly interpret the address, data, and status lines. The Disassembler typically uses 32 Address lines, 32 Data lines, a Read line, and a Write line.

In the typical R3051 system as shown in Figure 3, the R3051's $\overline{Rd}$ output is used as the read line and the R3051's $\overline{Wr}$ output is used as the write line. The Disassembler uses the read and write signals as clocks to strobe the address and data into the Logic Analyzer. Since the top speed of the State traces on the HP16500 is 35 MHz and the fastest possible memory cycle is 2 clocks, the Disassembler can easily support 40 MHz R3051 CPUs and has a theoretical limitation of 70 MHz.

The Address lines can be gathered from the Address Latch outputs and Addr(3:2). Not all 32 address lines need to be attached, as the user can format the address line's MSB channel probes to not show up in the state trace listing if desired. In such a case, the user can use the extra channel probes for other purposes.

In general, Data lines can be gathered from the A/D bus. Some systems, with only one set of Data Transceivers, can gather the data from the memory side of the Data Transceivers in order to reduce A/D bus loading. The R3051 connections to the five HP16500 Channel Probe Pod sets are listed in Table 1.

The Disassembler has three status lines, Write, AccTyp(2) and AccTyp(0). The R3051's $\overline{Wr}$ output can be used as the write line so that the Disassembler can distinguish between a read and a write cycle. AccTyp(2) and AccTyp(0) are optional connections for cached code and in general should be grounded or at least left unconnected. The optional use of AccTyp(2)

POD chan	5 sig	POD chan	4 sig	POD chan	3 sig	POD chan	2 sig	POD chan	1 sig
15	X	15	A/D(31)	15	A/D(15)	15	A(31)	15	A(15)
14	X	14	A/D(30)	14	A/D(14)	14	A(30)	14	A(14)
13	X	13	A/D(29)	13	A/D(13)	13	A(29)	13	A(13)
12	Gnd	12	A/D(28)	12	A/D(12)	12	A(28)	12	A(12)
11	X	11	A/D(27)	11	A/D(11)	11	A(27)	11	A(11)
10	Note 2	10	A/D(26)	10	A/D(10)	10	A(26)	10	A(10)
9	X	9	A/D(25)	9	A/D(9)	9	A(25)	9	A(9)
8	X	8	A/D(24)	8	A/D(8)	8	A(24)	8	A(8)
7	X	7	A/D(23)	7	A/D(7)	7	A(23)	7	A(7)
6	X	6	A/D(22)	6	A/D(6)	6	A(22)	6	A(6)
5	X	5	A/D(21)	5	A/D(5)	5	A(21)	5	A(5)
4	$\overline{Wr}$	4	A/D(20)	4	A/D(4)	4	A(20)	4	A(4)
3	X	3	A/D(19)	3	A/D(3)	3	A(19)	3	Addr(3)
2	X	2	A/D(18)	2	A/D(2)	2	A(18)	2	Addr(2)
1	X	1	A/D(17)	1	A/D(1)	1	A(17)	1	Gnd
0	X	0	A/D(16)	0	A/D(0)	0	A(16)	0	Gnd
NClk		MClk	$\overline{Rd}$	LClk		KClk		JClk	$\overline{Wr}$

2883 tbl 01

Table 1. R3051 Default Pod Connections on the HP16500 Logic Analyzer

NOTES:
1. Master Clock Format: J↑ + M↑
2. POD5(12) is AccTyp(2) and POD5(10) is AccTyp(0). If AccTyp(2) is grounded then AccTyp(0) is not used by the Disassembler and can be used for other purposes. See text for further explanation.
3. A(31:4) are connected to the Address Latch outputs. The rest of the signals are connected to R3051 outputs. X's denote unused probes that can be assigned by the user.

and AccTyp(0) will be explained in more detail in the Cached Code/Data section. The 16-channel status pod has 13 unused channels that can be used to display other signals, e.g., the Byte Enables.

To a limited extent, the default ordering of the channel probe connections can be changed by the user. The relative ordering of the bits must still occur from MSB to LSB for the address/data/status bus labels such that the Pod Number and Channel Numbers go from MSB to LSB. An example of reformatting the Pod interface is shown in Table 2 and Figure 4. The example in Table 2 and Figure 4 also demonstrates the use of the HP16500's demultiplexed clock feature. When using the demultiplexed clock, the address and data lines can use the same probes. This allows both the address and data to be taken from the multiplexed A/D(31:0) bus. The address is slave-clocked with ALE and the data is master-clocked with $\overline{Wr}$ or $\overline{Rd}$. When using two clocks, only the 8 LSB probes on each pod can be used since the channels are internally multiplexed by the HP16500. Demultiplexed clocking is limited to 50 nsec master to slave clock recovery, which limits its use to 25 MHz CPU systems.

The HP16500 allows an extensive number of multi-level traps and triggers so that the code trace for the area of interest can be found. Care should be taken when setting up trigger conditions. Sometimes when in the trace/trigger menu, the Disassembler format in the data field trigger condition can conceal a trap condition. Changing the Disassembler format temporarily to hex format while in the trigger menu can prevent such confusion.

POD chan	5 sig	POD chan	4 sig	POD chan	3 sig	POD chan	2 sig	POD chan	1 sig
15		15		15		15		15	X
14		14		14		14		14	X
13		13		13		13		13	X
12		12		12		12		12	Gnd
11		11		11		11		11	X
10		10		10		10		10	Note 3
9		9		9		9		9	X
8		8		8		8		8	X
7	A/D(31)	7	A/D(23)	7	A/D(15)	7	A/D(7)	7	X
6	A/D(30)	6	A/D(22)	6	A/D(14)	6	A/D(6)	6	X
5	A/D(29)	5	A/D(21)	5	A/D(13)	5	A/D(5)	5	X
4	A/D(28)	4	A/D(20)	4	A/D(12)	4	A/D(4)	4	Wr
3	A/D(27)	3	A/D(19)	3	A/D(11)	3	A/D(3)	3	Addr(3)
2	A/D(26)	2	A/D(18)	2	A/D(10)	2	A/D(2)	2	Addr(2)
1	A/D(25)	1	A/D(17)	1	A/D(9)	1	A/D(1)	1	Gnd
0	A/D(24)	0	A/D(16)	0	A/D(8)	0	A/D(0)	0	Gnd
NClk		MClk	$\overline{Rd}$	LClk		KClk	ALE	JClk	$\overline{Wr}$

2883 tbl 02

Table 2. Example of Reformatted Pod Connections

NOTES:
1. Master Clock Format: J↑+M↑
2. Slave Clock Format: KØ
3. POD5(12) is AccTyp(2) and POD5(10) is AccTyp(0). If AccTyp(2) is grounded then AccTyp(0) is not used by the Disassembler and can be used for other purposes. See text for further explanation.
4. On Master/Slave Pods, only the 8 LSB probes are actually connected. E.g., A/D(23:16) is connected to Pod4(7:0).
5. X's denote unused probes that can be assigned by the user.

```
State/Timing          Format

          Master Clock        Slave Clock
            J ↑+ M ↑              K ↓

Pods          Pod 5               Pod 4               Pod 3               Pod 2               Pod 1
          Master | Slave      Master | Slave      Master | Slave      Master | Slave        Clock
Label   7 .... 07 .... 0   7 .... 07 .... 0   7 .... 07 .... 0   7 .... 07 .... 0   7 .... 07 .... 0
-------------------------------------------------------------------------------------------------------
ADDR      ........********    ........********    ........********    ........****....    ............****
DATA      ********........    ********........    ********........    ********........    ................
STAT      ................    ................    ................    ................    ****************
```

2883 drw 04

Figure 4. Example of Reformatted Pod Format

When Running with Cached Code/Data

All Logic Analyzers and Disassemblers can only capture external CPU memory accesses. Since the R3051 is capable of running code and accessing data in its internal caches, such accesses are not seen by the external memory system. Thus in order for the Disassembler to accurately reflect the complete instruction/data flow, the R3051 must be run uncached.

As the target system becomes more and more functional, it becomes necessary to begin running cached code and data. Running cached code/data will affect the Disassembler's accuracy in the following ways:

Cached Instructions —
1. Instruction fetch i-cache hits are not seen.
2. Only the last word of a cachable 4-word burst instruction i-cache miss will be seen.

Cached Data Loads —
1. Data load d-cache hits are not seen.
2. Only the last word of a cachable 4-word data block refill d-cache miss will be seen.
3. If the load instruction was an i-cache hit (not seen) then the associated data fetch if seen will be listed as an instruction. The data fetch is assumed to be the second (due to pipelining) read cycle after the load instruction.

Cached Data Stores —
1. Data stores are handled correctly, since the R3051 maintains a write-through cache policy which ALWAYS updates main memory as well as the d-cache.
2. Because the R3051 has a 4-word deep write buffer, a data store may or may not occur on the second (due to pipelining) memory cycle following its instruction fetch. Multiple stores are always handled in the proper FIFO order, but each store may be interspersed with later instruction fetches.

Other than running the software uncached, the following less intrusive methods may be used to help interpret cached code/data:

1. Use the R3051's testability mode to invoke the Force I-Cache Miss Mode. This will put all instruction fetches onto the external main memory interface so that the logic analyzer can see all of them. However, forced i-cache misses may or may not be 4-word burst reads.

   In general, 4-word burst reads can be displayed properly if a more complex read strobe is formatted:

   J clock: $\overline{\text{Ack}}$ == LOW
   M clock: $\overline{\text{RdCEn}}$ == LOW
   N clock: $\overline{\text{SysClk}}$ == positive edge triggered

   The HP16500 OR's level conditions together, OR's edge conditions together and AND's level conditions with edge conditions. Thus the above strobe clocks the state when:

   ($\overline{\text{SysClk}}$ == ≠) AND [($\overline{\text{Ack}}$ == 0) OR ($\overline{\text{RdCEn}}$ == 0)]

   This example clock setup is only applicable to systems that happen to bring Ack low at the same time RdCEn is low on 4-word burst reads or don't bring Ack low on 4-word burst reads. Also 1/2 clock margin on the memory read access time is necessary in this example. Thus depending on the particular system design, variants of RdCEn, Ack, and SysClk can be combined or temporarily modified to create a 4-word read strobe and a write strobe.

2. Latch the R3051's Diag(1:0) outputs with ALE. On external main memory reads, if LatchedDiag(1) == 1 then the fetch is cachable and can be used as an indication that the state trace entry should be interpreted judiciously. When LatchedDiag(1) == 1, LatchedDiag(0) == 1 indicates a cachable instruction fetch and LatchedDiag(0) == 0 indicates a cachable data load.

LatchedDiag(1:0) are the R3051's equivalents of the R3000's AccTyp(2) and AccTyp(0). As such they can be connected to the Disassembler's AccTyp(2) and AccTyp(0) probes. This allows the Disassembler to differentiate between cached instructions and data so that they can be displayed properly. However, AccTyp(2) and Diag(1) are undefined for writes, e.g., when the write buffer is full or on partial word stores. So if the AccTyp(2) probe is used, in order for the Disassembler to interpret write cycles correctly, LatchedDiag(1) needs to be AND'ed with $\overline{Wr}$ as shown in Figure 5, so that it is always low during write cycles.

3. Use the Reset Mode Vector to set the R3051 to use single word data refills instead of 4-word data block refills. This will allow all 4 words on a data load d-cache misses to be seen.

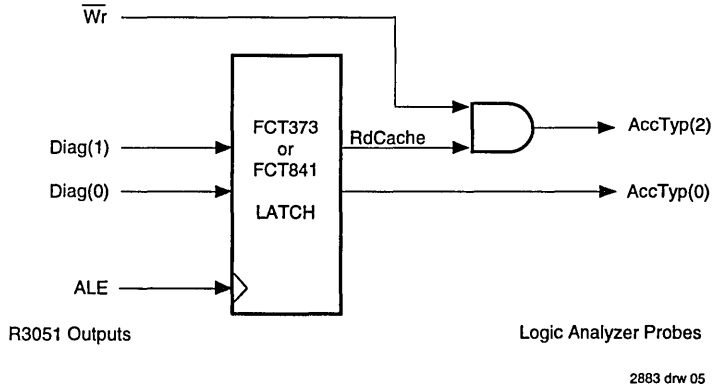


Figure 5. Using Diag(1:0) with the Disassembler

State/Timing E | Listing 1 | Invasm | Print | Run | Markers Off

Label>	DATA	ADDR	CLKN	BAWRRA	ALE	WRNRDN
Base>	Hex	Hex	Hex	Binary	Binary	Binary
274	8C490000	4	1	111110	0	11
275	8C490000	0	0	111110	0	11
276	00000000	4	1	110111	1	01
277	00000000	4	0	110110	0	01
278	00000000	4	1	110110	0	01
279	00000000	4	0	110110	0	01
280	00000000	4	1	110110	0	01
281	00000000	4	0	110110	0	01
282	00000000	4	1	110110	0	01
283	00000000	4	0	110110	0	01
284	00000000	4	1	110110	0	01
285	00000000	4	0	100110	0	01
286	00000000	4	1	100110	0	01
287	00000000	4	0	111110	0	11
288	1FC00240	4	1	111101	1	10
289	1FC00240	4	0	111100	0	10

2883 drw 06

Figure 6. R3051 State Trace Listing using Clk2xIn

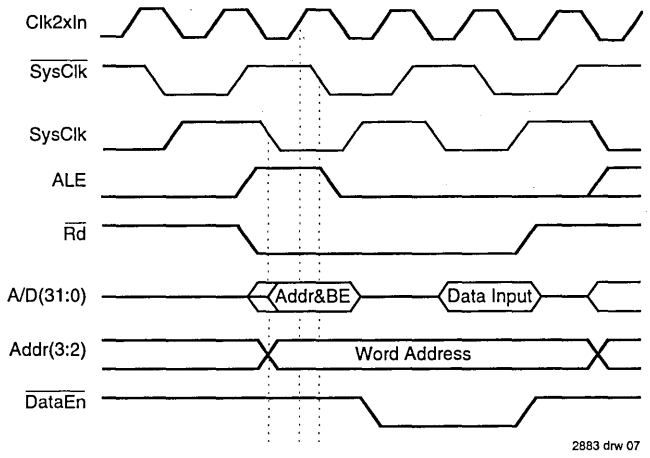


Figure 7. Choosing a Clock Edge

Using State Trace Listings and Timing Waveforms

The IDT7RS364 Disassembler is a good tool for easing the use of a Logic Analyzer when debugging a target system. However, sometimes, even lower level detail is needed to examine clock by clock behavior of particular bus cycles. The HP16500 performs this function in its State Analyzer mode by sampling with the CPU's system clock as shown in Figure 6. Because the state analyzer mode has a maximum speed of 35 MHz, certain restrictions apply. Ideally because the R3051 uses both edges of its SysClk output to generate control lines, it is preferable to use Clk2xIn or to clock on both edges of either SysClk or its buffered/inverted version SysClk. On the HP16500, high speed clocks should always use their ground shield on the probe to reference the input properly so that the probe does not sense signal overdrive. The edge of the reference clock should be chosen carefully so that it ideally clocks just before ALE de-asserts as shown in Figure 7. This allows the address to be seen along with the data on the multiplexed A/D bus so that dedicated address lines probes are not required. When choosing a clock, keep in mind that the HP16500 has 10 nsec setup time and 1 nsec hold time relative to the clock. In addition, the HP16500's Time Tagging feature if used is limited to 16.67 MHz.

Systems running with a Clk2xIn over 35 MHz (17.5 MHz CPU) can either clock the State Analyzer mode less frequently or use the Timing Analyzer mode. When clocking less frequently, care must be taken to chose a clock edge that adequately strobes ALE during its high period so that the address can be determined. Because the R3051 only has a 1/2 clock intercycle memory latency, Rd and Wr and other control lines may not be seen to de-assert between memory cycles when clocked at the SysClk frequency.

The HP16500 Logic Analyzer's Timing mode displays signals in waveform format as shown in Figure 8 and is capable of internally generating a 100 MHz (10 nsec) sample clock. To maintain all the functional timing relationships relative to the Clk2xIn, the timing mode allows asynchronous sampling up to 50 MHz CPU speed. The disadvantage of using the Timing mode is that the value of busses is hard to decipher when shown in waveform format. If necessary, HP16500 can be set up in its mixed mode display to display both state and timing modes on the same screen.

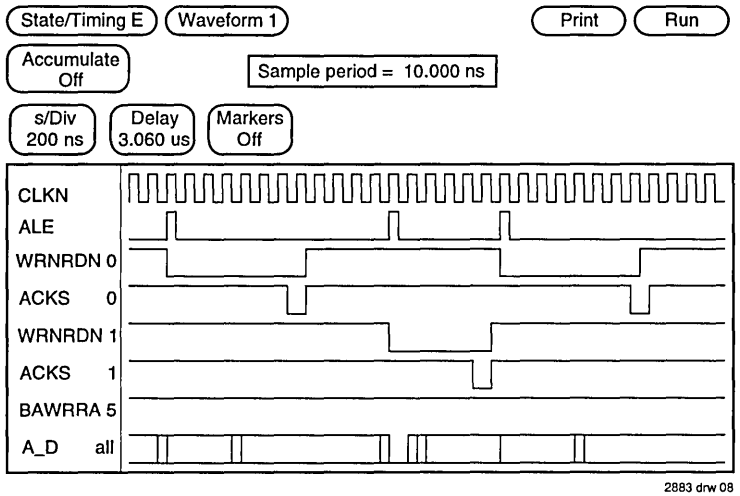


Figure 8. R3051 Timing Mode Waveform

SUMMARY

The use of the HP16500 and the IDT7RS364 Disassembler is but one example of the availability and compatibility of R3000 tools and software that can be used on the R3051. The Disassembler formats logic analyzer state traces into assembly level mnemonics to allow easier user interpretation. Similarly, other R3000 software, compilers, as well as other development tools such as the IDT7RS901 IDT/sim ROMable Kernel/Boot Monitor can also be used on R3051 systems with little or no modification.

Integrated Device Technology, Inc.

USING THE IDT79R3051™ AND THE IDT79R3081™ WITH THE HP16500 LOGIC ANALYZER

APPLICATION NOTE AN-111

Supplement to Application Note AN-93

By Gary Szilagyi

INTRODUCTION

In Application Note-93, the use of IDT's 7RS364 disassembler with the HP16500 Logic Analyzer for the IDT79R3051™ RISController™ family of CPUs was discussed in detail. However, the original versions of the disassembler were form-fitted for the R3000 CPU interface of a 32-bit non-multiplexed bus design. In order to accommodate the high level of integration on-board the R3051, including the 4kB–8kB of instruction cache, 2kB of data cache, 4-deep read and write buffers and the R3000A execution engine—all in a single 84-pin package, the 32-bit bus required multiplexing address and data pins. Although the original versions of the disassembler remain compatible with the new family of IDT's RISControllers, an effort was made to simplify the interface between R3051 and the disassembler to accommodate simple triggering schemes, as well as future IDT embedded controllers that continue in the path of the R3051 family.

THE IDT7RS364 DISASSEMBLER AND THE IDTR3051

The IDT7RS364 Disassembler consists of a software package that greatly eases the task of debugging software on the IDTR3051 family of CPUs. The HP16500 allows the capture of executed hex/binary machine opcodes in a typical Logic Analyzer State Trace Listing format with the ability to decode and display the acquisitions in the R3000 assembly code mnemonic format, as seen in Figure 1. Thus, the engineer does not have to resort to look-up tables, and can effectively determine the exact processor state for easy software debugging.

The original versions of the disassembler were form-fitted to the R3000 CPU interface. Although the derivative products of the IDT R3051 family are compatible, the $\overline{RD}$ and $\overline{WR}$ signals used for data acquisitions by the disassembler package causes some confusion during a high-speed burst read. As discussed in Application Note AN-93, the work-around was to create a more complex read strobe in order to capture a four-word burst read by setting up a trigger mechanism on the HP16500 that looks like: [($\overline{SysClk}$ ==↑) AND [($\overline{ACK}$ == 0) OR $\overline{RDCEN}$ == 0)]. However, this is only applicable to systems that bring the $\overline{ACK}$ signal LOW at precisely the same time the $\overline{RDCEN}$ is LOW, or that don't bring it LOW at all during a four word burst read. If, for instance, the $\overline{ACK}$ signal triggered in the phase between two successive $\overline{RDCEN}$s, a duplicated capture would occur. The disassembler was modified a second time to remedy this situation. In a read cycle, the $\overline{RD}$ pin will be asserted LOW for the entire cycle and the $\overline{RDCEN}$ signal toggles to successfully pass each of the four words across the bus. The newest version of the disassembler

State/Timing E | Listing 1 | Invasm | Print | Run

Markers Off

Label> / Base>	ADDR Hex	R3000 Mnemonic hex	STAT Hex	Time Absolute
-6	1FC00000	J 0x1FC00220	0010	0 s
-5	1FC00004	NOP	0010	760 ns
-4	1FC00220	LUI v0,0x0010	0010	1.52 us
-3	1FC00224	MTC0 v0,$12	0010	2.24 us
-2	1FC00228	MTC0 zero,$13	0010	3.00 us
-1	1FC0022C	LUI v0,0xA000	0010	3.76 us
0	1FC00230	LUI t0,0xAAAA	0010	4.52 us
1	1FC00234	ORI t0,t0,0x5555	0010	5.24 us
2	1FC00238	SW t0,0x0000(v0)	0010	6.00 us
3	1FC0023C	SW zero,0x0004(v0)	0010	6.76 us
4	00000000	STORE DATA 0xAAAA5555	0000	7.40 us
5	1FC00240	LW t1,0x0000(v0)	0010	7.88 us
6	00000004	STORE DATA 0x00000000	0000	8.52 us
7	1FC00244	NOP	0010	9.00 us
8	00000000	LOAD DATA 0xAAAA5555	0010	9.64 us
9	1FC00248	B 0x1FC00258	0010	10.32 us

Figure 1. R3051 Address/Data Trace List on a Logic Analyzer

begins "LOAD" captures not on $\overline{RD}$, but rather upon the $\overline{RDCEN}$. For interleaved memory systems that do not toggle the $\overline{RDCEN}$ pin, please refer to section "Hazards" for more details. During a write cycle, it triggers upon the rising edge (from LOW-to-HIGH) of the $\overline{WR}$ signal. Thus, the newest revision of the disassembler now expects the $\overline{RDCEN}$ and the $\overline{WR}$ signals as clocks to strobe the address and data into the HP16500, as well as the $\overline{WR}$, DIAG_1 and DIAG_0 to verify and decode the processor status

INTERFACING THE HP16500 TO THE '385 EVALUATION BOARD

In order to insure proper operation of the disassembler, the correct interface between the R305x target system and the HP16500 must be available. The disassembler requires a particular pinout setup on the logic analyzer's five 16-channel probe pod sets. The interface protocol must be followed for correct interpretation of the address, data, and status lines by the pre-processor. Table 1 displays the default pod connections that the HP16500 expects (same setup for the 7RS385 evaluation board). This information is stored on disk in the configuration file "DIS_305x_E". When loaded, this file not only loads the disassembler, but also all the state and timing information, including the default pod connections expected at the system interface.

Application Note-93 discusses in detail the interface between typical R305x based systems and the logic analyzer. Rather than repeat that discussion, the interface between the 7RS385 Evaluation board and the disassembler requires some elaboration. For instance, the '385 Hardware User's Manual shows the connections to be made from the board's five 20-pin logic analyzer sockets and the logic analyzer's five, 16-channel pods. Note however that in section 2–5 of the '385 Hardware User's Manual, the connections on the status pod (pod#5) are incorrect. In order to be consistent with the protocol of the disassembler, some of the pins need to be connected as follows:

- WR (J12 pin #17) needs to be on pod #5 channel #4
- RDCEN (J12 pin #14) needs to be on pod #5 channel #5

The disassembler also requires status lines for determining processor status: $\overline{WR}$, $\overline{RDCEN}$, DIAG_1, and DIAG_0. The $\overline{WR}$ signal distinguishes between read and write cycles. The $\overline{RDCEN}$ pin is used to identify a false trigger for applications that assert the $\overline{RDCEN}$ signal during writes. In order to avoid a duplicate capture, the $\overline{RDCEN}$ signal is polled to determine if it was the cause of the acquisition. If it was, then a trigger-

Table 1. R3051 Default Pod Connections on the HP16500 Logic Analyzer

POD chan	5 sig	POD chan	4 sig	POD chan	3 sig	POD chan	2 sig	POD chan	1 sig
15	X	15	A/D(31)	15	A/D(15)	15	A(31)	15	A(15)
14	X	14	A/D(30)	14	A/D(14)	14	A(30)	14	A(14)
13	X	13	A/D(29)	13	A/D(13)	13	A(29)	13	A(13)
12	Diag_1[2]	12	A/D(28)	12	A/D(12)	12	A(28)	12	A(12)
11	X	11	A/D(27)	11	A/D(11)	11	A(27)	11	A(11)
10	Diag_0	10	A/D(26)	10	A/D(10)	10	A(26)	10	A(10)
9	X	9	A/D(25)	9	A/D(9)	9	A(25)	9	A(9)
8	X	8	A/D(24)	8	A/D(8)	8	A(24)	8	A(8)
7	X	7	A/D(23)	7	A/D(7)	7	A(23)	7	A(7)
6	X	6	A/D(22)	6	A/D(6)	6	A(22)	6	A(6)
5	$\overline{RDCEN}$	5	A/D(21)	5	A/D(5)	5	A(21)	5	A(5)
4	$\overline{WR}$	4	A/D(20)	4	A/D(4)	4	A(20)	4	A(4)
3	X	3	A/D(19)	3	A/D(3)	3	A(19)	3	Addr(3)
2	X	2	A/D(18)	2	A/D(2)	2	A(18)	2	Addr(2)
1	X	1	A/D(17)	1	A/D(1)	1	A(17)	1	$\overline{BEN(1)}$
0	X	0	A/D(16)	0	A/D(0)	0	A(16)	0	$\overline{BEN(2)}$
NClk	$\overline{WR}$	MClk	$\overline{RDCEN}$	LClk		KClk		JClk	

NOTES:

1. Master Clock Format: N↑ + M↑ (default for the 7RS385 Evaluation Board setup)
2. POD5(12) is Diag_1 and POD5(10) is Diag_0 (Diag pins are not latched on the 7RS385 Eval Board). If running uncached, then Diag_1 MUST be grounded (GND), and Diag_0 is not used by disassembler.
3. A(31:4) are connected to the Address Latch outputs. The rest of the signals are connected to R3051 outputs. X's denote unused probes that can be assigned by the user.

error message, "T.E", and the store instruction along with the write data on the bus is displayed (e.g. "T.E. (STORE 0xxxxxxxxx)). The diagnostic pin DIAG_1 distinguishes if the external memory read was cacheable, and if so, determines with DIAG_0 if it was an instruction or data read. Note that for the newest IDT embedded controller, the R3081, DIAG_1 is defined during writes, yielding cache information for "STORE" instructions. A second version of the disassembler, "DIS_3081", exploits this feature for external cache support. By defining the DIAG_1 pin during writes, the CPU will signal whether the data being written was retained in the on-chip data cache. Keep in mind that the DIAG_0 pin remains undefined during write cycles. This information is extremely helpful to the programmer to determine the processor's state when tracing through the software.

The diagnostic pins on the '385 board are **NOT LATCHED**, and therefore are time-multiplexed pins. Thus, the user must either latch these pins with an external latch as seen in Figure 2 or proper decoding of cached code, or connect both diagnostic pins to GND. Although the disassembler is capable of interpreting the bus transactions of cached code, keep in mind that all logic analyzers and disassemblers can only capture external CPU memory accesses. The R3051 has large internal caches, and is capable of running much of its code from within. In order for the disassembler to accurately reflect the entire instruction/data flow, the R3051 must be ran uncached. For more information regarding running cached code and data, please refer to Application Note AN-93 for a complete discussion.

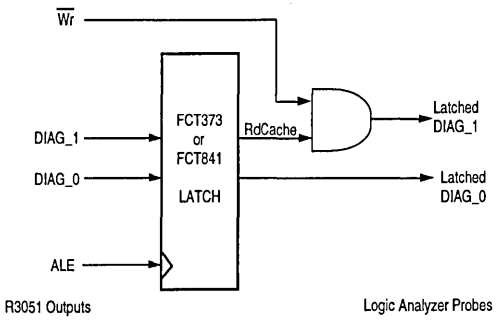


Figure 2. R3051 Address/Data Trace List on a Logic Analyzer

LOADING AND RUNNING THE DISASSEMBLER

Included in the software package are two files. The first is the disassembler application "DIS_305x". The second is the setup file, "DIS_305x_E", containing all the state and timing information required by the disassembler, as well as the assigned pod connections expected by the HP16500 for the R305x target system.

After the HP operating system boots up completely, the system configuration screen as shown in Figure 3 should be displayed. To load the disassembler into the HP16500, the following steps must be taken:

1. Insert the disassembler diskette into the front disk drive.
2. Select the "Configuration" field as shown in Figure 3. A pop-up menu with options will appear. Choose the "Front Disk" under the pop-up menu.
3. A new screen will appear that looks like Figure 4. Select the "Load" and "State/Timing" fields, and load in the configuration file "Dis_305x_E" by selecting "Execute" as shown in Figure 4.

The HP16500 will then load the disassembler, as well as all the state and timing information and the expected pin-configuration as shown in Table 1 previously. Once the disassembler application and setup files are loaded into the HP, the logic analyzer is ready to set trace conditions for data acquisition.

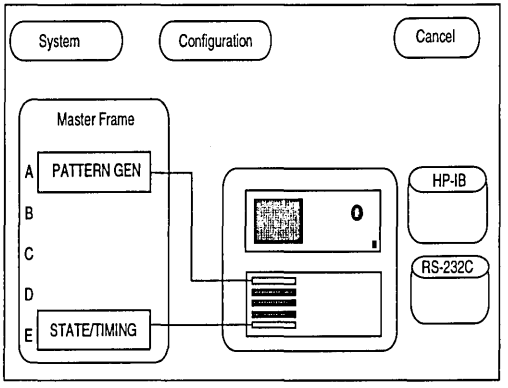


Figure 3. HP16500 Screen Display

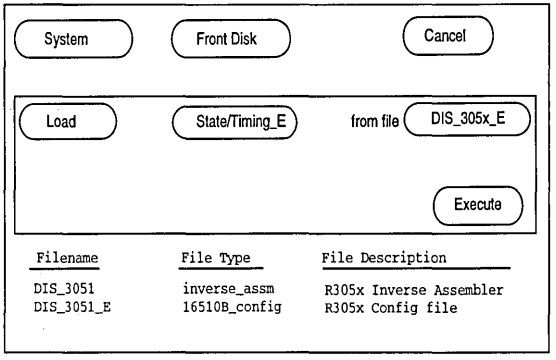


Figure 4. HP16500 Load Screen Display

With the application files loaded, the disassembler is almost ready to be triggered by the target system. Follow the steps below that describe how to run and trigger the disassembler package:

1. Select the "System" field as shown in Figure 4. A pop-up menu will appear with the option of "State/Timing". Choose this field to enter the state and timing mode of acquisition.
2. A new window will appear that is shown in Figure 5. Under the "Configuration" menu lies options that allow the user to set display or change the current configuration of the interface, clocks, and pod connections.
3. Trigger the HP16500.

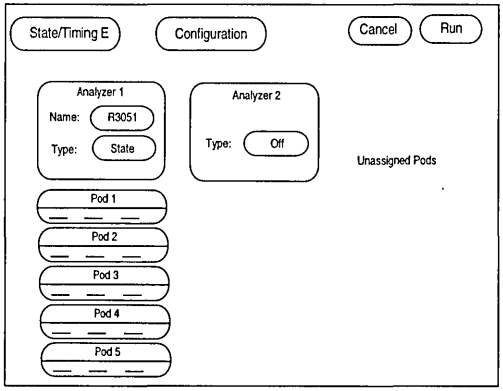


Figure 5. HP16500 State/Timing Mode Display

Once triggered, the logic analyzer will begin its acquisition, and go directly to the "Listing" field. The addresses and disassembled data will be displayed. Note however that the displayed disassembly may be incorrect. This is due to an "unsynchronized" system. The captured data needs to be synchronized with the logic analyzer's display to insure correct disassembly of the bus. The problem of unsynchronized captures arises due to the incomplete status of the processor state for data loads. As a result, when an instruction fetch is scrolled to the top of the screen, and a load data is displayed, but the corresponding load instruction was "cut off" or scrolled off the screen, the disassembler software looses it reference point by which it identifies the load data. As a result, the load data may be decoded incorrectly as an instruction as seen in Figure 6. Notice in this Figure the instruction on line -2. It was disassembled as an instruction instead of as a data load. Also notice the address of the instruction in the sequence of the four word fetch to main memory. This is an unsynchronized display because the corresponding load instruction was scrolled off the top of the display, and due to the way the disassembler interprets and tags the load datas, the reference point was lost. As a result, the load data was interpreted and decoded as an instruction. As shown in Figure 7, the correctly synchronized system has the load instruction displayed at the top of the screen (identified by its address), and the load data is interpreted correctly.

State/Timing E | Listing 1 | Invasm | Print | Run

Markers Off

Label> / Base>	ADDR Hex	R3000 Mnemonic hex	STAT Hex	Time Absolute
-3	1FC00224	NOP	0010	2.24 us
-2	**1FC00228**	**SRL t4,zero,t8**	**0010**	**3.00 us**
-1	1FC0022C	NOP	0010	3.76 us
0	1FC00230	J 0X1FC084F0	0010	4.52 us
1	1FC00234	NOP	0010	5.24 us
2	1FC00238	LW v0,0x0000(s0)	0010	6.00 us
3	1FC0023C	NOP	0010	6.76 us
4	00000000	STORE DATA 0xAAAA5555	0000	7.40 us
5	1FC00240	LW t1,0x0000(v0)	0010	7.88 us
6	00000004	STORE DATA 0x00000000	0000	8.52 us
7	1FC00244	NOP	0010	9.00 us
8	00000000	LOAD DATA 0xAAAA5555	0010	9.64 us
9	1FC00248	B 0x1FC00258	0010	10.32 us

Figure 6. Incorrectly Synchronized Capture (Note line -2)

State/Timing E | Listing 1 | Invasm | Print | Run

Markers Off

Label>	ADDR	R3000 Mnemonic	STAT	Time
Base>	Hex	hex	Hex	Absolute
-4	1FC00220	LW v0,0x0008(s0)	0010	2.24 us
-3	1FC00224	NOP	0010	2.24 us
-2	**1FC00228**	**LOAD DATA 0x12620003**	**0010**	**3.00 us**
-1	1FC0022C	NOP	0010	3.76 us
0	1FC00230	J 0X1FC084F0	0010	4.52 us
1	1FC00234	NOP	0010	5.24 us
2	1FC00238	LW v0,0x0000(s0)	0010	6.00 us
3	1FC0023C	NOP	0010	6.76 us
4	00000000	STORE DATA 0xAAAA5555	0000	7.40 us
5	1FC00240	LW t1,0x0000(v0)	0010	7.88 us
6	00000004	STORE DATA 0x00000000	0000	8.52 us
7	1FC00244	NOP	0010	9.00 us
8	00000000	LOAD DATA 0xAAAA5555	0010	9.64 us
9	1FC00248	B 0x1FC00258	0010	10.32 us

Figure 7. Correctly Synchronized Capture (Note line -2)

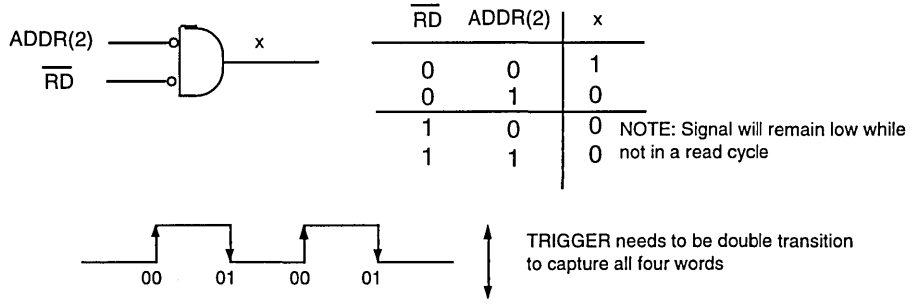


Figure 8. Simulated $\overline{\text{RDCEN}}$ signal

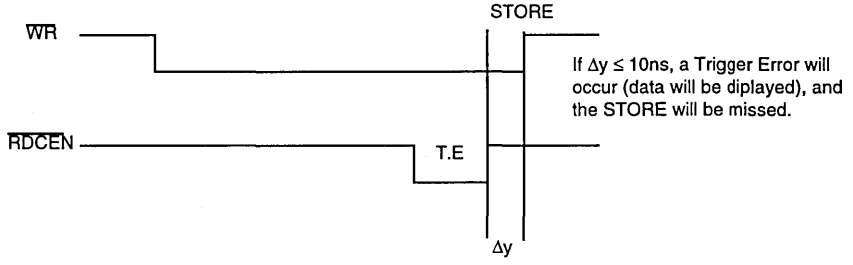


Figure 9. $\overline{\text{RDCEN}}$ Asserted during STORE

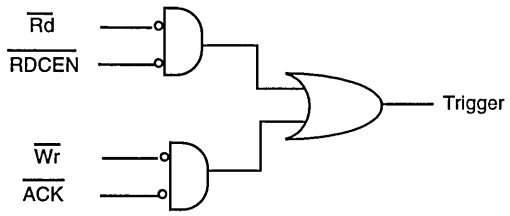


Figure 10. Simple Trigger Logic

To synchronize the system and to insure valid results, the following steps must be taken:

1. Identify the first instruction fetch by its address, not its displayed mnemonic, of the captured data and scroll this line to the top of the screen display.
2. At the top of the HP16500 screen is the field "Ivasm". Select this, and the currently displayed capture will be synchronized.
3. Always make sure that each new capture, or a jump ahead in the analyzer's buffer memory is re-synchronized properly or erroneous data might be displayed. The same applies for any move backwards for any displayed capture.

HAZARDS

For interleaved memory systems that do not toggle the $\overline{\text{RDCEN}}$ four times, but rather keep it asserted, the only data to be captured during quad-word reads will be the last word of the transfer. In order to fix this, the user might wish to simulate a $\overline{\text{RDCEN}}$ strobe during the quad-word read by utilizing the lower order address pins Addr(3:2). This can be accomplished by gating the Addr(2) pin of this 2-bit bus with the $\overline{\text{RD}}$ signal from the CPU. Whenever the next word in the sequence comes across the bus during a read cycle, the transition from LOW-to-HIGH, or HIGH-to-LOW will begin an acquisition, and thus simulate the strobbing of $\overline{\text{RDCEN}}$. Note however, the trigger transition on the HP must be set to both rising and falling transitions as seen in Figure 8.

Another hazard to be cautious about is if the $\overline{\text{RDCEN}}$ comes at precisely, or within a 10ns window (Δy) of the rising edge of the $\overline{\text{WR}}$ signal. If so, then this would be regarded as an invalid write with a trigger error (T.E) ocurring and the data on the bus at the time of the invalid capture will be displayed. In this case, the capture on the rising edge of write will be missed and the data displayed with the T.E. is the valid capture as shown in Figure 9. During any case that a $\overline{\text{RDCEN}}$ comes in on a write cycle, a T.E. will occur.

Finally, a feature in HW that would be extremely useful for triggering is a specified trigger signal for the HP logic analyzer that would distinguish between the status of reads and writes triggered by $\overline{\text{ACK}}$. The trigger would simply be established by gating the read and write signals and ORing the results as shown in Figure 10. This should eliminate any trigger edge problems associated with simple data acquisitions for inverse assembly.

SUMMARY

The use of the HP16500 and the IDT7RS364 Disassembler helps to ease the task of software development and debugging on the R305x and the R3081. The disassembler formats logic analyzer state traces into assembly level mnemonics to allow easier user interpretation. It is one of the many useful development tools already available for IDT's MIPS R3000 compatatible CPUs. Similarly, other R3000 software, compilers, as well as other development tools such as the IDT7RS901 IDT/sim ROMable Kernel/Boot Monitor can also be used on R3051 and R3081 systems with little or no modification.

Integrated Device Technology, Inc.

IDT/C™ BINARY UTILITIES

APPLICATION NOTE AN-125

By Evelyn Zhan

INTRODUCTION

IDT/C™ is a development package which contains a cross compiler, optimizing scheduler, cross assembler, linker, and downloader. It is intended for cross-development with an IDT RISController™ as the target architecture. The 'C' compiler is compliant with ANSI 'C' standard and performs the optimizations available in state of the art 'C' compilers. In version 5.0 the assembler is compatible with files written for the MIPS. The assembler supports the R30xx machine instructions and architecture described in the book by Gerry Kane, "MIPS RISC architecture." The cross compiler package runs on a variety of host machines and operating systems and is part of IDT's Cross Development System tools which includes other packages such as IDT/SIM™, a debug monitor and diagnostic tool; and IDT/KIT™, a set of run time support libraries in source form to enable quick implementation of embedded applications. This application note describes the binary utilities of the IDT/C toolchain.

Archive (gar):

The **gar** program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files. Archive files are libraries of files which are typically used for the link process. Files are created by a compiler into a format known as the object format and can then be stored as members in an archive file. These members are then used by the link editor to generate a final executable code. The **gar** command is a powerful command that creates and manipulates archive libraries. These libraries can help user organize development effort and control the generation of executables.

usage: *gar [-] switches[mod [relpos]] archive [member ...]*

switches must be one of the following operations:

- **d** Delete modules from archive. Specify the names of modules to be deleted as *member...* to delete.
- **m** move member in an archive. You can use the 'a', 'b', or 'i' modifiers to move them to specified place. If no modifiers are used with m, the *member...* will be moved to the end of the archive.
- **p** Print the specified members of the archive to the standard output file. If there are no member arguments, all the files in the archive are printed.
- **q** Quick append. Add files *member...* to the end of archive, without checking for replacement.
- **r** Insert files *member...* into archive with replacement. By default, new members are added at the end of the file; you can use modifiers 'a', 'b', 'i' to request place ment relative to some existing member.
- **t** Display a table listing the contents of archive, or those of the files listed in *member...* that are present in the archive.
- **x** Extract members from the archive. If you do not specify a member, all files in the archive are extracted.

A number of modifiers (*mod*) may immediately follow the switches keyletter, to specify variations on an operation's behavior:

- **a** Add new files after an existing member(*relpos*) of the archive.
- **b** Add new files before an existing member (*relpos*) of the archive.
- **c** Create the archive.
- **i** Add new files before an existing member (*relpos*)of the archive. Same as 'b'.
- **o** Preserve the original dates of the members when extracting them. Otherwise it is stamped with the time of extraction.
- **s** Write an object file index into the archive, or update an existing one.
- **u** Insert only those of the files you list that are newer than existing members of the same names. This modifier is allowed only for the operation 'r'. i.e. 'gar -ru'...

usage: *gar -M [< script file]*

If you use the single command-line option '-M' with **gar**, you can control its operation with a rudimentary command language. This form of **gar** operates interactively if standard input is coming directly from a terminal.

Here are the commands you can use in **gar** scripts, or when using **gar** interactively.

ADDLIB *archive (module, module, ...module)*
Add all the contents of archive (or, if specified, each named module from archive) to the current archive. Requires prior use of OPEN or CREATE.

ADDMOD *member, member, ...member*
Add each named member as a module in the current archive. Requires prior use of OPEN or CREATE.

CLEAR
Discard the contents of the current archive, canceling the effect of any operations since the last SAVE.

CREATE *archive*
Creates an archive, and makes it the current archive. The new archive is not actually saved as archive until you use SAVE.

DELETE *module, module, ...module*
Delete each listed module from the current archive. Requires prior use of OPEN or CREATE.
END
Exit from **gar**. This command does not save the output file.
EXTRACT *module, module, ...module*
Extract each module from the current archive. Requires prior use of OPEN or CREATE.
LIST
Display full contents of the current archive. Requires prior use of OPEN or CREATE.
OPEN *archive*
Opens an existing archive for use as the current archive.
REPLACE *module, module, ...module*
In the current archive, replace each existing module from files in the current working directory.
SAVE
Commit your changes to the current archive, and actually save it as a file with the name specified in the last CREATE or OPEN command.
Example of usage of gar:

```
add.c:

int Add(int a, int b)
{
int  c;

c = a + b;
return  c;
}
sub.c:

int Sub(int a, int b)
{
int  c;

c = a - b;
return  c;
}
mult.c:

int Mult (int a, int b)
{
int  c;

c = a * b;
return  c;
}
div.c:

int Div(int a, int b)
{
int  c;

c = a / b;
return  c;
}

Sample C code:

#define  SIZE 50
int a[SIZE][SIZE], b[SIZE][SIZE],
  c[SIZE][SIZE];

main()
{
int  i, j, k;

for (i = 0; i < SIZE; i++)
     for (j = 0; j < SIZE; j++)
         a[i][j] = b[i][j] = 7;


printf("Beginning Matrix Multiplication.
  \n");
for (i = 0; i < SIZE; i++)
     for (j = 0; j < SIZE; j++)
          {
           c[i][j] = 0;
           for (k = 0; k < SIZE; k++)
                c[i][j] = Add(c[i][j],
 Mult(a[j][k], b[k][j]));
          }
 printf("DONE Matrix Multiplication. \n");
}

Makefile:

LIBRARY = libmylib
EXEC = main
LIBOBJS = add.o sub.o mult.o div.o
SREC = $(EXEC).srec

all: $(LIBRARY).a $(EXEC) $(SREC)
$(SREC): $(EXEC)
     objcopy -O srec $(EXEC) $(SREC)
$(LIBRARY).a: $(LIBOBJS)
    gar -rc $(LIBRARY).a $(LIBOBJS)
    gnm $(LIBRARY).a > $(LIBRARY).nm

$(EXEC): main.o idt_csu.o $(LIBRARY).a
     gcc -nostdinc -nostdlib -g -msoft-float
  -I/IDTC -L/IDTC -Ttext 80020000 -o main
  idt_csu.o main.o \
     -lmylib -lkil -lc -lm -llnk -lgcc
     gsize -x $(EXEC) > $(EXEC).size
     objdump -d $(EXEC)  >  $(EXEC).dis

.c.o:
   gcc -nostdinc -g -msoft-float -c -I/IDTC
  $*.c
.S.o:
   gcc -nostdinc -g -msoft-float -
  xassembler-with-cpp -c -I/IDTC $*.S
```

```
  gar -rc libmylib.a add.o sub.o mult.o
  div.o
This creates a library called libmylib.a
  containing the files add.o sub.o mult.o
  and div.o

  gar -t libmylib.a
Comes back with:
add.o
sub.o
mult.o
div.o

  gar -x libmylib.a
This extracts the files add.o, sub.o, mult.o
  and div.o from libmylib.a

  gar -d libmylib.a sub.o
Deletes sub.o from the archive

  gar -t libmylib.a
Displays add.o, mult.o and div.o

  gar -r libmylib.a sub.o
Add sub.o back into the archive
```

Name (gnm):

The **gnm** utility generates symbol table for the object file. The file can be a simple object file, an executable file, or an archive file. Each symbol is preceded by a value which defines the characteristics of the symbol itself. The **gnm** command is well used by users to provide information on the structure and content of object and executable files.

usage: *gnm [-n] objectfile > outfile.nm*

List the symbol table for *objectfile*, sorted by symbol name (-n option is sorted by symbol address), into file *outfile.nm*.

For each symbol, **gnm** shows:

- The symbol value.
- The symbol type. If lowercase, the symbol is local; if uppercase, the symbol is global.
  - A Absolute.
  - B BSS (uninitialized data).
  - C Common.
  - D Initialized data.
  - I Indirect reference.
  - T Text (program code).
  - U Undefined.
- The symbol name.

Example of usage of **gnm**:

```
  gnm  libmylib.a  >  libmylib.nm
libmylib.nm contains:
add.o:
00000000 T Add
00000000 t Add
00000000 t __gnu_compiled_c
00000000 t gcc2_compiled.

sub.o:
00000000 T Sub
00000000 t Sub
00000000 t __gnu_compiled_c
00000000 t gcc2_compiled.

mult.o:
00000000 T Mult
00000000 t Mult
00000000 t __gnu_compiled_c
00000000 t gcc2_compiled.

div.o:
00000000 T Div
00000000 t Div
00000000 t __gnu_compiled_c
00000000 t gcc2_compiled.


  gnm  -n libmylib.a  >  libmylib.nm1
libmylib.nm1 contains:
add.o:
00000000 T Add
00000000 t Add
00000000 t gcc2_compiled.
00000000 t __gnu_compiled_c

sub.o:
00000000 T Sub
00000000 t Sub
00000000 t gcc2_compiled.
00000000 t __gnu_compiled_c

mult.o:
00000000 T Mult
00000000 t Mult
00000000 t gcc2_compiled.
00000000 t __gnu_compiled_c

div.o:
00000000 T Div
00000000 t Div
00000000 t gcc2_compiled.
00000000 t __gnu_compiled_c
```

The addresses above are zeros since the text is relocatable.

Object Copy (objcopy):

This utility is used to convert ecoff files to S-record, so that code can be downloaded to the IDT evaluation board or to a PROM burner.

usage: objcopy -O srec [-b *num*]
[-i *bytenum*] [-p]*objectfile outfile.srec*

objcopy -O srec objectfile outfile.srec
This converts the ecoff objectfile to Motorola S3 record format, for downloading to the evaluation boards.
objcopy -O srec -b num *objectfile outfile.srec*
-b *num* is useful when programming EPROMS for boards which require bytewide EPROMS.
-b 0 creates S-record files corresponding to 0th byte slice of 4-byte word.
-b 1 creates S-record files corresponding to 1st byte slice of 4-byte word.
-b 2 creates S-record files corresponding to 2nd byte slice of 4-byte word.
-b 3 creates S-record files corresponding to 3rd byte slice of 4-byte word.
objcopy -O srec -b *num* -i *bytenum objectfile outfile.srec*
-i option must be used in conjunction with -b option. It is useful for programming EPROMS for boards that have interleaved addressing.
-i 1 interleave one byte.
-i 2 interleave two bytes. etc.
objcopy -O srec -p -b *num objectfile outfile.srec*
-p option is used to create prommable S-records. It should be used with -b to create bytewide PROMS. It orders the sequence of sections to be .text, .data, .bss and starts the address fields from address 0x00000000.

Object Dump (objdump):

Displays information about ecoff files. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just wnat their program to compile and work.

usage: objdump [-h] [-d] [-t] objectfile > *outfile*

objdump -h objectfile > *outfile*
Display summary information from the section headers of the *objectfile*.
objdump -d *objectfile* > *outfile*
Display the assembler mnemonics for the machine instructions from *objectfile*. This is very useful when doing machine level debugging. User can set a break point at a certain virtual address for a corresponding assembly instruction.
objdump -t *objectfile* > *outfile*
Print the symbol table entries of the file.
Example of usage of **objdump**:

```
objdump -h main > main.od

main.od contains:
main:     file format ecoff-bigmips

SECTION 0 [.scommon]     : size 00000000 vma
  00000000 align 2**4
SECTION 1 [.reginfo]     : size 0000001c vma
  00000000 align 2**4
SECTION 2 [.text]        : size 00005f40 vma
  80020000 align 2**4
 ALLOC, LOAD, CODE
SECTION 3 [.rdata]       : size 000004a0 vma
  80025f40 align 2**4
 ALLOC, LOAD, READONLY, DATA
SECTION 4 [.data]        : size 00000ca0 vma
  800263e0 align 2**4
 ALLOC, LOAD, DATA
SECTION 5 [.lit8]        : size 00000000 vma
  80027080 align 2**4
 ALLOC, LOAD, READONLY, DATA
SECTION 6 [.lit4]        : size 00000000 vma
  80027080 align 2**4
 ALLOC, LOAD, READONLY, DATA
SECTION 7 [.sdata]       : size 00000080 vma
  80027080 align 2**4
 ALLOC, LOAD, DATA
SECTION 8 [.sbss]        : size 00000080 vma
  80027100 align 2**4
 ALLOC
SECTION 9 [.bss]         : size 000078d0 vma
  80027180 align 2**4
 ALLOC

objdump -t main  >  main.sym

main.sym contains:

main:     file format ecoff-bigmips

SYMBOL TABLE:
[  0] e 80020000 st 6 sc 1 indx 1     start
      Local symbol: 195
[  1] e 80027100 st 1 sc 5 indx fffff
 _fbss
[  2] e 8002ea50 st 1 sc 5 indx fffff
 end
[  3] e 8002f080 st 1 sc 5 indx fffff
 _gp
[  4] e 80020880 st 6 sc 1 indx 5
 init_exc_vecs
      Local symbol: 261
[  5] e 800206a4 st 6 sc 1 indx d
 config_memory
      Local symbol: 244
[  6] e 80020bb8 st 6 sc 1 indx 3
 config_Icache
      Local symbol: 280
[  7] e 80020b70 st 6 sc 1 indx 1
 config_Dcache
      Local symbol: 278
[  8] e 80020d58 st 6 sc 1 indx 9
 flush_Icache
      Local symbol: 286
[  9] e 80020ce8 st 6 sc 1 indx 7
 flush_Dcache
```

```
      Local symbol: 284
[ 10] e 80020660 st 6 sc 1 indx 9
 init_tlb
      Local symbol: 240
.........................
```

```
objdump -d main  > main.dis

main.dis contains:
main:     file format ecoff-bigmips

Disassembly of section .text:
80020000 <start> lui $v0,8208
80020004 <start+4> mtc0 $v0,$12
80020008 <start+8> mtc0 $zero,$13
8002000c <start+c> lui $t3,43690
80020010 <start+10> ori $t3,$t3,21845
80020014 <start+14> mtc1 $t3,$f0
80020018 <start+18> mtc1 $zero,$f1
8002001c <start+1c> mfc1 $t0,$f0
80020020 <start+20> mfc1 $t1,$f1
...
80020028 <start+28> bne $t0,$t3,80020040
 <start+40>
...
80020030 <start+30> bnez $t1,80020040
 <start+40>
...
80020038 <start+38> j 80020048 <start+48>
.............................
80020164 <start+164> lui $v0,49088
80020168 <start+168> jr $v0
...
80020170 <main> addiu $sp,$sp,-40
80020174 <main+4> sw $ra,36($sp)
80020178 <main+8> sw $s8,32($sp)
8002017c <main+c> move $s8,$sp
80020180 <main+10> jal 80025f0c <__main>
...
80020188 <main+18> sw $zero,16($s8)
8002018c <main+1c> lw $v0,16($s8)
...
80020194 <main+24> slti $v1,$v0,50
80020198 <main+28> beqz $v1,80020280
 <main+110>
...
800201a0 <main+30> sw $zero,20($s8)
800201a4 <main+34> lw $v0,20($s8)
...
800201ac <main+3c> slti $v1,$v0,50
800201b0 <main+40> beqz $v1,80020264
 <main+f4>
...
800201b8 <main+48> lw $v0,16($s8)
.............................
```

Index Archive Library (ranlib):

Generates an index to the contents of an archive and stores it in the archive. **Ranlib** converts each archive to a form that can be linked more rapidly. It does this by adding a table of contents called __.SYMDEF to the beginning of the archive. **Ranlib** uses **gar** to reconstruct the archive. Sufficient temporary file space must be available in the file system that contains the current directory.

usage: ranlib archive

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

Size (gsize):

This command is used to get sizes of different sections in the object file. It prints the number of bytes required by the text, data, and bss portions, and their sum in hex and decimal of each object file.

usage: gsize [-d | -o | -x | radix=number]
objectfile... > *outfile*

Lists the section sizes, and the total size for each of the objectfile or archive in its argument list into *outfile*. The size of each section is given in decimal ('-d', or 'radix=10'); octal ('-o', or 'radix=8'); or hexadecimal ('-x', or 'radix=16').

Example of usage of **gsize**:

```
gsize main  > main.size

main.size contains:

text      data     bss      dec      hex
 filename
5f40    11c0    7950    59984    ea50    main
```

SUMMARY

The IDT/C binary utilities include **GAR, GNM, OBJCOPY, OBJDUMP, RANLIB** and **GSIZE**. Together, they are very useful tools for programmers to develop and debug their applications.

Integrated Device Technology, Inc.

THE ELF-64™ TOOL CHAIN

APPLICATION NOTE AN-126

By Ketan Deshpande

INTRODUCTION

This application note describes the 64-bit C development tool chain available from Cygnus. The ELF-64™ tool chain is a 64-bit C-compiler tool chain that can be used to generate code for the R4600™ (Orion™) processor operating in an embedded application environment. It is based on the GNU tool chain available in the public domain. The executable created is an ELF (Executable and Linking Format) file.

TOOL CHAIN COMPONENTS

The ELF-64 tool chain consists of the following parts:

1. C-Compiler
2. Assembler
3. Linker
4. Source level Debugger
5. Librarian / Archiver
6. Binary Utilities

The C-compiler is ANSI C compliant and performs optimizations found in all state-of-the-art C-compilers. The compiler generates an intermediate assembly language file from a C file and calls the assembler to generate an ELF object file. The assembler supports the entire MIPS™ ISA (described in the book by Gerry Kane, "MIPS RISC Architecture"). The words "compiler" & "assembler" are used to refer to the cross-development environment too.

The linker links the object files created by the compiler, assembler and the librarian to create an ELF "executable".

The debugger (gdb) provides remote source level debugging capability over a serial link; this is very useful when developing embedded applications.

The librarian/archiver allows the user to create archives of code sections that are frequently used, for linking with various applications.

The binary utilities are useful in extracting information about the ELF file created, generating a disassembled version of the executable, displaying section size information and converting to different file formats.

COMMAND LINE OPTIONS

The C-compiler and linker support a number of options. This application note mentions only a common subset of these options. For a complete listing and description of all options, the user should refer to the manual.

Compiler options

1. Options controlling the ISA level:

The ELF-64 compiler / assembler supports -mips1, -mips2 and -mips3 switches, to generate code for MIPS ISA I, II or III.

-mips1: Generates code for R30xx processors. This generates instructions that access 32-bit data.

-mips2: Generates code for R4x00 and R60xx processors. This generates instructions that access 32-bit data, and some R4x00 specific instructions.

-mips3: Generates code for R4x00 processors. This generates instructions that access 64-bit data, such as double word accesses.

For best utilization of the 64-bit Orion architecture, the mips3 switch should be used, which is also the default. When using the -mips3 switch, the compiler defaults to using 64-bit general purpose registers and 64-bit floating point registers. Integers and long words are 32-bits long, "long long" words are 64-bits. All addresses generated are 32-bits long. The compiler can be told to use non-default sizes for scalar data types, and to use specific processor pipelines for proper instruction scheduling.

2. Optimization & debugging options:

The most commonly used optimization options are -O and -O2, which perform a number of optimizations. The -O2 option performs all optimizations, except loop unrolling (which can be forced by using -funroll-loops) and omitting the frame pointer (-fomit-frame-pointer). Optimization can be switched off completely using -O0.

The -g option tells the compiler to insert debugging information in the object file. This is necessary when debugging with gdb. A debugging level can be specified (1, 2 or 3), depending on the amount of information the user wants to insert. The default is 2, which is typically sufficient to be able to debug using gdb.

For debugging purposes, using "-g -O0" or just "-g" is recommended. If optimization is also specified during debugging, some statements might get moved around, which could be confusing to the person doing the debugging.

3. Options changing the default data sizes:

The -mlong64 switch forces the compiler to generate code using 64 bit wide long words and addresses (pointers).

The -mint64 switch forces the compiler to generate code using 64 bit wide integers. The -mlong64 switch is assumed in this case.

The -mgp32 switch forces the compiler to generate code assuming the general purpose registers in the Orion are only 32 bits long.

The -mfp32 switch forces the compiler to generate code assuming the floating point registers in the Orion are only 32 bits long.

4. Options for proper scheduling:

Using the "-mcpu=" option tells the compiler to use a specific processor pipeline while scheduling instructions. -mcpu=Orion or -mcpu=r4600 tells the compiler to use the

Orion pipeline, and -mcpu=r4400 tells the compiler to use the R4400 pipeline. The compiler defaults to using -mcpu=Orion.

5. Floating point code generation:

The ELF-64 compiler defaults to generating hardware instructions for performing floating point operations. To force the compiler to use an emulation library, the -msoft-float option is specified, and the appropriate library used, at link time. Since the Orion has a Floating Point Accelerator, a user should never need to use this option, though the capability is available in the tool chain and may be used for future CPU products.

6. Other options:

-nostdinc: This option tells the compiler not to look in the standard include path for the include files. This is useful during embedded applications development, when the user needs to use non-standard libraries, which have their own include files.

-Wa or -Wl: This option allows the user to pass assembler and linker options on the C-compiler command line. e.g. -Wa,-alh instructs the compiler to invoke the assembler to list assembly and high-level source code to the display.

Linker options

1. Options controlling different sections in the executable:

The ELF-64 linker places the different sections in the ELF file at certain default addresses. These addresses can be changed using the -T option. To force the linker to place the .text section at a specific address, the option -Ttext <address> can be used. Similarly, use -Tdata and -Tbss to force the linker to locate the .data and .bss at specific addresses. In a case where all 3 section addresses are specified, it is the user's responsibility to see that the sections do not overlap. The linker uses a default script to place the different sections in the ELF file. Users can specify their own script files, thus finely controlling the appearance of the ELF executable, using the -T<scriptfilename> switch. A discussion of linker scripts is outside the scope of this application note; a sample linker script is shown below:

```
 OUTPUT_FORMAT("elf32-bigmips")    /* Output
file Format */
  OUTPUT_ARCH(mips)
  _DYNAMIC_LINK = 0;
  SECTIONS
  {
    /* Read-only sections, merged into text
segment: */
   /* .text section begins at address 0xbfc00000
*/
    .text 0xbfc00000   :
    {
      _ftext = . ;
      *(.text)
        CREATE_OBJECT_SYMBOLS    /* Create a
symbol for each input file */
      _etext = .;
    }
    .init     ALIGN(8):
      { *(.init)     } =0
    .fini     ALIGN(8)   :
      { *(.fini)     } =0
    .ctors    ALIGN(8)   :
      { *(.ctors)    }
    .dtors    ALIGN(8)   :
      { *(.dtors)    }
    /* Read only data section, aligned on 8-
byte boundary */
    .rodata   ALIGN(8)   :
      { *(.rodata)   }
    .rodata1 ALIGN(8)    :
      {
        *(.rodata1)
        . = ALIGN(8);
      }
    .reginfo . : { *(.reginfo) }
    .data   . :
    {
      _fdata = . ;
      *(.data)
      CONSTRUCTORS
    }
    .data1 ALIGN(8)    :
      { *(.data1) }
    _gp = . + 0x8000;
    .lit8 . : { *(.lit8) }
    .lit4 . : { *(.lit4) }

    /* Keep the small data sections together,
so single-instruction offsets can access them
all,  and  initialized  data  all  before
uninitialized, so we can shorten the on-disk
segment size.  */

    .sdata    ALIGN(8)   : { *(.sdata) }
    _edata   =   .;
    __bss_start = 0xa0000200 ;
   .sbss    ALIGN(8)  : { *(.sbss) *(.scommon)
}
    .bss      0xa0000200 :
    {
    _fbss = .;
     *(.bss)
       *(COMMON)    /*  All  uninitialized  &
unallocated data from all
                    input files */
     _end = . ;
     end = . ;
    }

    /* Debug sections.  These should never be
loadable, but they must have
      zero addresses for the debuggers to work
correctly.  */
    .line          0 :
      { *(.line)           }
    .debug         0 :
      { *(.debug)          }
```

```
    .debug_sfnames     0 :
      { *(.debug_sfnames)      }
    .debug_srcinfo     0 :
      { *(.debug_srcinfo)      }
    .debug_macinfo     0 :
      { *(.debug_macinfo)      }
    .debug_pubnames    0 :
      { *(.debug_pubnames)     }
    .debug_aranges     0 :
      { *(.debug_aranges)      }
}
```

The linker puts "small" data into the small bss (.sbss) and small data (.sdata) sections. "Small" data is data that is smaller than a certain size. This size can be changed from the default 8 bytes using -G <size>. If -G 0 is used, nothing will be placed in .sbss and .sdata. Elements placed in .sbss and .sdata can be accessed in a single instruction using _gp that is appropriately set, resulting in fast data access.

2. Other options:

-nostdlib: This option tells the compiler not to look in the standard library search path for the specified library files. This is useful during embedded applications development, when the user needs to use non-standard libraries.

New instructions

The ELF-64 compiler implements the "branch likely" instructions in the Orion, when -mips2 or -mips3 is specified. When faced with a choice, the compiler attempts to use the conventional branch instruction and fill the branch with a branch independent operation. However, if it cannot do that, it converts the instruction to a branch likely instruction, and copies the target instruction into the branch delay slot.

Another set of instructions implemented by the compiler are those instructions that can give access to unaligned data: LWL, LWR, SWL, SWR, LDL, LDR, SDL, SDR. Using __attribute__ ((packed)) to declare a variable inside a C structure causes the compiler to generate the above instructions whenever the packed data element is accessed.

Assembler Directives

1. .set mips*n*

This directive allows the user to embed instructions from a higher level MIPS ISA, in a sequence of instructions that belong to another ISA. e.g. .set mips3 would allow the user to specifically enter ISA III instructions in ISA II or ISA I code. .set mips0 resets code generation to the default ISA.

When compiling an assembly file at a specific MIPS ISA, if instructions from a higher ISA are used, the assembler reports a warning, but assembles them anyway.

2. .set noreorder / .set reorder

Instructions in the block between the above directives are left as they are; no attempt is made to schedule them according to the pipeline requirements. It is the user's responsibility to see that the delay slots are properly filled, and hazards are taken care of. The assembler defaults to .set reorder.

3. .set noat

This directive instructs the assembler not to use the "at" register, which is used by the assembler to expand certain synthetic instructions. The assembler can be instructed to use the at register using .set at. All the instructions between the .set noat and .set at should be native instructions, or if synthetic instructions are used, should not require the at register. The assembler defaults to .set at.

Binary Utilities

1. nm

This utility is used to display the symbol table from an ELF file. It lists the symbols from an ELF object file, along with the virtual address for each symbol. It also displays the section (text, data, bss etc.) in which this symbol was located.

e.g. nm matmult > mat.nm

The following is a part of mat.nm

```
80012000 T start
80012000 A _ftext
800123b0 T main
80012710 T Mult
80012770 T Add
  .....
80018010   B matrix1
8001a720   B matrix2
8001ce30   B matrix3
```

The symbols tagged with a T are text symbols, those with a B are uninitialized data that are placed in the .bss section, and those with a D are initialized data, and are placed in the .data section. The symbols tagged with an A are absolute addresses.

2. objcopy

This utility is used to convert the ELF executable to S-record format, suitable for downloading to a board like the IDT evaluation board. This utility can also be used to build S-records from which PROMs can be built (using the -p option). This can also be used to create S-records for byte-wide PROMs (using the -b option), with an interleaving factor, if necessary (using the -i option).

e.g. objcopy -O srec matmult matmult.sre

e.g. objcopy -O srec -p -b 0 -i 1 myprom myprom.sre creates an S-record file that can be used to build the zeroth byte-slice of an interleaved PROM.

3. objdump

This utility displays information about ELF object files. It can be used to generate symbol table information, similar to nm, (using the -t switch), generate a disassembly listing (using the -d switch) or section header information (using the -h option).

e.g. objdump -d matmult > matmult.dis

The following is a part of matmult.dis:

```
80012000 <eprol> lui $gp,32770
80012004 <start+4> addiu $gp,$gp,-352
80012008 <start+8> lui $v0,32769
8001200c <start+c> addiu $v0,$v0,32544
80012010 <start+10> lui $v1,32770
80012014 <start+14> addiu $v1,$v1,-2032
```

e.g. objdump -h matmult > matmult.hdr
The following is a part of matmult.hdr:

```
  SECTION 2 [.text]     : size 00004f50 vma
80012000 align 2**4
   ALLOC, LOAD, CODE
  SECTION 3 [.rdata]    : size 00000330 vma
80016f50 align 2**4
   ALLOC, LOAD, READONLY, DATA
  SECTION 4 [.data]     : size 00000c20 vma
80017280 align 2**4
   ALLOC, LOAD, DATA
  SECTION 7 [.sdata]    : size 00000080 vma
80017ea0 align 2**4
   ALLOC, LOAD, DATA
  SECTION 8 [.sbss]     : size 00000060 vma
80017f20 align 2**4
   ALLOC
  SECTION 9 [.bss]      : size 00007890 vma
80017f80 align 2**4
```

4. **size**

This utility is used to display the sizes of all sections in an ELF file, in decimal or hex format. It also displays the total size of all sections in the ELF file.

e.g. size matmult displays:

```
text   data bss    dec     hex    filename
20304 4048 30960 55312  d810   matmult
```

Integrated Device Technology, Inc.

GDB - IDT/C™ 5.0 SOURCE LEVEL DEBUGGER

APPLICATION NOTE AN-128

By Upendra Kulkarni

INTRODUCTION

GDB, the source level debugger component of IDT/C™ 5.0, allows users to debug programs written in C and/or assembler code. GDB provides remote debugging capabilities, where the debugger itself runs on a computer such as a Sun or an IBM (compatible) PC (the HOST) and the code being debugged runs on a different system (the TARGET). The host and the target are connected by, and communicate through, a RS232C serial communication link. This application note is intended to provide some help in getting started with GDB; some of the most commonly asked questions are answered; some features not documented elsewhere are discussed.

Detailed command summaries of all GDB commands can be found in documentation of "IDT/C Cross Compiler System Version 5.0" specifically in "Cygnus Support GNU Developer's Kit Reference Manual Volume 1."

Detailed description of internal workings of GDB can be found in "GNU Debugger Internal Architecture" by Robert Pizzi (rpizzi@llnl.gov). This paper is also useful for people who wish to make enhancements to GDB. The paper is available via anonymous ftp from sisal.llnl.gov (128.115.19.65) in the pub/gdb Document directory.

BEFORE USING GDB

Hardware

GDB shipped with IDT/C 5.0 will work only when a serial port on the host is hooked up to the "tty0" port of the target board which must be running IDT/SIM™. "tty0" is the console port of the target board. Ordinarily, upon resetting the target board, a sign-on message is displayed on the console at "tty0". The sign-on message ends with the prompt "<IDT>" provided by IDT/SIM in the target board. GDB tends to send a "reset board" command over the serial link to the target in case of trouble. GDB, then, looks for the "<IDT>" string to return over the serial link. Upon receiving the "<IDT>" string GDB recognizes "wellness" of the target board and then sends the target board into "debug" mode. Obviously, this entire process will work only if the serial link from the host was connected to the console ("tty0") port of the target board.

Console output presented by the program being debugged on the target (using printf, for example), does not interfere with the GDB messages even though the same serial link is shared by both. The console outputs from user code are also displayed on the same screen as the GDB screen. The user is expected to be familiar enough with the source code being debugged, to be able to distinguish between GDB messages and messages printed by the code being debugged. "Hello, I reached here" is a message not likely to have come from GDB; but "Stack Full" could have come from either source.

Software

The serial port used by GDB on the host needs to be set for baud rate of 9600, 8 bits data, no parity, and 1 stop bit.

On the MIPS host, the serial device used for GDB needs to be in a mode other than the "respawn" mode. In the "respawn" mode, the operating system looks for a remote log in from the serial device. This conflicts with GDB activities and GDB fails to initialize.

On DOS hosts, the following two lines must be executed each time, just before executing GDB:

```
\idtc\asynctsr.com 1          (replace 1 by the COM port number used by GDB)
mode COM1: 9600,n,8,1         (replace COM1 by appropriate port name used by GDB)
```

On DOS hosts, the above lines can be put in a batch file for easy invocation. Note that asynctsr.com uses some memory every time you run it. You may wish to invoke it using "loadhigh" to minimize loss of conventional memory. If you start running out of memory, you may have to reboot the computer. You may have a TSR manager that knows how to remove older instances of asynctsr.com from memory.

On DOS hosts, it is important to note that GDB will not function at all unless SHARE is invoked manually or through the AUTOEXEC.BAT file.

GETTING STARTED

Init files

You may wish to use the "init files" feature of GDB to execute certain GDB commands automatically at the time of invocation of GDB. "Init files" on Sun and MIPS hosts are named ".gdbinit". You can have a "init file" in your home directory and another one in your current directory as well. The "init file" in the home is executed first and the one in the current directory is executed after that.

Currently, the "init file" feature is not implemented for DOS hosts. However, GDB can be invoked with the "-command filename" (or "-x filename") switch to achieve the same effect. Commands in the file "filename" will be executed automatically after starting GDB. In the future, the "init file" name for DOS is likely to be GDB.INI.

You may suppress the automatic execution of "init files" by invoking GDB with the -nx switch.

Preparing code for GDB

Source code must be compiled with one of the following switches:

-g - Same as -g2 below.
-g1 - Produces minimum information needed by GDB to be able to debug. No information about local variables or line numbers is generated.
-g2 - Produces maximum debugging information.
-g3 - Accepted but does not do anything more than -g2.

To debug malfunctioning code in initial stages, optimization level of zero (-O0 or no -O switch at all) is recommended during compiling and linking. This preserves the source code sequence and makes tracing through code easier. -O switches of all levels are, however, accepted in combination with -g switches of any level. Substantial experience with compiler optimizations is necessary to be able to debug optimized code.

Downloading code to target

Users familiar with debugging code in local environments will be tempted to follow the intuitively natural sequence of starting the debugging process: invoke the debugger, load the executable, initialize global settings, set a breakpoint, run. Strictly speaking this sequence also works for remote debugging. However, reversing the order of first two steps can result in substantial savings in time.

Invoking GDB first and using the "load" command from GDB to download code (to be debugged) to the target can take more than five times as much time as downloading code first and then invoking GDB. The download protocol used by the "load" command of GDB is very elaborate and time consuming. It is recommended, therefore, that the s-record file generated from the code to be debugged be downloaded first using the "load" command of IDT/SIM. This download process is no different from that employed during normal running of downloaded code. To run the code after downloading it to the board, the user would ordinarily enter the IDT/SIM command "go". The "go" command should NOT be entered if GDB is to be used for debugging.

After finishing the download the next step (which is optional but recommended) is to issue the debug command to the IDT/SIM. At the <IDT> prompt, enter:

debug tty0

Next, exit the monitor process or program - a terminal emulator in case of DOS, the "cu" command in case of Sun, etc. In the case of MIPS computers this is slightly confusing unless RISCWindows is being used.

In the case of MIPS computers, one needs to hook up a physically separate VT100 terminal to the "tty0" port of the board (as opposed to running a terminal emulator on the host). "tty1" of the target board is hooked up to the MIPS host. The file download takes place over "tty1". Once the download is over, the user is required to physically disconnect the VT100 terminal from the "tty0" port. The user is further required to move the cable from "tty1" to "tty0". If two serial ports of the host, and two serial cables, are available, then the cable hooked to "tty1" may be left where it is. The second serial cable from the host can be connected to "tty0" once the VT100 terminal is disconnected. Note that whichever serial device of the host is finally connected to "tty0", needs to be in a mode other than "respawn".

Note that once the code to be debugged has been downloaded to the target board, and the optional "debug tty0" command is issued, the target board must not be reset by the user using the reset button or in any other way. The next step at this point is to start GDB.

Invoking GDB

The last stage of an invocation of "gcc" is the linker stage. The linker stage is automatically invoked if the "-c" switch is not used while invoking "gcc". The linker can be invoked explicitly as "ld" (or "gld" via a link on most Unix systems). The linker produces a file referred to as the executable file (or code). In order to create downloadable s-record file from this executable file, use the "objcopy" binary utility.

To invoke GDB, simply enter:

gdb FILE

where, FILE is the name of the executable file as described in the previous paragraph.

After displaying the sign-on message, the (gdb) prompt will be displayed and GDB is ready to receive commands from the user.

At this point GDB knows nothing about the target. To introduce the target to GDB enter:

target mips com1 (if you are using DOS. Use the appropriate com port.)

OR

target mips /dev/ttya (if you are using Sunos4.1.3. Use the appropriate tty device.)

OR

target mips /dev/tty1 (if you are using Riscos5.01 on MIPS. Use the appropriate tty device.)

The system may respond with all of the following messages:

Timed out waiting for remote packet
Failed to initialize; trying to reset board
Remote MIPS debugging using com1/ttya/tty1

Ignore the first two lines. GDB has been successfully initialized and is now ready to receive commands.

Hitting a "return" at the (gdb) prompt repeats the last command issued to GDB. You can use short forms (first few letters) of all GDB commands as long as the number of letters are enough to uniquely identify the command and/or the

arguments.

MOST COMMONLY ASKED QUESTIONS

1. Is there on-line help on GDB?

Yes. At any point during debugging, you can receive on-line help on GDB commands. The starting point is to enter "help" at the (gdb) prompt. The main help screen gives instructions on how to obtain help in more detail on every specific command. Successive screens offer increasingly detailed help.

2. After debugging for some time, I forgot which part of the code is currently getting executed. How do I figure out where I am?

Use the GDB command "info frame". This command displays a lot of useful information including current frame pointer, stack pointer, stack level, pc location, saved registers, return address, addresses of local variables.

3. How do I display the current register values? Can I see special CP0 registers in IDT79R3081™ and IDT79R3041™ RISControllers™?

To see current register values, use the GDB command "info registers". This command will display all general purpose registers and all CP0 registers for the IDT79R3051™ RISController. CP0 registers unique to other RISControllers cannot be displayed using this release of GDB.

4. In the DOS platform compiler, what exactly does ASYNCTSR.COM do?

GDB functions in the "DOS extender" world, where there is no DOS I/O. It is difficult to get interrupts, in this case serial I/O, delivered in that region. ASYNCTSR.COM is, loosely speaking, a device driver which stays memory resident, and acts as the missing link between serial I/O and GDB. Its job is to intercept serial data and make it available to GDB.

5. Can I set breakpoints identified by line numbers in an assembler source file?

Unfortunately, GDB does not maintain any line number information about assembler source code. It is not possible to set a breakpoint using line numbers in assembler source as can be done with C source code.

However, there is a work-around, which is not very easy but can prove to be useful under some circumstances. To set a breakpoint at line number "linenum" in an assembler file "myasmfile.S" please follow these steps:

i. Add the following statement at the beginning of the file:
.file fileno "myasmfile.S" (fileno is any number)

ii. At line number "linenum - 1", add the following line:
.loc fileno linenum

iii. Now, while using GDB, to set a breakpoint at above location, at (gdb) prompt enter:
breakpoint "myasmfile.S":linenum

This procedure is rather cumbersome, especially if a number of breakpoints are desired in a number of assembler files. However, until a better solution becomes available, this will have to do.

6. Why does -g switch force less optimization while compiling even though the manual says that -O switch can be used along with -g switch?

Strictly speaking the -g switch need not perform less optimization if -O switch is used in conjunction with the -g switch. However, an exception is made to this rule in order to maintain compatibility with the assembler produced by Mips Corp. If -g switch is used, the branch delay slots are never filled with any useful instruction; they are always filled with a "nop", even if -O2 switch is specified. In the absence of -g switch, an effort is made to fill the delay slot with a useful instruction.

7. Why does GDB time out if the code is doing something useful? What does "set timeout" command do?

GDB assumes that if the target board does not respond to any query within 5 seconds, synchronization over the serial communication path is lost. Under such circumstances, GDB resets the board. This may be undesirable if the board was indeed expected to not respond within 5 seconds for a legitimate reason. To avoid such undesirable circumstances, the GDB manual describes a command called "set timeout seconds". A negative number of seconds in the command was expected to have GDB never time out under any circumstances, a feature useful in debugging real time applications where a breakpoint would be expected to be reached only under very rare conditions which occurred once every few hours or so.

Unfortunately, the "set timeout" command was not implemented correctly in GDB, and does not work as expected. In future releases of GDB, this command will be removed and time out will never occur. If the user believes that a malfunction on the target board is causing lack of response to GDB, the user will be expected to reset the target board manually. GDB will have no decision making intelligence regarding resetting the board.

Integrated Device Technology, Inc.

COPYING INITIALIZED DATA TO RAM

APPLICATION NOTE AN-132

By: Ketan Deshpande and Sugavaneswaran Subramanian

INTRODUCTION

Writing ROM-able code using IDT/C™ 5.0 or IDT/C 6.0 puts restrictions on initialized data declarations. Initialized data end up in ROM space, making it impossible to change such data during program execution. This restriction is neither obvious, nor acceptable to a number of C programmers. One technique to eliminate this restriction is explained in this application note. The most effective implementation requires modification to the C compiler utilities, which may be offered in future releases of IDT/C.

OVERVIEW

IDT's C Compiler tool chains IDT/C 5.0 and 6.0 provide a means of developing embedded applications based on the IDT R30xx and R4x00 RISControllers™. IDT/C 5.0 generates ECOFF format files; IDT/C 6.0 generates ELF files. For purposes of this discussion, both output file formats will be referred to as "executable". Any differences in formats / tool chains will be noted wherever appropriate.

IDT/C organizes the executable into sections by default, as shown below:

1) .text: All instructions from all source files.
2) .rdata (ECOFF) / .rodata (ELF): All initialized data that are declared constant. (Most commonly found elements here are strings.)
3) .data: All initialized data. Data may get moved between *.rdata* and *.data* depending on what the compiler believes is constant.
4) *.bss*: All uninitialized data.
5) *.sdata*: All initialized data smaller than the size specified by the -G option.
6) *.sbss*: All uninitialized data smaller than the size specified by the -G option.

The layout of sections and determining what exactly goes into which section can be controlled using a linker script file, and by adding -T<script filename> in the linker command line.

Both IDT/C 5.0 and 6.0 allow creation of user-defined sections and embedding user-defined symbols in the executable generated, using the linker script. This flexibility is key to the technique discussed below.

PROBLEM

Initialized data in the .data section get programmed into ROM space when the PROMs are created. This is the only way that the code can "remember" the initial values of all initialized data, in an embedded environment. However, this makes it impossible for the user to modify these values. The user can get around this by not initializing the variables at the point of declaration (making them uninitialized and thus forcing them into the *.bss* section) and then initializing them in code. The drawback of this approach is that the user needs to remember where to initialize each such data structure. Another way would be to have two structures: one initialized, one uninitialized, and in the code, copy the one in *.data* to the one in *.bss*. This method has speed and space disadvantages.

This Application Note describes a three-step method to overcome this problem. Briefly, the logic can be explained as (a) build the code assuming that the *.data* section will be in the RAM space; (b) in reality, burn the *.data* section in the ROM; (c) right at the start of code execution, move the *.data* section from ROM to RAM where the code expects it to be already.

Using IDT/C, the steps would be:

1. Link the executable program in such a way that the instructions look for *.data* section in the RAM address area.
2. Build S-records using a modified version of objcopy that relocates the *.data* section to ROM area while converting the executable to S-record. This "saves" the initialized contents of the *.data* section.
3. Make the startup code copy this relocated section from ROM area to its designated place in RAM area. This is the RAM address area where the instructions will be looking for the *.data* section (as explained in step 1 above). This method has been tested and found to work with relocating *.data* from IDT/sim™; it can be extended easily to cover .rdata / .rodata too.

ADVANTAGES

1. Allows software programmers to use initialized data without restrictions.
2. Removes the necessity for additional code/data spread out all over the application for modifying initialized data.
3. Speeds up program execution, since accesses that used to go to ROM are now directed to RAM.

DISADVANTAGES

1. Increased startup time because of the code to copy the *.data* section to RAM. However, this is only a one-time effort, and hence is not a major overhead.

STEPS INVOLVED

1. Determine what section(s) of the executable are to be relocated.
2. Modify the linker script to add informational sections (for objcopy) and symbols (for the startup code) that define the source and target of relocation.
3. Modify the startup code to copy data from the relocated address (ROM) to its real address (RAM).
4. Compile and link the application using the new linker script, such that the .data section now lies in RAM.
5. Use the version of objcopy that has support for this relocation, to build PROMs.

SECTIONS TO BE RELOCATED

Let us assume that only the *.data* section needs to be relocated.

MODIFYING THE LINKER SCRIPT

Linker scripts for IDT/C 5.0 and 6.0 are slightly different; the modifications done are very similar.

The following information needs to be inserted into the linker script to enable both objcopy and the startup code to perform the relocation and data movement.

a) Sections *.start*, *.endsect*:

This is done by inserting section lines in the linker script.

The program "objcopy" relocates all sections between these two sections to the address defined by _src_start.

b) Symbols _src_start, _src_end:

This is done by inserting symbol lines in the linker script.

The startup code copies data from _src_start to _tgt_start, until _src_end is reached.

c) Symbol _tgt_start:

This is done by inserting a symbol line in the linker script.

The startup code copies all data that was relocated, to this RAM address.

The modified linker scripts are listed on the following pages, with the changes highlighted.

Sample Linker Script for IDT/C 5.0:

```
OUTPUT_FORMAT("ecoff-bigmips")
ENTRY(start)
SECTIONS
{
  .text  0xbfc00000 : {
     _ftext = . ;
    *(.init)
     eprol  =  .;
    *(.text)
    *(.fini)
     etext  =  .;
     _etext  =  .;
  }
  .rdata  . : {
    *(.rdata)
  }

/* Relocate the sections between .start and
  .endsect, to begin from the current ad-
  dress */
  .start . : {}
  _src_start = . ;
  _tgt_start = 0xa0000200 ;
/* _tgt_start should be equal to the start
  of the .data section below */

  .data  0xa0000200 : {
    _fdata = .;
    *(.data)
    CONSTRUCTORS
   edata  =  .;
    _edata  =  .;
  }

/* OK, this is all we wanted to relocate */
  .endsect . : {}
  _src_end = . ;

  .reginfo . : {}
  .scommon . : {}
  .bss  . : {
   _fbss = .;
    *(.bss)
    *(COMMON)
  }
   end = .;
   _end = .;
}
```

Sample Linker script file for IDT/C 6.0:

```
OUTPUT_FORMAT("elf32-bigmips")
OUTPUT_ARCH(mips)
_DYNAMIC_LINK = 0;
SECTIONS
{
  /* Read-only sections, merged into text
  segment: */
  .text 0xbfc00000  :
  {
    _ftext = . ;
    *(.text)
    CREATE_OBJECT_SYMBOLS
    _etext = .;
  }
  .init     ALIGN(8)  : { *(.init)     } =0
  .fini     ALIGN(8)  : { *(.fini)     } =0
  .ctors    ALIGN(8)  : { *(.ctors)    }
  .dtors    ALIGN(8)  : { *(.dtors)    }
  .rodata   ALIGN(8)  : { *(.rodata)   }
  .rodata1  ALIGN(8)  :
    {
      *(.rodata1)
      . = ALIGN(8);
    }
  .reginfo . : { *(.reginfo) }

/* Relocate the sections between .start and
  .endsect, to begin from the current ad-
  dress */
  .start . : {}
  _src_start = . ;
  _tgt_start = 0xa0000200 ;
/* _tgt_start should be equal to the start
  of the .data section below */

  .data  0xa0000200 :
  {
    _fdata = . ;
    *(.data)
```

```
    CONSTRUCTORS
  }
  .data1 ALIGN(8)   : { *(.data1) }
  _gp = . + 0x8000;
  .lit8 . : { *(.lit8) }
  .lit4 . : { *(.lit4) }
  .sdata   ALIGN(8)   : { *(.sdata) }
  _edata  =  .;

/* OK, this is all we wanted to relocate */
  .endsect . : {}
  _src_end = . ;

  __bss_start = . ;
  .sbss    ALIGN(8)   : { *(.sbss)
 *(.scommon) }
  .bss      . :
  {
  _fbss = .;
   *(.bss)
   *(COMMON)
   _end = . ;
   end = . ;
  }

  .line                   0 : { *(.line)
  }
  .debug                  0 : { *(.debug)
  }
  .debug_sfnames          0 : {
 *(.debug_sfnames)          }
  .debug_srcinfo          0 : {
 *(.debug_srcinfo)          }
  .debug_macinfo          0 : {
 *(.debug_macinfo)          }
  .debug_pubnames         0 : {
 *(.debug_pubnames)         }
  .debug_aranges          0 : {
 *(.debug_aranges)          }
}
```

Modifying the startup code

Typically, embedded applications have code that performs CPU control register initialization, cache flushing, memory sizing, initializing .bss etc. With the *.data* section in its new positions in ROM, the code will still look to RAM addresses for initialized data. Before any such references are attempted, the .data section should be copied out into it's real place. A good place to do this is usually after *.bss* initialization. The code segment below demonstrates how this can be done. The same code can be used for IDTC/5.0 and 6.0; though for the R4x00 processors, the user may want to use double-word loads and stores for faster execution.

```
      la   t0, _src_start
      la   t1, _tgt_start
      la   t2, _src_end
2:    lw   t3, 0(t0)
      nop
      sw     t3, 0(t1)
      addu   t0, 4
      addu   t1, 4
      blt    t1, t2, 2b
      nop
```

Modification to OBJCOPY

The binary utility "objcopy" needs to be modified to make it intelligent enough to recognize the sections that the linker script was asked to create, and to move the appropriate sections to their temporary PROM addresses. Most of the code modifications needed to perform this movement are in the function setup_section() in the file objcopy.c (the main source code file for the objcopy utility), and are shown on the next page, in boldface. Some adjacent code is shown for reference. Initialization of the variables may not be shown explicitly; it is mentioned wherever appropriate.

```
  setup_section(......)
  {
  ...../* Original variable declarations here
*/
    int sec_addr;
    static int new_data_addr = 0;
    static int move_section = FALSE;
  ......
  ....../* Original code here */
    if (!bfd_set_section_size (obfd,
                    osection,
                      bfd_section_size (ibfd,
isection)))
      {
        err = "size";
        goto loser;
      }

    /* start_address = bfd_get_start_address
(ibfd);
       in copy_object() */
      if  (!new_data_addr)  new_data_addr  =
start_address;
    new_data_addr += bfd_section_size (ibfd,
isection);

   /* Got section .start? Now remember current
address
      and keep track of new relocation address
*/
     if  (!strcmp(bfd_get_section_name(ibfd,
isection),
        ".start"))
      move_section = TRUE;
    /* Got section .endsect? Stop relocation
*/
   else if (!strcmp(bfd_get_section_name(ibfd,
```

```
              isection), ".endsect"))
            move_section = FALSE;

    if (move_section) sec_addr = new_data_addr;
     else sec_addr = bfd_section_vma (ibfd,
isection);

    /* Actually do the relocation */
    if (bfd_set_section_vma (obfd, osection,
sec_addr)
        == false)
      {
        err = "vma";
        goto loser;
      }

    if (bfd_set_section_alignment (obfd,
                  osection,
                      bfd_section_alignment
(ibfd, isection))
        == false)
      {
        err = "alignment";
        goto loser;
      }

  ......
```

Compile, link the application and build PROMs

This can be done in the usual manner. The scripts shown above setup the *.data* section to reside in RAM area. The new version of objcopy with this option may be available in future releases of IDT/C.

SUMMARY

This application note described a technique that relocated certain sections to ROM and then copied them to their designated locations in RAM. This method has been demonstrated on the *.data* section; it can very easily be extended to include other sections too.

The advantages are: provide C programmers with the ability to use initialized variables much more freely, removal of the need for extra code or data, faster access without requiring any extra ROM space.

Integrated Device Technology, Inc.

SCATTER LINKER

APPLICATION NOTE AN-133

By: Sugavaneswaran (Sugan) Subramanian

INTRODUCTION

In general, a compiler has four major components. They are Preprocessor, Compiler, Assembler, and Linker. This application note explains the "scatter" feature of SGI/MIPS compiler in the context of MIPS R3000/R4000 RISC processors.

What role does a linker play in an embedded environment?

In the embedded environment, the linker plays a major role in laying out the application code into RAM/ROM of the target system in the most productive manner. In embedded applications, the code section and the data section reside in known fixed memory locations. All compilers that create applications for embedded systems have a mechanism to specify the start address for the code section. They also give the programmer a choice of either making the data section follow the code section or to start the data section at an address before the start of the code section or to start the data section at an address after the end of the code section. The linker generally, lets the programmer layout the code in the following manner:

1. One uncached code section and one uncached data section
2. One cached code section and one uncached data section
3. One uncached code section and one cached data section
4. One cached code section and one cached data section

A scatter linker offers more choices, and is, therefore, an integral part of the new SGI/MIPS cross-compiler for R3000/R4000 target running on SUN SPARC host.

Why do we need multiple sections of code and data?

Embedded systems usually have slower main memory interface than desktop systems. In such systems that have a MIPS R3000/R4000 based (RISC) CPU, the code that resides in the instruction cache executes many times faster than the code that is executed from an uncached space (Main Memory). Also, the data that reside in the data cache can be accessed many times faster the data that reside in Main Memory. The following cases may arise :

1. In some code-intensive applications , to get the most optimal performance out of an instruction cache, the programmer must have a section of code that is most frequently executed and that is small enough to fit inside the instruction cache, always cached, and the rest of the code in Main Memory.
2. In some data-intensive applications, to get the most optimal performance out of a data cache, the programmer must have a section of data that is most frequently accessed and that is small enough to fit inside the data cache, always cached, and the rest of the data in Main Memory.
3. In some applications that are code and data intensive, a combination of the previous two mechanisms should be applied.

What is a Scatter Linker?

A scatter linker is a linker that lets the programmer develop an embedded application such that it has one or more text sections and one or more data sections. The process of creating an application that has one or more text sections and one or more data sections is called scattering. This is usually done with the help of a linker script language. In this application note, various features of scatter linker supported by the new SGI/MIPS cross compiler are described.

The new compiler uses a linker script language to layout the executable code at the programmer's demand. A switch in the link line of the compilation lets you specify the file that contains the linker script having the layout information of the executable code. A complete description of the linker script language is beyond the scope of this application note. However, an illustration of one simple linker script and one complex linker script presented here is believed to be sufficient to provide an introduction to the scatter linker.

How do you invoke linker-script in a link line of the SGI/MIPS compiler?

The following switches are useful when invoking a linker-script:

"-elspec" tells the linker that the following element is going to be the name of a ASCII-text file containing linker-script

"-rom" tells the linker not to pad any sections with UNIX-based page size

"-elsmap" tells the linker to generate the map of linker script to standard output (screen). The output can be redirected to a file. This switch can be used without "-elspec" and "-rom". Gives an elaborate description of the layout of various sections of all the objects in the linker line in a pseudo linker-script form.

example1:

```
ldr4000 -elspec simple_script  -rom -elsmap -o
app1_simple crt0.o file1.o file2.o        >
simple_script.map
```

example2:

```
ldr4000 -elspec complex_script  -rom -elsmap -o
app1_complex crt0.o file1.o    file2.o   >
simple_script.map
```

How do you create linker script for executable code that has one text section and one data section?

By default, all executable code that is created by the SGI/MIPS cross compiler has the following set of valid sections:
1. A ".text" section containing executable instructions.
2. A ".MIPS.option" section
3. A ".reginfo" section
4. A ".rodata" section containing read-only data. Read only data include immediate values, char constants, and string constants
5. A ".data" section containing intialized data. Initialized data include initialized global variables, initialized variables with "static" type, and variable with "const" type.
6. A ".bss" section containing uninitialized data. Uninitialized data include uninitialized global variables and function names.

A segment is a collection of sections. Sections 1-4 are considered loadable, readable and executable and are grouped together and put into a segment with unique attributes. Sections 5-6 are considered loadable, readable, and writable and are grouped together and put into another segment with unique attributes. This is the default layout. The programmer is free to change the attributes and/or the contents of a segment.

A simple script includes all the default sections.

The following is an example:

Example1 (simple_script) :

```
# Creates an executable with one text section and one data
  section.
# It has the text section start at 0xa0020000 and has the
  data section
# following the text section
# comment is preceded by a #
# file:  simple_script

# The following segment contains elements that are
  readable and
# executable
  beginseg
     segtype LOAD  #  Makes the segment loadable
                   #  A segment is considered loadable
                   #  if its contents can be put in a valid
                      section
                   #  of main memory or cache in the
                      target
                   #  system

     segflags R X  #  Makes the segment readable and
                   #  Executable

     vaddr 0xa0020000 # Gives the start address for the
                        segment
                      # This a valid virtual address in the
                        target
                      # system

     segalign 0x1000 # specifies the UNIX OS based
                       page
                     # alignment between the current
                       segment
                     # and the following segment. It is
                       ignored
                     # when using "-rom" switch in the
                       link line

     contents        # specifies the sections that are
                       going to
                     # be put in this segment
# The following link-script command "noheaders"
# makes sure that the section header information is
# skipped in the executable code
     noheaders
     default         #  Includes are the all the loadable
                     #  sections that are readable and
                     #  executable
  endseg

# The following segment contains elements that are
  readable and writable
  beginseg
     segtype LOAD
     segflags R W  #  Makes the segment readable and
                   #  Writable
# If you want to make the data section start at a new
  address,
# then enter the stuff within quotes in the following line
# "vaddr <data-start-addr>".
# where <data-start-addr> is a valid hex number i.e. 0x<8-
  hexdigits>
     segalign 0x1000
     contents
     default
  endseg
```

How do you create linker script for executable code that has more than one text section and more than one data section?

Executable code that is created by SGI/MIPS cross compiler has the following set of valid sections:
1. One or more sections containing executable code with unique attributes as long as they are all within 256 MegaByte boundary. Due to the boundary limitation on code section, one cannot have a section of executable code in Cache and the rest of the executable code in Main Memory
2. A ".MIPS.option" section
3. A ".reginfo" section
4. One or more sections containing read only data with unique attributes.
5. One or more sections containing initialized data with unique attributes.

6. One or more sections containing uninitialized data with unique attributes.

Whenever you want to create an application that needs to have more than one code section and more than one data section, it is necessary that you know the unique attributes that identify them. And, if two sections have similar attributes, they should have different names. Moreover, you can have a code section inside a data section or a data section inside a code section.

The executable file format for the executable code is ELF. Each one of the ELF sections are identified by

1. Section type
2. Section flag

The following is the list of valid ELF sections with the attributes for them:

1. code section
   Section type: PROGBITS
   PROGBITS ==> Contents are loaded into the memory before execution
   Section flag: ALLOC EXECINSTR
   ALLOC ==> Contents have a valid section of memory in the target system
   EXECINSTR ==> Contents are executable machine instructions
2. read only data section
   Section type: PROGBITS
   Section flag: ALLOC
3. initialized data section
   Section type: PROGBITS
   Section flag: ALLOC WRITE
4. uninitialized data section
   Section type: NOBITS
   NOBITS ==> Contents are not loaded into the memory before execution
   Section flag: ALLOC WRITE

The following example describes how to create linker script that hasmultiple text and data sections for the executable code.

Example2 (complex_script):

```
  beginseg
     segtype LOAD
     segflags R X
     vaddr 0xa0020000
     segalign 0x1000
     contents
     noheaders
     beginscn .text
# scntype specifies section type for code section to be
  PROGBITS
     scntype PROGBITS
# scntype specifies section flag for code section to be
  ALLOC
#                                          EXECINSTR
     scnflags ALLOC EXECINSTR
     scnalign 4
# .text is the name of the section
# It contains code section for object crt0.o
#       read only data section (.rodata) for all objects
#       .MIPS.options section for all objects
#       .reginfo section for all objects
     section .text in crt0.o
     section .rodata
     section .MIPS.options
     section .reginfo
     endscn
endseg
beginseg
  segtype LOAD
  segflags R W
  segalign 0x1000
  contents
     beginscn .data
     scntype  PROGBITS
     scnflags ALLOC WRITE
     scnalign 4
# .data is the name of the section
# It contains data section for object crt0.o
     section .data in crt0.o
     endscn
endseg
beginseg
  segtype LOAD
  segflags R X
  segalign 0x1000
  contents
     beginscn .text2
     scntype  PROGBITS
     scnflags ALLOC EXECINSTR
     scnalign 4
# .text2 is the name of the section
# It is the second code section
# It contains text section for object file1.o
# It contains text section for object file2.o
     section .text in file1.o
     section .text in file2.o
     endscn
endseg
beginseg
  segtype LOAD
  segflags R W
  segalign 0x1000
  contents
     beginscn .data2
     scntype  PROGBITS
     scnflags ALLOC WRITE
     scnalign 4
# .data2 is the name of the section
# It contains initialized data section for object file1.o
# It contains initialized data section for object file2.o
     section .data in file1.o
     section .data in file2.o
     endscn
endseg
beginseg
  segtype LOAD
  segflags R W
```

```
        segalign 0x1000
    contents
        beginscn .bss
        scntype  NOBITS
        scnflags ALLOC WRITE
        scnalign 4
# .bss is the name of the section
# It contains uninitialized data section for all objects
        section .bss
        endscn
endseg
```

CONCLUSION:

In embedded systems with SGI/MIPS R3000/R4000 CPU, having some sections of code in Instruction cache and some sections of code in Main Memory, and some portion of data in data cache and the rest in Main Memory can greatly improve performance in data intensive and/or code intensive applications. The scatter linker is an integral part of the new SGI/MIPS compiler and has the ability to produce such code.

Integrated Device Technology, Inc.

SETTING UP THE SGI INDY™ AS A DOWNLOAD PLATFORM FOR IDT'S RISC EVAL BOARDS

TECHNICAL NOTE TN-16

By Ketan Deshpande

This note explains how to set up the SGI INDY™ workstation as a platform for downloading code onto an IDT evaluation board.

You will need the following items:

1. An SGI INDY workstation running IRIX™ 5.1.1 or higher, with at least one serial port. Two ports are necessary if you want to do terminal emulation from the SIM.
2. A RISC evaluation board from IDT. The board must have IDT/SIM™ 4.0 or later.
3. A software development tool chain that will produce executable code for MIPS RISC processors, and S-records from that executable code.
4. The UUCP utility on the INDY workstation. If this is not installed, it should be available on one of the CD-ROMs containing IRIX software. The UUCP utility is available on the IRIX Operating System CD-ROM as package eoe2.sw.uucp.
5. An RS232C cable that plugs into the serial port on an INDY. MINIDIN 8 cables can be used. One cable is sufficient, two are needed if you want to do terminal emulation.

ASSUMPTION

It is assumed that:

1. The software tool chain is set up on the workstation properly.
2. You have created a small software program, and compiled and linked it with the appropriate libraries from your tool chain.
3. You also have created S-record files from the executable. You will need to download the S-record file to the evaluation board.

HARDWARE

To set up the hardware for downloading the S-record file from the INDY to the target board, you need to set up a serial link between the workstation and the board. This is a one-time effort only.

Locate the serial ports on the back of the INDY. They are next to the mouse and keyboard ports, and are marked "1" and "2". By default, IDT/SIM uses the tty0 port of the board to communicate with a terminal. (To display prompts, echo the keyboard input, etc.) Hence, this port needs to be connected to the host. Locate the tty0 port of the IDT evaluation board. Connect it to a serial port on the INDY using the serial cable mentioned above. Once the cable is hooked up, connect the power supply to the board and switch it on.

Let us assume for further discussion that the INDY's serial port 2 has been connected to the board's tty0.

SOFTWARE

Setting up the serial port

In the INDY, serial port "n" is associated with the device file /dev/ttydn. Make sure the access permissions for the file /dev/ttyd2 are set to "rw-rw-rw-". You can view the current mode using "ls -l /dev/ttyd2". If the mode is different, you will need to log in as root and use "chmod 666 /dev/ttyd2" to set the proper mode. Also, still working as root, in the /etc/inittab file, look for the line that has ttyd2. If necessary, change it so that it the third field is "off", rather than "respawn" or "on". The line now should look like this:

```
t2:23:off:/sbin/getty -N ttyd2 co_9600
```

If you want to do terminal emulation from the SIM (not really necessary since you have the capability to open multiple windows on the INDY), connect the serial port 1 to tty2 (or AUX) on the evaluation board and make sure that the line (in /etc/inittab) for tty1 looks like the one below:

```
t1:23:respawn:/sbin/getty ttyd1 co_9600
```

After changing the /etc/inittab file, type "telinit q" to make those changes known to IRIX.

Setting up UUCP

Log in as root on the INDY. This can be done easily by opening another window and using the command "su - root" or "login root". Change current directory to the UUCP directory (/etc/uucp or /usr/lib/uucp).

Add the following line to the file Systems:

```
board Any dev Any
```

Add the following line to the file Devices:

```
dev ttyd1 - 9600 direct
```

The above two lines inform UUCP that there is a device called board of type dev that is directly connected to ttyd1 and communicates at 9600 baud.

The above changes need to be made only once, and now it is assumed that there is a working serial connection between the INDY and the board.

Now, working as yourself (not as root), change directory to where your S-Record file exists, and at the shell prompt, type:

```
cu board
```

The system will respond in a minute, saying "Connected".

Press <RET> a couple of times. You should see the IDT/SIM prompt "<IDT>". If you do not see this prompt, the serial connection has not been set up properly. Please go over the steps mentioned above and check that all steps have been taken. You may try resetting the board too.

To download over the serial port, at the SIM prompt, type:

```
load -a tty0
```

The cursor will go to the next line and freeze there. Now, type:

```
~$cat your-srec-filename
```

In the process of typing, you will notice that your hostname suddenly appears in between the ~ and the $. This is done by cu, and is expected. Another way to download over the serial port would be as follows:

At the SIM prompt, type:

```
load -a tty0
```

In another window, type:

```
cat your-srec-filename > /dev/ttyd2
```

After you type in the command and press Enter, there may be a pause of a few seconds, and after that you will see rows of dots showing that the file is being downloaded.

After the download is complete, a message like the following will be displayed:

```
Done. (num) records, initial pc: (address)
```

and the <IDT> prompt will return. At this point you can start using the SIM commands again.

To disconnect from the board, type "~" and press Enter. This is a tilde (~) followed by a period (.). The display will say Disconnected and the shell prompt will reappear.

Integrated Device Technology, Inc.

USING HP'S R4X00 DISSEMBLER SOCKET FOR H/W AND S/W DEBUG

TECHNICAL NOTE TN-17

By Sami Khan

INTRODUCTION

The IDT79R4600™ is the newest member of IDT's RISController™ family and provides full applications upward compatibility with the earlier members of the MIPS family.

This note explains the use of Hewlett-Packard Logic Analyzer preprocessor pod for R4000PC as a debugging tool. An important part of system design involves choosing the correct design and debugging tools which can help the system designers debug their system with ease and efficiency.

THE HP E2438A PREPROCESSOR

The HP E2438 Preprocessor provides a complete interface for state analysis between any target system and an HP 1660A™, HP 16540/16541A,D™, or HP 16550 Logic Analyzer™. The package includes a preprocessor socket and system software. The software includes configuration files and disassembler software for both little endian and big endian systems.

SYSTEM SETUP

The preprocessor socket has two configuration switches. Switch SW1 is used to select processor operating frequency. If the target system is operating above 35Mhz, SW1 must be in the ON position. If the target is operating between 10Mhz and 35Mhz, switch SW1 must be OFF.

Similarly, switch SW2 is used to select interface clocking mode. The ON position is for state-per-clock mode, which means that every microprocessor clock cycle will clock the logic analyzer. When SW2 is in the OFF position, the interface is in state-per-bus-cycle mode, meaning that only valid data transfers (microprocessor bus cycles) are clocked into the logic analyzer. Note that in the state-per-clock mode, disassembly is not available.

The setup involves plugging the preprocessor interface connector into the microprocessor socket. Please refer to the "Preprocessors Interface User's Guide" from HP for connecting the interface socket to the logic analyzer pods. Next, load the configuration file into the logic analyzer. This would automatically load the inverse assembler file for a big endian system (file IR4K_BE). For a little endian system, load inverse assembler file IR4K_LE. If the configuration file is saved to the disk with the current inverse assembler, the next time that configuration is loaded, the current inverse assembler will also automatically be loaded with it.

Setting up the logic analyzer also involves setting up the triggering point. The triggering point depends on the type of cycle the system designer is trying to capture. The triggering point can be set on any Address, Data, or Control signals or any bus activity.

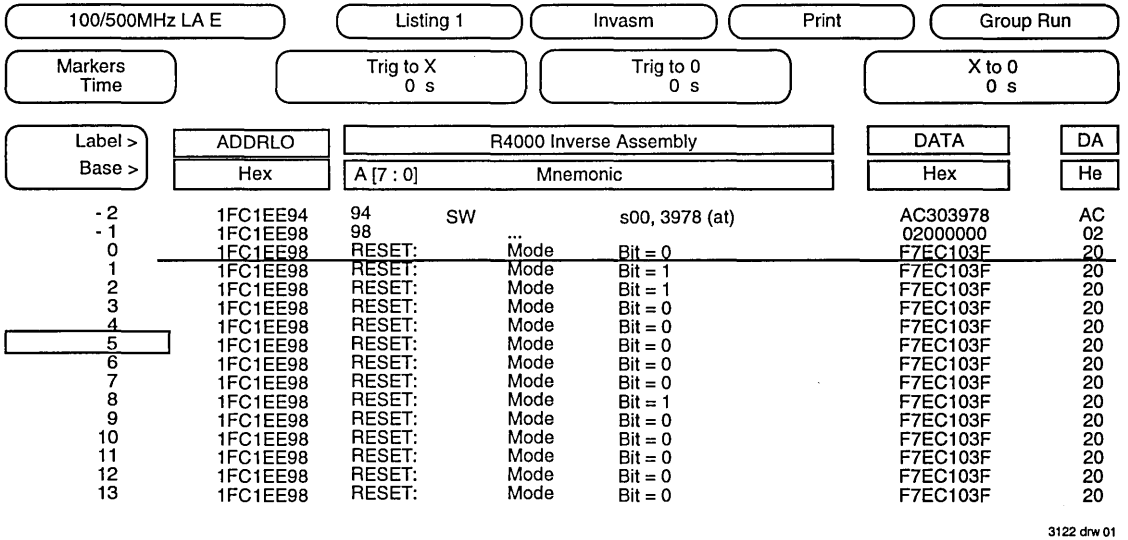


Figure 1

DEBUGGING AN R4600 SYSTEM

Debugging of an R4600 system starts with the debugging of CPU's reset interface. This involves debugging of warm and cold reset logic, debugging the mode bit interface and fetching the reset vector.

Debugging the CPU reset interface involves getting the CPU to read the correct mode bits. To capture mode bits using the disassembler software, trigger the logic analyzer on the "COLD RESET" signal being low. After 256 mode bits read cycles, the processor should fetch the reset vector from virtual address 0xbfc00000 (0x1fc00000 physical). Figures 1 and 2 show the mode bits read sequence at the reset interface. After 256 read cycles, the processor fetches the reset vector from physical address 0x1fc00000, as shown in Figure 2.

The preprocessor interface can then be used for software debugging. The instruction can be run cached or uncached. For full execution trace, the software must be run uncached, so that the disassembler sees all executed instructions. The logic analyzer captures all bus activity and every instruction executed can be seen on the display.

Figures 3 and 4 explain the execution trace for uncached instructions and data. The code starts executing from physical address 0x00020000 (0xA0020000 virtual). Sequential fetches can be seen on the bus interface which explains the execution flow of the code.

When configured as a timing analyzer, the timing relationship for CPU signals can be read which is helpful in debugging hardware, as shown in Figure 5.

Code can be run cached, but full execution trace information will not be available. Internal caches are used most of the time and only external bus activity can be seen by the analyzer. Mostly, this external activity corresponds to the refilling of the internal caches. Only during the initial filling of caches, the analyzer can capture all the bus activity. Moreover, this does not guarantee that every instruction fetched is executed. For example, CPU supports sub-block ordering for block refill. This only guarantees that the first instruction for a block refill will be executed. Figures 6 and 7 explain this fact. Figure 6 shows the initial cache refill cycles for the code fetch. Once caches are filled, code continues to execute from it and the only bus activities that can be captured by the analyzer are the data load or store operations or cache refill operations as shown in Figure 7.

Note that whether or not the software is executed through the cache, the order of loads and stores seen on the bus will be the same as the order in which they are executed by the CPU. The R4600 insures strong ordering, which guarantees this order. Also note, however, that the R4600 integrates an on-chip write buffer. Thus, data being written may not be due to the most recent store instruction executed, but rather due to a previous store instruction which was executed, and whose data was captured by the on-chip write buffer. Once a load operation is required, all such pending writes will be executed prior to the data load.

SUMMARY

The use of HP's preprocessor socket is one example of the debugging tools for an R4600 system. The logic analyzer is useful for initial debugging of the system and it can be used further for software and hardware debug. The Disassembler formats logic analyzer state traces into assembly level mnemonics to allow easier user interpretation.

When used in conjunction with tools such as an embedded monitor program, a ROM emulator, and/or remote target high-level language debug tools, overall development time can be reduced substantially.

100/500MHz LA E | Listing 1 | Invasm | Print | Group Run

Markers Time | Trig to X 0 s | Trig to 0 0 s | X to 0 0 s

Label > Base >	ADDRLO Hex	R4000 Inverse Assembly A [7 : 0]	Mnemonic	STAT Hex	DA He
252	00012614	RESET:	Mode Bit = 0	FF1BCBF4	02
253	00012614	RESET:	Mode Bit = 0	FF1BCBF4	02
254	00012614	RESET:	Mode Bit = 0	FF1BCBF4	02
255	00012614	RESET:	Mode Bit = 0	FF1BCBF4	02
256	1FC00000	00	J 01FC00388	101BFBE0	0B
257	1FC00004	04	NOP	101BFBE4	00
258	1FC00388	88	LUI v00, 2001	101BFBE8	3C
259	1FC0038C	8C	MTCO v00, SR	101BFBEC	40
260	1FC00390	90	MTCO r00, Cause	101BFBE0	40
261	1FC00394	94	ADDIU v00, r00, 0003	101BFBE4	24
262	1FC00394	98	MTCO v00, r16, rsvd	101BFBE8	40
263	1FC0039C	9C	ADDIU v00, r00, 0080	101BFBEC	24
264	1FC003A0	A0	BNE v00, r00, 01FC003A0	101BFBE0	14
265	1FC003A4	A4	ADDIU v00, v00, FFFF	101BFBE4	24
266	1FC003A0	A0	BNE v00, r00, 01FC003A0	101BFBE0	14
267	1FC003A4	A4	ADDIU v00, v00, FFFF	101BFBE4	24

3122 drw 02

Figure 2

100/500MHz LA E | Listing 1 | Invasm | Print | Run

Markers Time

Acquisition Time
22 Jun 1994 16:56:33

Label > Base >	ADDRLO Hex	R4000 Inverse Assembly A [7 : 0]	Mnemonic		STAT Hex	DA He
-1	00001604	08	NOP		001BEFE4	00
0	00020000	00	MTCO	v00, SR	001BEFE0	3C
1	00020004	04	LUI	v00, 2001	001BEFE4	3C
2	00020008	08	ADDIU	v00, R00, 0003	001BEFE8	40
3	002000C	0C	MTCO	r00, Cause	001BEFEC	40
4	00020010	10	ADDIU	v00, r00, 0080	001BEFE0	40
5	0020014	14	MTCO	v00, r16, rsvd	001BEFE4	40
6	00020018	18	ADDIU	v00, v00, FFFF	001BEFE8	14
7	0002001C	1C	BNE	v00, r00, 00002001C	001BEFEC	14
8	00020018	18	LUI	v00, v00, FFFF	001BEFE8	14
9	0002001C	1C	LUI	v00, r00, 00002001C	001BEFEC	14
10	00020018	18	SW	v00, v00, FFFF	001BEFE8	14
11	0002001C	1C	ADDIU	v00, r00, 00002001C	001BEFEC	14
12	00020018	18	LW	v00, v00, FFFF	001BEFE8	14
13	0002001C	1C	SW	v00, r00, 00002001C	001BEFEC	14
14	00020018	18	BEQ	v00, v00, FFFF	001BEFE8	14

3122 drw 03

Figure 3

100/500MHz LA E | Listing 1 | Invasm | Print | Run

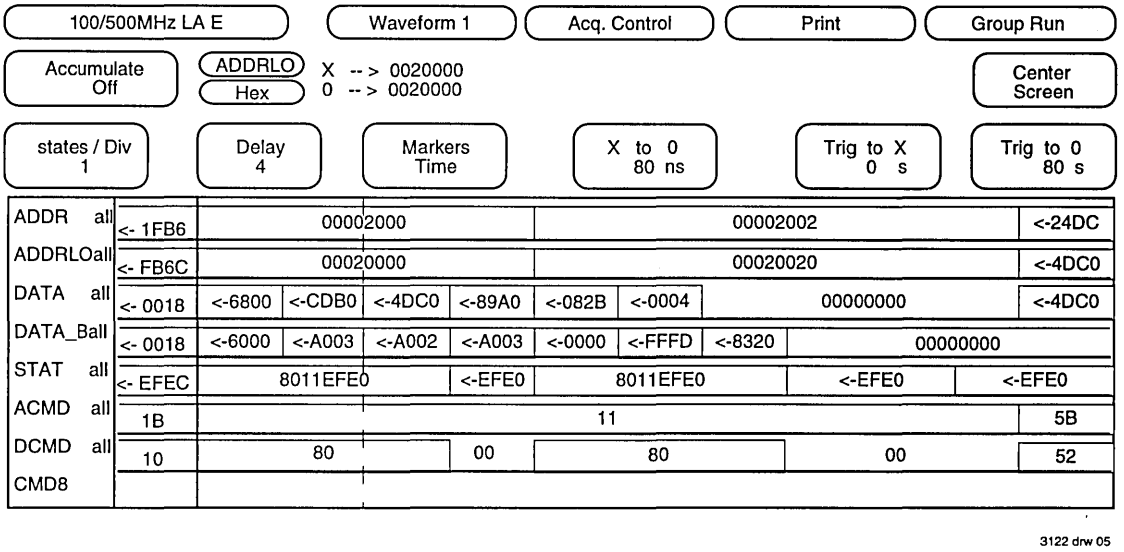
Markers Off

Acquisition Time
23 Jun 1994 08:45:49

Label > Base >	ADDRLO Hex	R4000 Inverse Assembly A [7 : 0]	Mnemonic		STAT Hex	DA He
264	00020020	20	LUI	t00, AAAA	001BEFE0	3C
265	00020024	24	LUI	v00, A000	001BEFE4	3C
266	00020028	28	SW	t00, 0000 (v00)	001BEFE8	25
267	0002002C	2C	ADDIU	t00, t00, 5555	001BEFEC	25
268	00020030	30	LW	t01, 0000 (v00)	001BEFE0	AC
269	00000000	00	mem write	AAAA5555- - - - - - - -	405BEFC0	AA
270	00020034	38	SW	r00, 0008, (v00)	001BEFE4	AC
271	00000008	08	mem write	00000000 - - - - - - - -	405BEFC8	00
272	00000000	00	mem read	AAAA5555- - - - - - - -	001BEFE0	AA
273	00020038	38	BEQ	t01, t00, 000020048	001BEFE8	00
274	0002003C	3C	NOP		001BEFEC	00
275	00020040	40	BEQ	r00, r00, 000020040	001BEFE0	00
276	0002004C	4C	NOP		001BEFEC	00
277	00020050	50	SW	r00, 0008 (v00)	001BEFE0	AC
278	00020054	54	SW	t00, 0010, (v00)	001BEFE4	AC
279	00000010	10	mem write	FFFFFFFF- - - - - - - -	001BEFE0	FF

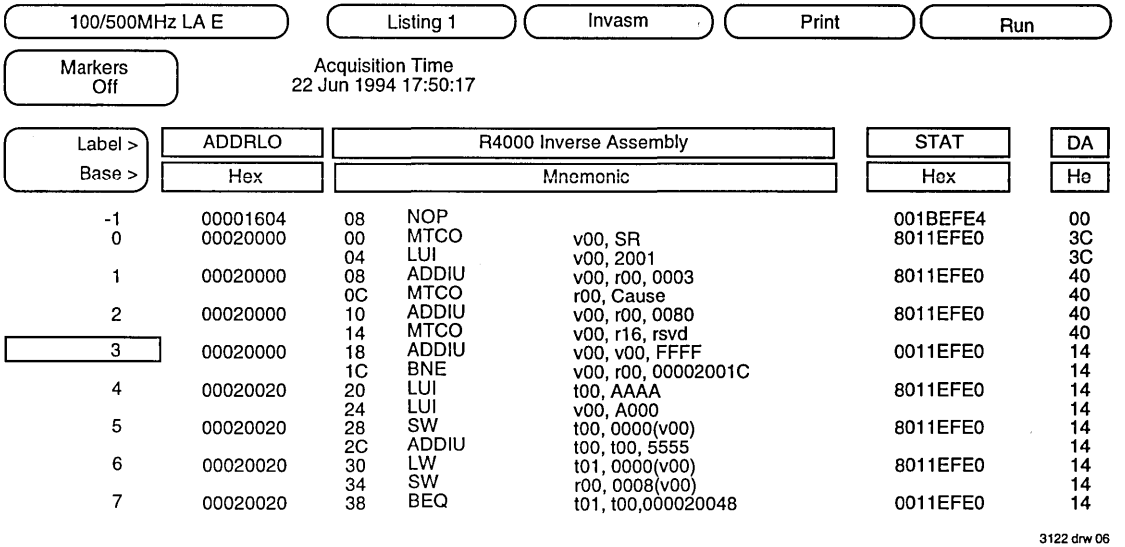
3122 drw 04

Figure 4



3122 drw 05

Figure 5



3122 drw 06

Figure 6

100/500MHz LA E | Listing 1 | Invasm | Print | Run

Markers Off

Acquisition Time
22 Jun 1994 16:30:45

Label > / Base >	ADDRLO Hex	R4000 Inverse Assembly A [7 : 0]	Mnemonic		STAT Hex	DA He
3	00020000	18	ADDIU	v00, v00, FFFF	0011EFE0	14
		1C	BNE	v00, r00, 00002001C		
4	00020020	20	LUI	t00, AAAA	8011EFE0	3C
		24	LUI	v00, A000		
5	00020020	28	SW	t00, 0000(v00)	8011EFE0	25
		2C	ADDIU	t00, t00, 5555		
6	00020020	30	LW	t01, 0000(v00)	8011EFE0	AC
		34	SW	r00, 0008,(v00)		
7	00020020	38	BEQ	t01, t00, 000020048	0011EFE0	00
		3C	NOP			
8	00000000	00	mem write	AAAA5555 ----------	405BEFC0	AA
9	00000008	08	mem write	00000000 ----------	505BEFC8	00
10	00000000	00	mem read	AAAA5555 ----------	001BEFE0	AA
11	00020040	40	BEQ	r00, r00, 000020040	8011EFE0	00
		44	NOP			
12	00020040	40	ADDIU	t00, r00, FFFF	8011EFE0	00

3122 drw 07

Figure 7

Integrated Device Technology, Inc.

EMBEDDING ASSEMBLY INSTRUCTIONS INSIDE C-SOURCE CODE

TECHNICAL NOTE TN-18

By Sugan Subramanian

INTRODUCTION

This is a tech note on how to embed assembly instructions inside C source code. It is targeted towards programmers who have some knowledge of C-language and R3000 assembly language.

In IDT/C™ 5.0, assembly instructions can be inlined inside any genuine block of C-code. A genuine block of C-code is a section of C-code enclosed by open and closed curly braces. The inlined assembly may include synthetic assembly instructions. These instructions are expanded during compile/assembly phase of the compiler. The format agreed by the IDT/C 5.0 compiler depends on whether or not the inlined assembly lines require arguments, and whether these arguments are read, written, or both.

Specifically there are 4 cases to consider:

- inline without any parameters
- inline with read only parameters
- inline with write only parameters
- inline with read and write parameters

These four cases will be discussed elaborately in the following sections.

INLINE ASSEMBLY LINES WITHOUT ANY PARAMETERS

Format:

asm("<asm instrct1> ; <asm istrct2> ; <asm instrct2>; ... <asm instrctn>")

e.g:

```
unsigned int get_addr()
{
asm("li $2,0x80020000 ; lui $3, 0");
}
```

Description:

1. "get_addr" is a function that takes no arguments and returns an unsigned integer.
2. The inlined portion of the function body computes the return value == (0x80020000) that is saved at $2 (or) v0 and initializes $3 (or) v1 with zero.

Constraints:

All assembly instructions including synthetic instructions are allowed.

All register names should have hardware mnemonics.

i.e.:

General registers are $0, $1, .. , $31

Coprocessor 0 registers (has TLB, configuration specific registers) are $0,$1, ... , $31

Coprocessor 1 registers (has Floating Point Accelerator specific registers) are $0, $1, ... , $31

Coprocessor 2 registers are $0, $1, ... , $31

Coprocessor 3 registers are $0, $1, ... , $31

INLINE ASSEMBLY LINES WHICH USE WRITE-ONLY PARAMETERS

Format:

asm("<asm instrct1> ; <asm istrct2> ; <asm instrct2>; ... <asm instrctn>"
: "=<write-only_param1 format>" (<write-only_param1 name>),
"=<write-only_param2 format>" (<write-only_param2 name>),
... "=<write-only_paramk format>" (<write-only_paramk name>));

e.g:

```
int i,j;
    void main()
    {
    Intialize_Globals();
    printf("value of i = %d and j = %d\n",i,j);
    }
    void Initialize_Globals()
    {
    asm("ori %0,$0,3 ; ori %1, $0, 4"
    : "=r" (i), "=r" (j));
    }
```

Description:

1. "Initialize_Globals" is a function that takes no arguments and returns nothing.
2. The inlined portion of the function body initializes the global variables "i" and "j". Uses "i" and "j" as write-only parameters. Parameter "i" is referenced by %0 and "j" is referenced by %1. "=r" is the format for both "i" and "j". "=r" specifies that the following write-only parameter has a general register associated with it.

Constraints:

All assembly instructions including synthetic instructions are allowed.

All register names should have hardware mnemonics.

In R3000, the following are the possible hardware mnemonic:

General registers are $0, $1, .. , $31

(has TLB, configuration specific registers)

Coprocessor 0 registers are $0, $1, ... , $31
(has Floating Point Accelerator specific registers)
Coprocessor 1 registers are $0, $1, ... , $31
Coprocessor 2 registers are $0, $1, ... , $31
Coprocessor 3 registers are $0, $1, ... , $31

Write-only parameters can be either global or local variables. Write-only parameters are indexed from 0 to n-1, where n is the number of parameters used in the inlined code. Inside the inlined code, write_only_parameter1 is accessed by %0, write_only_parameter2 is accessed by %1, and so on. These are the formats that are allowed for write-only parameters:

"=r" ___ Specifies that the write-only parameter has a general register assigned to it.

"=f" ___ Specifies that the write-only parameter has a floating point register assigned to it.

INLINE ASSEMBLY LINES WHICH USES READ-ONLY PARAMETERS

Format: asm("<asm instrct1> ; <asm istrct2> ; <asm instrct2>; ... <asm instrctn>"
:: "<read-only_param1 format>" (<read-only_param1 name>),
"<read-only_param2 format>" (<read-only_param2 name>),
..."<read-only_paramk format>" (<read-only_paramk name>));

e.g:

```
void main()
{
print("INLINE VAL = %d\n", return_3());
}

int return_3()
{
asm("ori $2,$0,%0 ; ori $3, $0, %1"
:: "n" (3), "n" (4));
}
```

Description:

1."return_3" is a function that takes no arguments and returns integer value 3.
2. The inlined portion of the function computes the return value.
3. Whenever we use read only parameters without write only parameters, we have to use two colons "::" preceding them to specify that there are no write only parameters.

Constraints:

All assembly instructions including synthetic instructions are allowed.
All register names should have hardware mnemonics.
i.e.
General registers are $0, $1, .. , $31
Coprocessor 0 registers (has TLB, configuration specific registers) are $0, $1, ... , $31
Coprocessor 1 registers (has Floating Point Accelerator specific registers) are $0, $1, ... , $31
Coprocessor 2 registers are $0, $1, ... , $31
Coprocessor 3 registers are $0, $1, ... , $31

Read-only parameters are indexed from 0 to n-1, where n is the number of parameters used in the inlined code. Inside the inlined code, Read-only_parameter1 is accessed by %0, Read-only_parameter2 is accessed by %1, and so on. These are the formats that are allowed for read-only parameters:

"r" ___ Specifies that the parameter has a general register assigned to it.

"f" ___ Specifies that the parameter has a floating point register assigned to it.

"n" ___ Specifies that the parameter is an immediate value.

"m"___ Specifies that the parameter is a memory address.

"o" ___ Specifies that the parameter is an offsettable memory address.

"X" ___ Specifies that the parameter can be any of the above.

INLINE ASSEMBLY LINES THAT USES WRITE-ONLY AND READ-ONLY PARAMETERS

Format:

asm("<asm instrct1> ; <asm istrct2> ; <asm instrct2>; ...
<asm instrctn>"
: "=<output_var1 format>" (<output_var1 name>),
"=<output_var2 format>" (<output_var2 name>),
... "=<output_vark format>" (<output_vark name>)
: "<input_var1 format>" (<input_var1 name>),
"<input_var2 format>" (<input_var2 name>),
..."<input_vark format>" (<input_vark name>));

e.g:

```
#define ARRAY_SIZE_IN_BYTES 40
int b[20];
void main()
{
int a[10];
int i,j,k;
{asm (
"
.set    noreorder
  li      $11,%2;
  addiu   %0,%3;
1:;
  sw      $11,0(%0);
  addiu  $11,-4;
  bnez    $11,1b;
  addiu   %0,-4;
  li      %1,%4;
.set    reorder"
```

```
:: "r" (a), "r" (j), "n"
(ARRAY_SIZE_IN_BYTES),
   "n" (10*4), "m" (b)
:  "$11");}

printf ("return val = %d\n",j);
i=-1;

while (++i < 10)
   printf("a[%d] = %d\n",i,a[i]);
}
```

Description:

1. This program initializes an integer array of 10 with values starting from 0 through 36 by an increment of 4 and displays the array.
2. The inlined portion not only initializes the array but demonstrates one peculiar inline feature, how to use read-write parameter.
3. We are allowed to use registers inside inlined assembly lines as long as we declare that they will be clobbered. This is done by giving register name(s) preceded by three colons (":::") if there are no write-only and read-only parameters, a colon (":") following the read only parameter(s) if there are read-only parameters, and two colons ("::") following the write only parameter(s) if there are only write-only parameters.

Constraints:

All assembly instructions including synthetic instructions are allowed.

All register names should have hardware mnemonics. i.e.

General registers are $0, $1, .. , $31
Coprocessor 0 registers (has TLB, configuration specific registers) are $0, $1, ... , $31
Coprocessor 1 registers (has Floating Point Accelerator specific registers) are $0, $1, ... , $31
Coprocessor 2 registers are $0, $1, ... , $31
Coprocessor 3 registers are $0, $1, ... , $31

Whenever a parameter is used for reading and writing, declare such parameters to be either read-only or write-only and not both. This convention eliminates a lot of confusion. In the previous example, parameter "j" and "a" are declared to be read-only and used for both reading and writing. It is appropriate because both "j" and "a" are of type "r" (have general registers associated with them). Only read-only parameters that have registers associated with them are writable.

GENERAL RULES WHILE INLINING ASSEMBLY LINES

Always enclose your inlined assembly lines by a block of ".set noreorder" and ".set reorder" directives so that compiler leaves the inlined assembly lines untouched even if the entire code is optimized. However, some harmless warning messages are generated by the assembler (IDT/C 5.0) when the synthetic assembly instructions are expanded; they can simply be ignored.

Declare all your variables that are read from and the immediate values that are used inside inlined assembly to be read-only parameters. Declare all variables that are written to as write-only parameters. Whenever a temporary register is used inside inlined assembly code always make sure it gets declared as clobbered.

SUMMARY

Inlining assembly lines inside of c-code is a boon in itself if the only way of optimizing your c-code is through having different sections of it in assembly. In IDT/C 5.0, inlining assembly lines is complemented by the ability to use local and global variable names as aliases to the registers assigned to them.

Integrated Device Technology, Inc.

IDT/C™ BINARY UTILITIES

TECHNICAL NOTE TN-19

By Evelyn Zhan

INTRODUCTION

This note briefly explains the most important binary utilities of IDT/C™ 5.0. This technical note lists all valid switches in each utility, and is intended to be useful as a quick reference card.

gar: Create, modify and extract from archives.

gar -rc archive *member1 member2 ...*
Create an archive library whose contents are member1 member2 ... that can be linked with different application programs.

gar -t archive
List contents of an archive.

gar -x archive *member1 member2...*
Extract member1, member2... from an archive. If no member is specified, then all files in the archive are extracted.

gar -d archive *member1 member2...*
Delete member1, member2... from the archive. If no member is specified, the archive is untouched.

gar -r archive *member1 member2...*
Insert member1, member2... into archive with replacement of the original members.

gar -q archive *member1 member2...*
Quick append the member1, member2... to the end of the archive without checking for replacement.

gnm: Generate symbol table for the object file.

gnm *objectfile > outfile.nm*
This lists the symbol table for objectfile, sorted by symbol name, into file outfile.nm.

gnm -n *objectfile > outfile.nm*
This lists the symbol table for objectfile, sorted by symbol address, into the file outfile.nm.

Note: *objectfile* above can be any of the following: ecoff-file, object-file, or an archive of ecoff-file.

objcopy: Used to convert ecoff files to S-record.

objcopy -O srec objectfile outfile.srec
This converts the ecoff objectfile to Motorola S3 record format, for downloading to the evaluation boards or PROM programmers.

objcopy -O srec -b num objectfile outfile.srec
-b num option is always used with "-O srec" option. It is useful when programming EPROMS for boards which require bytewide EPROMS.
-b 0 creates S-record files corresponding to the 0th byte slice of 4-byte word.
-b 1 creates S-record files corresponding to the 1st byte slice of 4-byte word.
-b 2 creates S-record files corresponding to the 2nd byte slice of 4-byte word.
-b 3 creates S-record files corresponding to the 3rd byte slice of 4-byte word.

objcopy -O srec -b num -i bytenum objectfile outfile.srec
-i option is only applicable when creating S-records (with the "-O srec" option), and must be used in conjunction with the -b option. It is useful for programming EPROMS for boards that require interleaved EPROMS.
-i 1 interleave one byte.
-i 2 interleave two bytes. etc.

objcopy -O srec -p -b num objectfile outfile.srec
-p option is used to create prommable S-records. It should be used with -b to create bytewide PROMS. It orders the sequence of sections to be .text, .data and .bss, and sets the address fields of the S-records created to begin from 0x00000000.

Note:objcopy only supports S-records now.

objdump: Display information about ecoff files.

objdump -h objectfile > outfile
Display summary information from the section headers of the objectfile, such as
.text, .rdata, .data, .sdata, .sbss and .bss.

objdump -d objectfile > outfile
Display the assembler mnemonics for the machine instructions from objectfile.

objdump -t objectfile > outfile
Print the symbol table entries from objectfile.

ranlib: Generate an index to the contents of an archive and stores it in the archive.

ranlib archive

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

gsize: Create a table of starting address and size for various sections of the code (.text, .data, .bss).

gsize [-d | -o | -x | radix=number] objectfile... > outfile

Lists the section sizes, and the total size for each of the objectfile or archive in its argument list into outfile. The size of each section is given in decimal ('-d', or 'radix=10'); octal ('-o', or 'radix=8'); or hexadecimal ('-x', or 'radix=16').

Integrated Device Technology, Inc.

IDT/SIM™ 5.1 SOURCE CODE

TECHNICAL NOTE TN-20

By Upendra Kulkarni

This Technical Note offers a quick overview of the source code environment of IDT/SIM™ (System Integration Manager) (version 5.1).

IDT offers a number of RISC evaluation boards each with a variety of unique features. Consequently, the IDT/SIM on each board has some features which are uniquely tailored for that specific board and some features which are common to all boards. The source code for IDT/SIM for all boards is maintained in a single directory tree structure.

Source code for IDT/SIM (version 5.1) is expected to be used by individuals who have designed boards using a member of IDT's RISController™ family and are in the process of modifying IDT/SIM to achieve compatibility with their boards. The capabilities of IDT/SIM are described in its data sheet and user's manual.

A good number of source files are common to all SIMs; there is absolutely nothing specific to a particular board in these files. There are other files which are common but have parts of code in them which are unique to specific boards - a feature implemented using "#if defined()" or "#ifdef" conditional compilation directives. There is a third variety of files which bear the same name but exist in different directories; this indicates that the files contain code which performs similar tasks for different target boards, but the implementations are so different that conditional compiling would lead to confusion instead of ease of understanding. Finally there are files which are entirely specific only to one particular board. These files have no conditional compile statements, no equivalents in any other subdirectory, and are called for compilation and linking only for one specific SIM for one specific board.

Evaluation boards currently supported are 79S385™, 79RS381™, 79S341™, and 79S460™. Specific "Makefiles" for each board are provided.

From the top-most level of directories, there are 3 main directories - COMMON, SIM3000, SIM4000.

COMMON directory has two subdirectories:

- header - contains common header (#include) files used by all SIMs for all evaluation boards.
- c_asm - contains "C" and "assembler" files which are common to all versions of SIM for all evaluation boards. Most of these files use conditional compiling for different boards.

SIM3000 directory contains source code specific to boards designed with R3000 derivatives in mind. Currently, these boards include 79S385, 79S381, and 79S341. There are a number of subdirectories containing "Makefile"s specific for each evaluation board and possibly different tool-chains. The directory names are suggestive of which tool-chain or which evaluation board the Makefile in that directory supports.

For example, a directory name "_RS385C50" suggests that there is a Makefile in this directory which will create a SIM for the 79S385 board and will use IDT/C™ 5.1 tool-chain for compiling, etc. Directory names are appropriately abbreviated for DOS. In addition to the directories for Makefiles, there is also a "header" directory containing header (#include) files related to R3000-derivative based boards.

After making changes to the source code, the user needs to go into the directory appropriate for the intended target board and tool-chain, and simply run "make" (or "gmake" in case of DOS). All of the object files, and s-record files are built in the same chosen directory. The name(s) of the final product file(s) can be obtained by studying the Makefile(s). Typically, for a board using four ROMs (79S385, 79S381) the file names of the final s-record files are idtmonb0, idtmonb1, idtmonb2, and idtmonb3. For the 79S341 board, the final s-records are in file "idtmon.prm". Each Makefile also creates a version of code which can be run out of RAM on the target board. (the RAM-version). The RAM-version allows the user to debug or test modifications to SIM without actually having to program a new set of ROMs every time a change is made to the source code. The board may contain older version of SIM in its ROM, and the user downloads the newly created RAM-version into the RAM using the "load" command as if the RAM-version were a user application program. Issuing a "go" after the download is completed invokes the new RAM-version SIM.

The SIM4000 directory is similar to the SIM3000 directory; the only difference is that the code pertains to R4000 derivatives. Currently the 79S460 board is supported. The Makefile for this board can be found in the directory "IDTELF64".

Following is a list of global symbols which are used extensively in the source files to achieve conditional compiling for a specific CPU or a specific evaluation board. Please review these symbols in the context of the files you are likely to modify. Conforming to these conditional compiling rules is critical to a successful port of the SIM code to a new board design. Additional symbols can be defined in the Makefiles with "-D" switches and can be used to uniquely identify and support specific features of specific boards in future. Although new global symbols can also be defined in the source files, it is highly recommended that they be defined in the Makefiles to facilitate easy access to software developers other than the creator of the symbols.

CPU_R4000: to identify code specific to R4000 and its derivatives.

CPU_R3000: to identify code specific to R3000 and its derivatives.

R381: to identify code specific to 79RS381 board.
RS341: to identify code specific to 79S341 board.
P4000: to identify code specific to 79S460 board.
INET: to indicate code to be executed only if ethernet support is available on the target board.
PROM: to indicate that networking code is running out of PROM.
IDTSIM: to indicate modifications to industry standard ethernet drivers for IDT-SIM compatibility.
KERNEL: related to ethernet drivers.
XDS: IDT/C 4.1.1 compatibility-specific code modifications. Obsolete.

Integrated Device Technology, Inc.

DEVICE DRIVERS CONTAINED IN IDT/SIM™ FOR IDT ORION™ and RISCONTROLLER™ FAMILIES

TECHNICAL NOTE TN-24

INTRODUCTION

This technical note describes the various device drivers currently available from IDT in IDT/sim and IDT/kit™.

Of course, these software tools are constantly being enhanced, and additional drivers implemented. For current information, IDT recommends you work with your local sales representative.

In addition, many third-party companies provide additional software support, including real-time operating systems, network protocol support, and device drivers. Information on these products is available through the Advantage-IDT program.

Device Drivers Listing

All drivers listed below can currently be found in IDT/sim 5.1 source code. Overall, support for 9 different devices is currently available.

In addition to shortening development time for systems using the specific devices listed here, these functions can also be used as templates for systems requiring identical functions, but using different peripheral devices to implement them.

Finally, note that this listing does NOT include a listing of devices for which initialization-only functions are available in IDT/sim. Examples of these include start-up routines for the CPUs themselves, initialization of external DRAM controller devices (such as the one found on the '381 board), and memory sizing routines. While these functions tend to be system-specific, firmware engineers can use the source code provided with IDT/sim as a template for these functions.

8251:

Serial I/O device driver.
Source code in: SIM3000/drivers/drv_8251

8254:

Programmable interval timer driver: Contains code to install the driver (call to install_new_dev() - SIM function), as well as the driver itself. Since this device does not trasnsfer any data, there are no read/write functions. As such it is not the most representative driver - However it is an i-o device present on the original R3000A evaluation board, the 7RS382.
Source code in: SIM3000/drivers/drv_8254

8530:

SCC Driver. (Serial Communications Controller). This driver implements the standard asynchronous UART functions contained in the 8530/85C30.
Source code in: SIM4000/drivers/drv_8530 and SIM3000/drivers/drv_8530

Centronics Driver.

Parallel port driver for old IBM/PC centronics interface. The driver works with the hardware implementation of Centronics found in the 79S385A evaluation system, which uses a parallel register/FIFO structure to receive data.
Source code in: SIM3000/drivers/drv_centron

SCSI Driver.

Source code in: SIM4000/drivers/scsi & SIM3000/drivers/scsi

PC Backplane ISA I/O 16-bit Driver.

This driver was implemented to support the use of the '341 board in a PC/AT. There is a support program called pcio15.exe, which allows the PC/AT to act as a terminal for the '341 board.
Source code can be found in: COMMON/c_asm: pcio16asm.S pcio16asm.s pcio16drv.c

68681/2681:

DUART driver. A function called timer_start() sets tty1 (for all R3000 based boards except '381) to 9600 baud. A function called timer_stop() calculates elapsed time based on baud rate. It changes the baud rate of the unit in io->icb_di->dev_unit. This also has timer_start and timer_stop that are subsets of those in the c_asm dir.
Source code can be found in: COMMON/c_asm/s68681cons.c
Code can also be found in SIM3000/drivers/drv_68681

SONIC:

Ethernet Controller. Implements UDP protocol. Source code for this function is found in SIM4000/net/netinet/udp*. Ethernet address resolution protocol code is found in SIM4000/net/netinet/if_ether.c
The driver supports the "ping" command using the "ICMP" protocol; source code for this is found in SIM4000/net/netinet/*icmp*
TFTP routines in SIM4000/net/cmdsi/tftplib.c
Source code can be found in SIM4000/net/drivers
Support functions are in SIM4000/net/net

uPD72001 (NEC):

Serial (DUART) I/O controller. This Duart is contained in the 79S460 evaluation system for the R4600.
Source code is found in SIM4000/mpsccons.c

Integrated
Device Technology, Inc.

Integrated Device Technology, Inc.
2975 Stender Way
P.O. Box 58015
Santa Clara, CA 95054-3090
Tel: (408) 727-6116
FAX: (408) 492-8674

ELECTRONIC ACCESS

Internet: www.idt.com
E-Mail: info@idt.com
FAX-On-Demand: 800-9-IDT-FAX (in U.S.)
408-492-8391 (outside U.S.)

MAN-RISC-10075