AT&T
Microelectronics

ATT92010 *Hobbit*™ Microprocessor Programmer's Reference Manual

January 1993

Copyright

Disclaimers

Trademarks

Printed in the United States of America

Contents

Chapter 1 **Functional Description**

1.1 List of Features 1-1

1.2 Data Types 1-3

1.3 Addressing and Alignment Restrictions 1-3

1.3.1 Big-endian Byte Ordering 1-4
1.3.2 Little-endian Byte Ordering 1-4
1.3.3 Alignment 1-4

1.4 Stack Cache 1-5

1.4.1 Organization 1-5
1.4.2 Maintenance 1-6
1.4.3 Integer Accumulator 1-7
1.4.4 Precautions 1-8

1.5 Control Registers 1-8

1.5.1 Configuration Register 1-9
1.5.2 Fault Register 1-10
1.5.3 JTAG ID Register 1-11
1.5.4 Interrupt Stack Pointer 1-12
1.5.5 Maximum Stack Pointer 1-12
1.5.6 Program Counter 1-13
1.5.7 Program Status Word 1-14
1.5.8 Shadow Register 1-16
1.5.9 Stack Pointer 1-17
1.5.10 Segment Table Base 1-18
1.5.11 Time One 1-19
1.5.12 Timer Two 1-19
1.5.13 Victory Base 1-20

1.6 Instruction Format 1-21

1.6.1 One-Parcel Format 1-21
1.6.2 Three-Parcel Format 1-23
1.6.3 Five-Parcel Format 1-24

1.7 Operand Addressing Modes 1-27

1.8 Integer Arithmetic Operation 1-28

1.8.1 Carry Bit 1-29
1.8.2 Overflow Bit 1-29
1.8.3 Division and Remainders 1-30
1.8.4 Tagged Integer Arithmetic 1-30

1.9 Fast Calling Sequence 1-30

1.10 Prefetching Strategy 1-32

1.10.1 Branch Prediction and Branch Folding 1-33
1.10.2 Conditional Branches 1-33
1.10.3 Tracing 1-33

1.11 Event Processing 1-34

1.11.1 Reset 1-35
1.11.2 Interrupt 1-36
1.11.3 Exceptions 1-38
1.11.4 Unimplemented Instruction 1-39
1.11.5 Event Processing Priority 1-40

Chapter 2 Memory Management

2.1 Address Translation 2-1

2.2 Address Mapping 2-3

2.2.1 Paged Segment Addresses 2-4
2.2.2 Nonpaged Segment Addresses 2-4

2.3 Segment Tables 2-4

2.3.1 Paged Segment Table Entries 2-5
2.3.2 Nonpaged Segment Table Entries 2-5
2.3.3 Mixed Paged and Nonpaged Segment Tables 2-6

2.4 Page Tables 2-6

2.5 Memory Management Operations 2-8

2.6 MMU Performance 2-9

Chapter 3 Instruction Set

3.1 Format 3-1

3.2 Pipeline Considerations 3-3

3.3 Descriptions 3-4

Chapter 4 Performance

4.1 Execution Time 4-1

4.2 Delays 4-5

4.3 Branch Folding 4-7

Appendix A Bus Arbitration and Electrical Characteristics

A.1 Bus Protocol A-1

A.2 Surrendering the Bus A-2

A.3 Bus Transaction Types A-4

A.3.1 Read Transactions A-4
A.3.2 Write Transactions A-4
A.3.3 Interlocked Bus Transfer A-6
A.3.4 Block Data Transfer A-6

A.4 Exception Handling A-6

A.4.1 Bus Retry A-6
A.4.2 Bus Error A-6
A.4.3 Rest A-8

Appendix B **Electrical Characteristics**

B.1 Absolute Maximum Ratings B-1

B.2 Handling Precautions B-1

Appendix C **Testability**

C.1 Test Access Port C-1

C.2 TAP Controller C-2

C.2.1 Instruction Register C-5
C.2.2 By-Pass Register C-5
C.2.3 Boundary-Scan Register C-6
C.2.4 Identification Register C-7

Glossary

Index

Preface

The ATT92010 *Hobbit*™ Microprocessor is the culmination of years of research on computer architecture and software design at AT&T Bell Laboratories. The ATT92010 *Hobbit* Microprocessor is a second generation implementation of the CRISP architecture. This architecture combines salient features of the RISC and CISC design philosophy to simultaneously optimize high performance and high code density.

This Programmer's Reference Manual is intended for the experienced design engineer. The material presented in this manual assumes familiarity with microprocessors and is organized into the following section.

- Chapter 1 Functional Description—A detailed discussion of the ATT92010 *Hobbit* Microprocessor and its features.
- Chapter 2 Memory Management—An overview of the ATT92010 *Hobbit* Microprocessor memory management unit.
- Chapter 3 Instruction Set—A detailed description of each instruction arranged alphabetically. For quick reference, the instruction name appears at the top of each page in this chapter.

  Chapter 4 Performance—This chapter describes discusses performance data based on detailed parameters.
- Appendix A, B and C—These appendixes present hardware information as a point of reference.

Chapter 1 Functional Description

The ATT92010 *Hobbit*™ Microprocessor is a high-performance 32-bit central processing unit. Derived from AT&T Bell Laboratories' CRISP (C-Language Reduced Instruction Set Processor) architecture, the microprocessor combines the best of RISC (Reduced Instruction Set Computing) devices, such as high performance, with the best of CISC (Complex Instruction Set Computing) devices, such as high code density.

1.1 List of Features

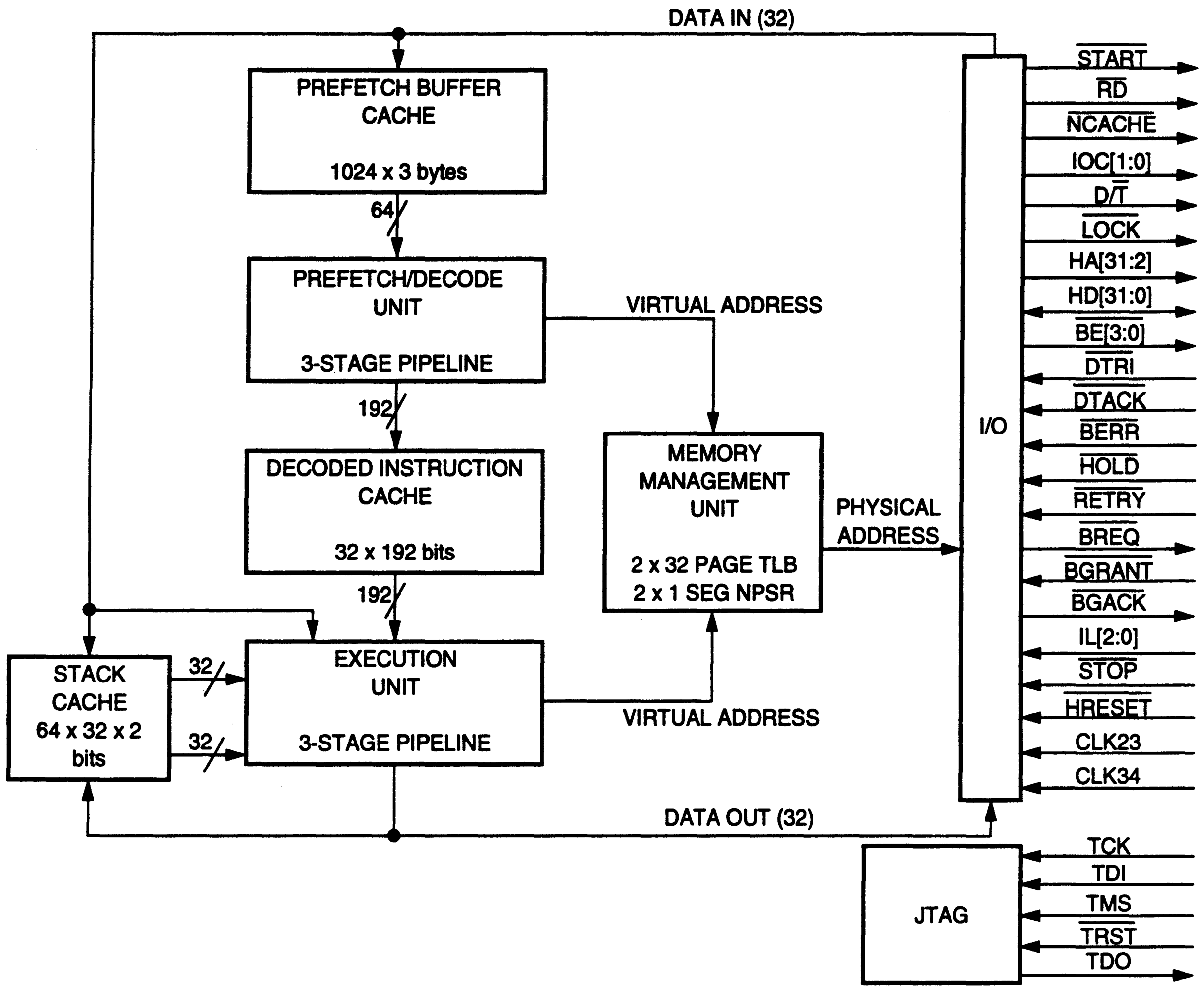
Major implementation features of the ATT92010 include:

- High Performance
  —Single-cycle instruction execution (for most instructions)
  —Operand bypass mechanism
  —Branch prediction and Branch folding
- On-Chip Integrated Resources
  —3 Kbyte encoded instruction cache (organized as 3-way set associative)
  —256 byte stack cache that holds top of user stack
  —32-entry, direct-mapped, decoded instruction cache
  —Memory Management Unit (MMU) with dual 32-entry translation look-aside buffers (TLB) for text and data address translation
- Big-endian / Little-endian Data Byte Ordering
- Low Power Consumption
  —250 mW at 3.3V, 20 MHz
  —900 mW at 5.0V, 30 MHz
  — <50μA in standby mode
- High Code Density
  —Rationalized Instruction set
  —Variable length instruction format
  —Memory-to-Memory architecture

- Low I/O Traffic
  —Integrated caches and Operand bypass
- Simple Programming Model
  —No programmer-visible registers

The ATT92010 *Hobbit* Microprocessor block diagram is shown in Figure 1-1.

Figure 1-1 ATT92010 Block Diagram



1.2 Data Types

Six integer data types are supported:

- Signed and unsigned bytes (8-bit)
- Signed and unsigned half-words (16-bit)
- Signed and unsigned words (32-bit)

Non-word operands are aligned and then expanded to 32-bit through sign extension (if signed) or clearing high-order bits (if unsigned).

The 32-bit ALU performs the requested function after alignment and expansion. Carry and overflow are determined relative to the 32-bit result.

For destinations less than 32-bit, the least significant bits of the 32-bit ALU result are selected. Changing a value by truncation constitutes neither overflow nor carry.

True three-operand (triadic) instructions are not provided. However, instruction encoding that provide two source operands and store the full 32-bit result in the accumulator are provided. This instruction is called a two-and-a-half-operand instruction.

For example, the mnemonic for an addition instruction is ADD3, while a two-operand (dyadic) addition is ADD. For this instruction, the two source operands are added and the full 32-bit result is stored in the accumulator.

1.3 Addressing and Alignment Restrictions

Numbering of bits within bytes is the same as with the Intel™ 80x86, Motorola™ 680x0 and the DEC™ VAX™. The numbering of bytes, within data words, is selectable for the User and Kernel modes. The User mode is set via the program status word (PSW) UL-bit (User Little endian-bit). The Kernel mode is set via the configuration register (CONFIG) KL-bit (Kernel Little endian-bit).

1.3.1 Big-endian Byte Ordering

When the PSW user little-endian bit and CONFIG kernel little-endian bit equals zero (0), the numbering of bytes within data words corresponds to that in the IBM 370 user mode and Motorola 680X0 kernel mode (see Figure 1-2).

Figure 1-2 Big-endian Byte Ordering

31 BYTE0 24	23 BYTE1 16	15 BYTE2 8	7 BYTE3 0

Note Text is always in big-endian order.

1.3.2 Little-endian Byte Ordering

When the PSW user little-endian bit and CONFIG kernel little-endian bit equals one (1), the numbering of bytes within data words corresponds to that in the VAX user mode and Intel 80X86 kernel mode (see Figure 1-3).

Figure 1-3 Little-endian Byte Ordering

31 BYTE3 24	23 BYTE2 16	15 BYTE1 8	7 BYTE0 0

1.3.3 Alignment

The ATT92010 *Hobbit* Microprocessor fetches words only; bytes and half-words are accessed by extracting them from the surrounding word. Likewise, all stores are done to word-addresses, with the appropriate write strobes enabled. However during reads, the byte-strobes indicate which bytes, within the word being fetched, will ultimately be extracted by the instruction.

All operand addresses should be naturally aligned for the operand type. If an operand fetch (or operand store) is to an address that is not properly aligned for the data type, an alignment exception is signaled. Instructions can only be fetched on half-word boundaries and should be suitably aligned even though no exception is signaled. Alignment occurs as the least significant bit of the address is ignored for text fetches.

1.4 Stack Cache

Registers are typically used to hold frequently referenced variables inside a CPU to reduce memory traffic and speed up operand accesses. The traditional stack holds local variables, incoming and outgoing arguments, compiler temporaries and registers being saved during procedure calls. Measurements have shown that these accesses to the stack are typically only a few tens of words concentrated around the top of the stack. The compiler attempts to move this data into registers whenever possible. The result is a substantial amount of memory traffic between a small number of general purpose registers and a few locations on the stack.

The ATT90210 *Hobbit* Microprocessor allocates data registers in a way that is radically different from traditional machines. Rather than have the compiler allocate registers and generate code to move data back and forth between registers and the stack, the microprocessor automatically maps the stack onto machine registers, called the Stack Cache. By tracking the top of the stack in high speed machine registers, useless traffic to and from the stack is avoided and a high degree of register allocation is achieved.

Registers are allocated by the hardware, rather than by a software compiler and general purpose registers are eliminated.

1.4.1 Organization

The stack cache consists of a bank of 64 registers (4 bytes wide) organized as a circular buffer maintained by two 28-bit registers holding quad-word addresses. These registers are:

- The Maximum Stack Pointer (MSP) — contains the address above the highest address of the data that is currently kept in the stack cache registers
- The Stack Pointer (SP) — delimits the lowest address of data in the stack cache

Only a simple range-check is needed to determine if an address resides within the stack cache. If SP ≤ADDR <MSP, it falls within the stack cache. Even though the stack cache limits are maintained on quad-word boundaries, the stack cache is byte addressable and appears as normal memory. All virtual addresses, generated to access data, can freely reference the stack cache.

Since, the stack cache can contain the top 64 words of the stack, most automatic variables and incoming and outgoing arguments will be in the stack cache. The stack cache is, therefore, a major factor in efficient instruction execution.

1.4.2 Maintenance

Six instructions maintain the stack cache:

- CALL — moves the return address onto the top of the stack and branches to the target address
- CATCH — guarantees the stack cache is filled at least as deep as the number of the bytes specified in its operand and is used after a CALL instruction to ensure an optimal portion of the stack is on-chip
- CRET — used by the kernel to load a new SP and MSP and execute the CATCH instruction. CRET also loads a new program status word (PSW) and program counter address and is used for context switches
- ENTER — allocates space on the new stack frame by subtracting its operand, the size of the new stack frame, from the SP
- POPN — deallocates the current stack frame by adding its argument to the SP
- RETURN — deallocates the current stack frame by adding its argument to the SP, then branches to the return value previously placed on the stack

ENTER and CATCH are also used when the stack cache circular buffer is not large enough to accommodate the entire stack frame. When a new procedure is entered, the ENTER instruction attempts to allocate a new set of registers equal to the size of the new stack frame. If free register space exists in the circular buffer, then only the SP needs to be modified. If not, then the entries nearest the MSP are flushed back to main memory.

- If the new stack frame size is less than 256 bytes, only the stack frame size, minus the number of free entries, must be flushed.
- If the new stack frame size is greater than or equal to 256 bytes, all valid stack cache entries are flushed and only part of the new stack frame nearest the SP is kept in the stack cache.

When control has returned to the calling procedure, flushed entries may need to be restored to the Stack Cache, via the CATCH instruction. The CATCH instruction argument specifies the number of stack cache entries that must be valid before the flow of execution can resume. The argument is used as a stack offset and a virtual address is generated. If this calculated address resides within the stack cache, execution continues.

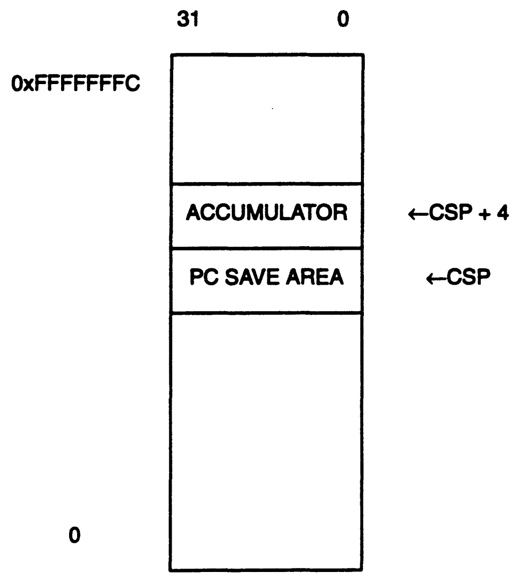
However, if the calculated address resides outside the address range of valid stack cache entries, quad-words pointed to by the MSP are restored to the stack cache from off-chip memory. Then the MSP is incremented until CATCH is satisfied or the stack cache is full.

1.4.3 Integer Accumulator

The integer accumulator is not an actual hardware register. It is the word in memory above the word addressed by the current stack pointer (CSP). The CSP is either the stack pointer (SP) or the interrupt stack pointer (ISP) as determined by the program status word (PSW).

The integer accumulator normally resides on-chip in the stack cache, but it may be off-chip if the SP = MSP or CSP = ISP.

Figure 1-4 Integer Accumulator



1.4.4 Precautions

If an address is generated in any processing stage (an indirect address calculations, for example) the stack cache is referenced if that address is greater than or equal to the SP and less than the MSP. This conceptual model is violated when executing with CSP = ISP. There are no problems with memory accesses as long as the stack cache, based at the SP, and the interrupt stack, based at the ISP, do not overlap. For similar reasons, the following addresses must not lie between the SP and MSP:

- The vector table, defined by the vector base (VB)
- The address translation tables used by the memory management unit (MMU)
- Any text address

1.5 Control Registers

The ATT92010 *Hobbit* Microprocessor control registers are shown in Table 1-1. Each register is describe in the sections that follow.

Table 1-1 Control Registers

Name	Description
CONFIG	Configuration Register
FAULT	Fault Register
ID	Identification Register
ISP	Interrupt Stack Pointer
MSP	Maximum Stack Pointer
PC	Program Counter
PSW	Program Status Word
SHAD	Shadow Register
SP	Stack Pointer
STB	Segment Table Base
TIMER1	Timer1 Register
TIMER2	Timer2 Register
VB	Vector Base

1.5.1 Configuration Register

The configuration register (CONFIG) is set to 0x0 upon reset.

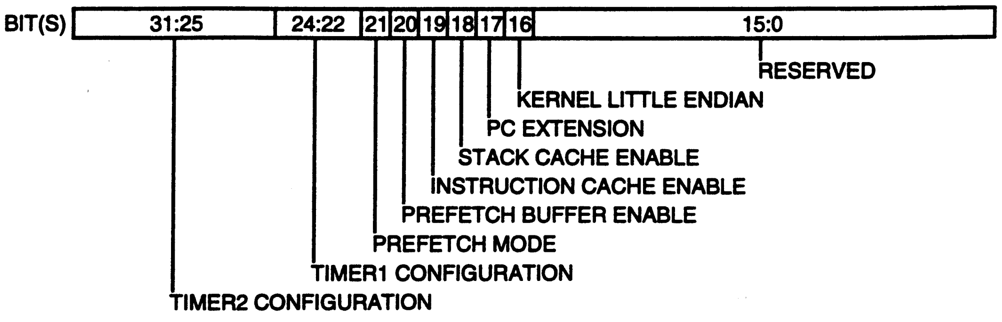


Table 1-2 Configuration Register (*Sheet 1 of 2*)

Bit(s)	Description
31:25	**Timer2 Configuration.** A 7-bit field that configures Timer2. • Bit 31. If 0, Timer2 does not generate an interrupt. If 1, Timer2 generates an interrupt using a Timer2 vector when an overflow occurs (goes from 0xFFFFFFFF to 0x0). This is a level one interrupt. An external level one interrupt and a Timer1 interrupt have priority over Timer2. • Bit 30. If 0, Timer2 is on all the time (with reference to bits 29:25). If 1, the timer only increments in kernel mode (PSW execution level bit is 0). • Bit 29:25 selects the internal event to increment Timer2. **Bit 29 / Bit 28 / Bit 27 / Bit 26 / Bit 25 / Event** 0 / 0 / 0 / 0 / 0 / Count clock cycles. 0 / 0 / 0 / 0 / 1 / Count completed instructions (folded branches are not counted). 1 / 1 / 1 / 1 / 1 / Do not increment the timer; a low power feature.
24:22	**Timer1 Configuration.** A 3-bit field that configures Timer1. • Bit 24. If 0, Timer1 does not generate an interrupt. If 1, Timer1 generates an interrupt using a Timer1 vector when an overflow occurs (goes from 0xFFFFFFFF to 0x0). This is a level one interrupt. An external level one interrupt has priority over Timer1. • Bit 23. If 0, Timer1 is on all the time (with reference to bit 22). If 1, the timer only increments in kernel mode (PSW execution level bit is 0). • Bit 22. If 0, Timer1 counts clock cycles. If 1, Timer1 counts completed instructions (folded branches are not counted).
21	**Prefetch Mode.** This bit controls prefetching of instructions. If 0, prefetching off-chip is not performed; predecoding from the prefetch buffer into the instruction cache is performed. If 1, aggressive prefetching is performed.

Table 1-2 Configuration Register (*Sheet 2 of 2*)

Bit(s)	Description
20	**Prefetch Buffer Enable.** A 0 disables the prefetch buffer from hitting; a 1 enables it. The prefetch buffer is neither flushed nor altered when this bit is modified.
19	**Instruction Cache Enable.** A 0 disables the instruction cache from accessing; a 1 enables it. The instruction cache is neither flushed nor altered when this bit is modified.
18	**Stack Cache Enable.** A 0 disables the stack cache from accessing; a 1 enables it. The stack cache is neither flushed nor altered when this bit is modified.
17	**PC Extension.** A 0 selects 0 extension of 16-bit absolute addresses; a 1 selects the extension of 16-bit absolute addresses where bits 31:29 are copied from bits 31:29 of the PC and bits 28:16 are set to 0.
16	**Kernel Little Endian.** A 0 selects data as big endian in kernel mode; a 1 selects data as little endian in kernel mode.
15:0	**Reserved.** They return 0 when read and should be written with 0 on CONFIG writes.

Special precautions must be taken when modifying the configuration register. The number of no-operation instructions (NOPs) that must follow the register write varies according to the bit(s) being modified and the number of wait-states being used by I/O transactions.

Modify CONFIG by following the CONFIG write with either a context return from kernel (CRET) instruction or kernel return (KRET) instruction.

1.5.2 Fault Register

This s reports the 32-bit operand aligned virtual address for the processing of exception IDs 0x8 and 0x9.

BIT(S)	31:0
	FAULT ADDRESS

Table 1-3 Fault Register

Bit(s)	Description
31:0	**Fault Address.** The address causing the current exception for use by the exception handler.

1.5.3 JTAG ID Register

This register is the JTAG device identification (ID) register. It is readable by serial shifting through the test access port (TAP) and through normal register access. This register is read only. If this register is written to, no operation is performed.

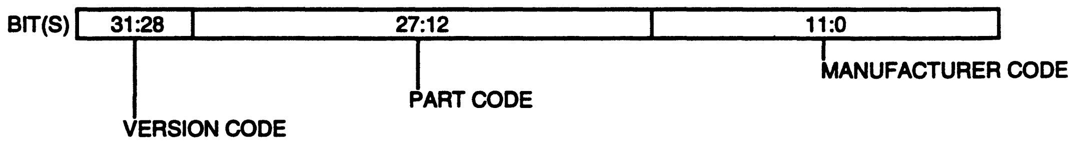


Table 1-4 Identification (ID) Register

Bit(s)	Description
31:28	**Version Code.** This field is 0x0 for mask one and 0x1 for mask two.
27:12	**Part Code.** This field is 0x0 for the ATT92010 *Hobbit* Microprocessor.
11:0	**Manufacturer Code.** This field is 0x3B for AT&T Microelectronics.

1.5.4 Interrupt Stack Pointer (ISP)

The interrupt stack pointer (ISP) is used to generate addresses whenever the program status word (PSW) current stack pointer bit is zero (0). For example, the address in stack offset modes, to locate the accumulator, and as the pointer manipulated by the instructions CALL, RETURN, POPN, and ENTER. The ISP is not associated with the stack cache.

The instructions CRET, KCALL, and KRET, and operating system sequences, interrupts, and exceptions use the ISP to maintain a stack of event blocks. The ISP must be valid at all times. A fault, on any ISP based address, during event processing will reset the ATT92010 *Hobbit* Microprocessor. Address translation is performed if the MMU is enabled by setting the PSW virtual/physical addressing mode bit to one (1).

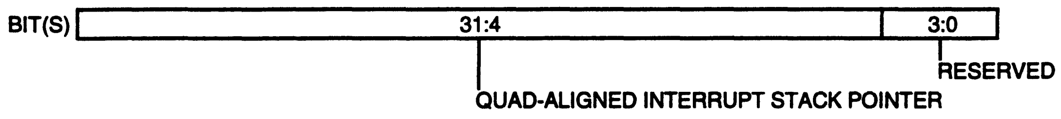


Table 1-5 Interrupt Stack Pointer (ISP)

Bit(s)	Description
31:4	**Quad-Aligned Interrupt Stack Pointer.** This is the address of the interrupt stack.
3:0	**Reserved.** These bits return 0 when read.

1.5.5 Maximum Stack Pointer (MSP)

The maximum stack pointer (MSP), in conjunction with the SP, is associated with the on-chip stack cache. If the stack cache is enabled and the current stack pointer is the SP, then any address greater than, or equal to, the SP and less than the MSP hits the stack cache.

Stack cache hits when SP < address < MSP

With a memory access that hits the stack cache, data is fetched or stored in the cache, not in external memory. The MSP must be greater than or equal to the SP and less than or equal to SP + 256 (stack cache size), or the result of stack cache accesses are dependent upon context and therefore are unpredictable.

When the SP is the direct destination of an instruction, through a CPU-prefixed instruction with the SP as the destination, the MSP is updated with the same value. This defines an empty stack cache (SP = MSP). The MSP is manipulated implicitly by CATCH, CRET, ENTER, POPN, and RETURN. Consequently, the MSP should only be modified by stack manipulation instructions. Address translation is performed if the MMU is enabled by setting the PSW virtual/physical addressing mode bit to one (virtual addressing is enabled).

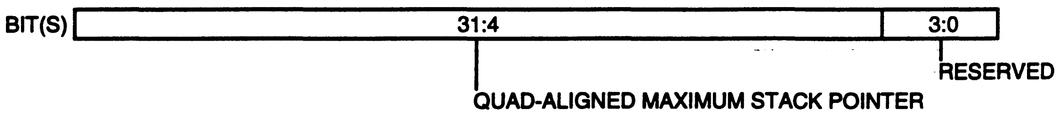


Table 1-6 Maximum Stack Pointer

Bit(s)	Description
31:4	**Quad-Aligned Maximum Stack Pointer.** This is the address above top of user stack.
3:0	**Reserved.** These bits return 0 when read.

1.5.6 Program Counter (PC)

The program counter (PC) addresses the instruction currently being executed. Instructions are aligned on parcel (half-word) boundaries. Since parcels are composed of 2 bytes, the PC is always a multiple of two and the low-order bit is always 0. The PC cannot be directly manipulated by a general instruction. It can only be read or modified by control-flow instructions CALL, CRET, JMP, KCALL, KRET, and RETURN and read by the move instruction LDRAA (Load Relative Address into Accumulator).

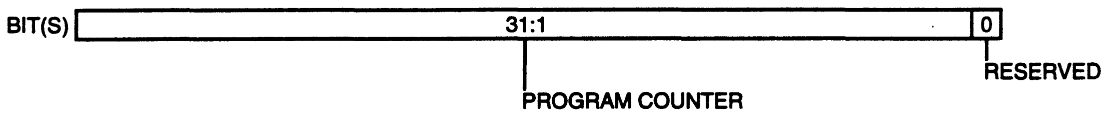


Table 1-7 Program Counter

Bit(s)	Description
31:1	**Program Counter.** This is the address of the current instruction.
0	**Reserved.**

1.5.7 Program Status Word (PSW)

The program status word (PSW) is set to 0x0 upon reset.

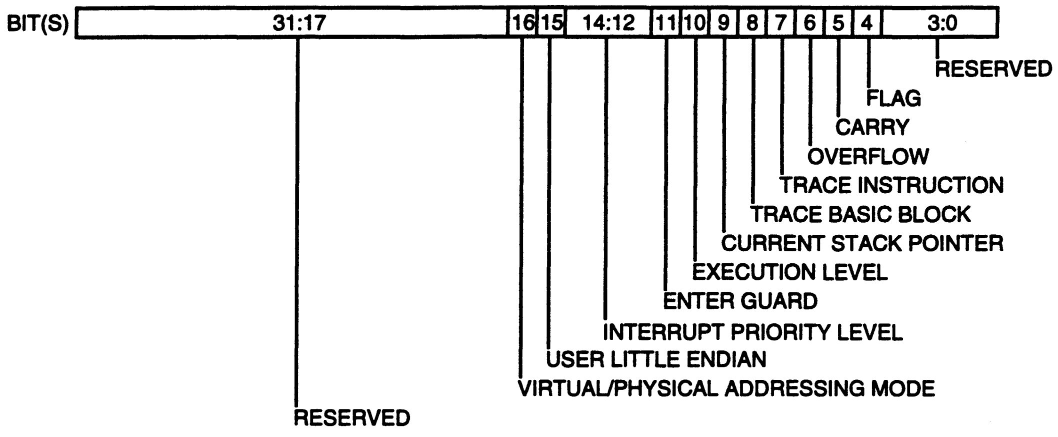


Table 1-8 Program Status Word (*Sheet 1 of 3*)

Bit(s)	Description
31:17	**Reserved.**
16	**Virtual/Physical Addressing Mode.** If 0, physical addressing (memory management disabled) is enabled, and $\overline{\text{NCACHE}}$ is asserted. If 1, virtual addressing is enabled (memory management enabled). Special precautions must be taken when explicitly modifying this bit. If it is explicitly modified, the section of code executing must be mapped physical address = virtual address. The safest means of manipulating this bit is through KRET.
15	**User Little-Endian.** If 0, data is selected as big-endian in user mode. If 1, data is selected as little-endian in user mode.

Table 1-8 Program Status Word (*Sheet 2 of 3*)

Bit(s)	Description
14:12	**Interrupt Priority Level.** Interrupts are accepted when the requesting device level (IL[2:0]) is less than interrupt priority level or equal to 0. When these bits equal 7, all interrupts are enabled.
11	**Enter Guard.** Set on an ENTER instruction that does not result in any stack cache flush. This bit is not cleared when the PSW is read.
10	**Execution Level.** If 0, execution at the kernel level is performed. If 1, execution at the user level is performed.
9	**Current Stack Pointer.** If 0, the ISP is used as the CSP for stack operations. If 1, the SP is used as the CSP for stack operations. If this bit is modified by a direct write to the PSW, thereby changing the CSP, it is necessary to update SHAD to the value of the new SP. This update is handled automatically by the CRET, KCALL, and KRET instructions. If this bit is set to 1, and it was previously 0, the instruction modifying the PSW should be followed by the instruction MOV %SP,%SHAD. If this bit is set to 0 when it was previously 1, the next instruction should be MOV %ISP,%SHAD. Due to interrupts and exceptions, it is recommended that this bit not be modified by a direct write to the PSW since the above operations cannot be guaranteed to be atomic.
8	**Trace Basic Block.** Controls basic block tracing. If 1, the ATT92010 *Hobbit* Microprocessor executes instructions until a CALL, RETURN, or any jump (folded or not) instruction, referred to as the N instruction, executes. The instruction following instruction N, referred to as N + 1, is not permitted in the execution unit, and a trace instruction is generated internally. This trace instruction blocks the pipeline and forces the ATT92010 *Hobbit* Microprocessor to take a trace exception using the PC of the N + 1 instruction as the exception PC. As branch folding is performed prior to the trace identifier, folded branches are not explicitly traceable. If both the trace instruction and the trace basic block bits are set to 1, the function is that of the trace instruction.
7	**Trace Instruction.** Controls instruction tracing. When 1, the ATT92010 *Hobbit* Microprocessor allows the next instruction, N, to execute normally. The instruction following instruction N, referred to as N + 1, is not permitted in the execution unit, and a trace instruction is generated on the fly. This trace instruction blocks the pipeline and forces the ATT92010 *Hobbit* Microprocessor to take a trace exception using the PC of the N + 1 instruction as the exception PC. As branch folding is performed prior to the trace identifier, folded branches are not explicitly traceable. If both the trace instruction and the trace basic block bits are set to 1, the function is that of the trace instruction.

Table 1-8 Program Status Word (*Sheet 3 of 3*)

Bit(s)	Description
6	**Overflow.** If 0, this bit indicates that an operation did not generate a signed overflow. If 1, this bit indicates that an operation generated a signed overflow. This bit is not cleared by a read of the PSW.
5	**Carry.** If 0, this bit indicates that an operation did not generate an unsigned overflow. If 1, this bit indicates that an operation generated an unsigned overflow. This bit is not cleared by a read of the PSW.
4	**Flag.** Set/cleared by CMP, TADD, TESTC, TESTV, and TSUB instructions. This bit is not cleared by a read of the PSW.
3:0	**Reserved.** These bits are reserved. They return 0 when read and must be written with 0 on PSW writes.

The exception and interrupt sequences alter only the lower 16-bits of the program status word (PSW). To remain restartable, the carry and overflow bits are not cleared on reading the PSW until the instruction completes. Reads of the PSW are not interlocked against flag setting. If an instruction sets the flag, carry, or overflow bits, there must be at least two intervening instructions, which do not use or modify these bits, before the PSW can be read accurately.

1.5.8 Shadow Register (SHAD)

The shadow register (SHAD) is a copy of the current stack pointer (CSP). It is maintained by the ATT92010 *Hobbit* Microprocessor's internal sequences to facilitate restarting of instructions. In the course of the CRET, ENTER, KCALL, KRET, and RETURN instructions, or any time the CSP is modified, SHAD is automatically updated to be consistent with the CSP.

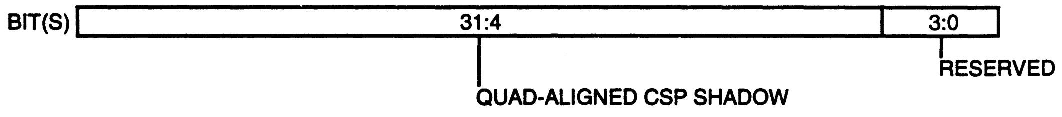


Table 1-9 Shadow Register

Bit(s)	Description
31:4	**Quad-Aligned CSP Shadow.** These bits contain a copy of the CSP.
3:0	**Reserved.** These bits return 0 when read.

If the program status word (PSW) current stack pointer (CSP) bit is modified by a direct write to the PSW, thereby changing the CSP, it is necessary to update SHAD to the value of the new SP. The instructions KCALL and KRET handle this automatically.

1.5.9 Stack Pointer (SP)

The stack pointer (SP) addresses the top of the stack. The stack grows downwards toward memory location zero (0). The SP is used to generate addresses (i.e., as the base address in offset modes, to locate the accumulator, and as the pointer manipulated by CALL, ENTER, POPN, and RETURN) whenever the PSW current stack pointer bit is one (1). Address translation is performed if the MMU is enabled by setting the PSW virtual/physical addressing mode bit to one (1).

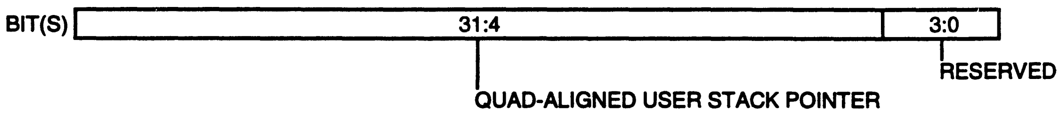


Table 1-10 Stack Pointer

Bit(s)	Description
31:4	**Quad-Aligned User Stack Pointer.** This is the user stack address.
3:0	**Reserved.** These bits return 0 when read.

1.5.10 Segment Table Base (STB)

When virtual addressing is turned on by the program status word (PSW) virtual/physical addressing mode bit, the segment table base (STB) contains a pointer to the start of the segment table used in address translation.

The base of the segment table is always page-size aligned, 4 Kbyte boundary. The STB is only used during miss processing, which in turn is used to fill entries in the on-chip translation look-aside buffer (TLB) or segment registers.

When the STB is written, the TLBs and segment registers of the memory management unit (MMU) are flushed, invalidating all entries. Neither the physically addressed prefetch buffer (PFB), the virtually addressed instruction cache (IC), nor the virtually addressed stack cache (SC) are flushed. Cache coherency is the responsibility of the user.

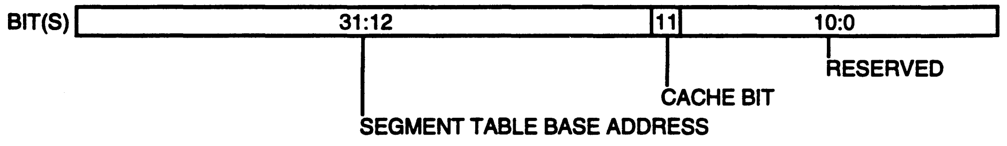


Table 1-11 Segment Table Base

Bit(s)	Description
31:12	**Segment Table Base Address.** This is the page-aligned base address of the segment table.
11	**Cache Bit.** A cacheable bit that is copied to the cacheable pin whenever a segment table access is made during misprocessing, indicating if segment table entries should be cached. If 1, $\overline{\text{NCACHE}}$ is deasserted and caching of segment table entries is allowed.
10:0	**Reserved.** Return 0 when read.

1.5.11 Timer One (TIMER1)

This register is a 32-bit internal register that is configured by the 3-bit field (24:22) of CONFIG to count various events.

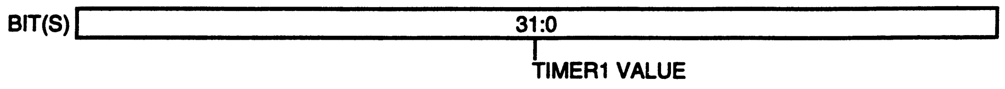


Table 1-12 Timer1

Bit(s)	Description
31:0	**Timer1 Value.** These bits contain the count value for Timer1.

1.5.12 Timer Two (TIMER2)

This register is a 32-bit internal register that is configured by the 7-bit field (31:25) of CONFIG to count various events.

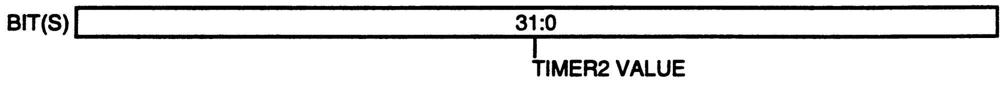


Table 1-13 Timer2

Bit(s)	Description
31:0	**Timer2 Value.** These bits contain the count value for Timer1.

1.5.13 Vector Base (VB)

The vector base (VB) is used as a table base that contains transfer addresses used by interrupts, exceptions and the KCALL instruction. Address translation is performed when the memory management unit (MMU) is enabled by setting the PSW virtual/physical bit to one (1). The Vector Table (see Table 1-14) should always be available. If access to the Vector Table is faulted, the ATT92010 *Hobbit* Microprocessor resets.

A memory fault causes an infinite loop until the interrupt stack is exhausted and the ATT92010 *Hobbit* Microprocessor resets. Consequently, an exception program counter (PC) should be present in memory. In addition, the niladic trap and unimplemented instruction handler must be in user memory space so that the handler can be accessed while in user mode.

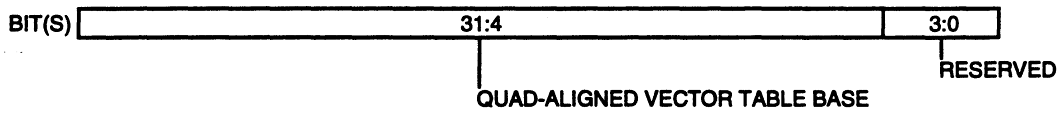


Table 1-14 Vector Base (VB)

Bit(s)	Description
31:4	**Quad-Aligned Vector Table Base.** The vector table (shown below) should always be available. VB + 52→ FP EXCEPTION VB + 48→ TIMER2 INTERRUPT VB + 44→ TIMER1 INTERRUPT VB + 40→ INTERRUPT 6 VB + 36→ INTERRUPT 5 VB + 32→ INTERRUPT 4 VB + 28→ INTERRUPT 3 VB + 24→ INTERRUPT 2 VB + 20→ INTERRUPT 1 VB + 16→ NONMASKABLE INTERRUPT VB + 12→ UNIMPLEMENTED INSTRUCTION VB + 8→ NILADIC TRAPS VB + 4→ EXCEPTION PC VB → KCALL PC
3:0	**Reserved.** These bits return 0 when read.

1.6 Instruction Format

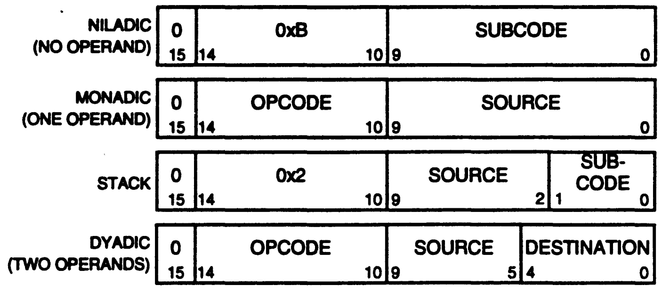
Instructions are composed of two-byte long parcels and are encoded in one-, three- and five-parcel lengths. A simple instruction is encoded in five-parcels, allowing for encoding of two complete 32-bit addresses in each instruction. In general, the one- and three-parcel instructions are more compact encoding of five-parcel instructions.

Instructions have a maximum of two operands that can be used for addressing modes. For the dyadic instructions, one source doubles as destination or the accumulator is selected to serve as an implicit destination. The instruction formats are

- One-parcel — for zero-, one- and two-operand instructions
- Three-parcel — for one- and two-operand instructions
- Five-parcel — for two-operand instructions

1.6.1 One-Parcel Format

Many of the most common zero-, one- and two-operand instruction types are encoded in one-parcel.



A zero (0) in the most significant bit distinguishes all one-parcel instruction formats. The *subcode* field distinguishes the niladic and stack instructions.

For operands, 5-bit immediate fields are sign extended and 5-bit stack offset fields are zero extended. All 10-bit fields are zero extended except for the CALL and JMP instruction which are sign extended. The 8-bit fields are zero extended, except for the ENTER instruction, which is one-filled. Tables 1-15, 1-16 and 1-17 detail the one-parcel instruction encoding.

Note that operand alignment restrictions allow some address offsets to be scaled; extending the effective addressing range. The scaling of certain immediate constants is possible by the specific operand value restrictions of the corresponding instructions. Five-bit offset values are multiplied by four before they are added to the stack pointer (SP). The 10-bit PC-relative offsets in the JMP and CALL instructions are multiplied by 2 before they are used; the other 10-bit values are multiplied by four before they are used.

Table 1-15 Monadics/Dyadics Encoding (One-Parcel)

opcode[4:3]	opcode[2:0]							
	000	001	010	011	100	101	110	111
00	KCALL	CALL	stack	JMP	JMPFN	JMPFY	JMPTN	JMPTY
01	unimp*	unimp*	MOV.WS	niladic	unimp*	ADD3.WS	AND3.CS	AND.SS
10	CMPEQ.CS	CMPGT.SS	CMPGT.CS	CMPEQ.SS	ADD.CS	ADD3.CS	ADD.SS	ADD3.SS
11	MOV.SS	MOV.IS	MOV.SI	MOV.II	MOV.CS	MOVA.SS	SHL3.CS	SHR3.CS

*The unimplemented instruction sequence is performed

C=5-bit immediate, I=5-bit indirect stack offset, S=5-bit word-aligned immediate

Table 1-16 Encoding Stack (One-Parcel)

subcode[1:0]			
00	01	10	11
ENTER	CATCH	RETURN	POPN

Table 1-17 Niladics Encoding (One-Parcel)

subcode[9:3]	subcode[2:0]							
	000	001	010	011	100	101	110	111
0000000	CPU	KRET	NOP	FLUSHI	FLUSHP	CRET	FLUSHD*	unimp*
0000001	TESTV	TESTC	CLRE	unimp*	unimp*	unimp*	unimp*	unimp*
000001x	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
00001xx	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
0001xxx	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
001xxxx	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
01xxxxx	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
1xxxxxx	trap†	trap†	trap†	trap†	trap†	trap†	trap†	trap†

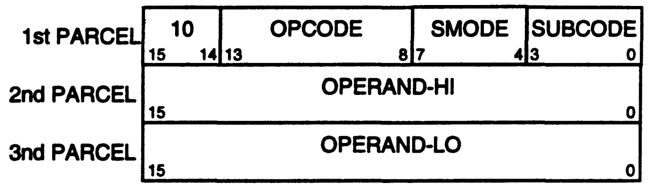
*The unimplemented instruction sequence is performed

†The niladic trap through VB + 8 is performed

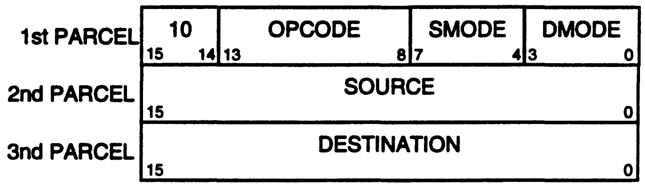
1.6.2 Three-Parcel Format

Three-parcel instructions are distinguished by a '10' in the two most significant bits. The subcode field distinguishes the different monadic instructions. The notation *operand-lo* represents the low-order 16-bits and *operand-hi* represents the high-order 16-bits. A similar convention applies to the source and destination operands of the five-parcel dyadic instructions. The three-parcel formats are shown below.

Monadic (One Operand)



Dyadic (Two Operand)



The 16-bit source and destination fields are sign extended to 32-bit when they are used in immediate or offset modes. When the 16-bit source and destination fields are used as absolute addresses, extension of the upper 16-bit depends on the setting of the CONFIG PC extension bit.

If the CONFIG PC extension bit is 1, bits 28:16 are replaced with 0 and bits 31:29 (the high-order 3 bits) are copied from bits 31:29 of the program counter. If the CONFIG PC extension bit is zero (0), the upper 16-bits are set to zero (0). The source and destination addressing mode fields are encoded in the same way for both three- and five-parcel instructions. Tables 1-18 and 1-19 detail the three-parcel instruction encoding.

Table 1-18 Encoding (Three-Parcel)

opcode[5:3]	opcode[2:0]							
	000	001	010	011	100	101	110	111
000	monadic	ORI	ANDI	ADDI	MOVA	UREM	MOV	DQM
001	unimp*	unimp*	unimp*	unimp*	TADD	TSUB	unimp*	unimp*
010	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
011	unimp*	unimp*	unimp*	unimp*	unimp*	CMPGT	CMPHI	CMPEQ
100	SUB	OR	AND	ADD	XOR	REM	MUL	DIV
101	unimp*	unimp*	unimp*	unimp*	SHR	USHR	SHL	UDIV
110	SUB3	OR3	AND3	ADD3	XOR3	REM3	MUL3	DIV3
111	unimp*	unimp*	unimp*	unimp*	SHR3	USHR3	SHL3	unimp*

*The unimplemented instruction sequence is performed.

Table 1-19 Monadic Subcoding (Three-Parcel)

subcode[9:3]	subcode[2:0]							
	000	001	010	011	100	101	110	111
0	KCALL	CALL	RETURN	JMP	JMPFN	JMPFY	JMPTN	JMPTY
1	CATCH	ENTER	LDRAA	FLUSHPTE	FLUSHPBE	FLUSHDCE*	unimp*	POPN

*The unimplemented instruction sequence is performed.

1.6.3 Five-Parcel Format

Five-parcel instructions are distinguished by a '11' in the two most significant bits. Five-parcel instructions are encoded similarly to three-parcel instructions.

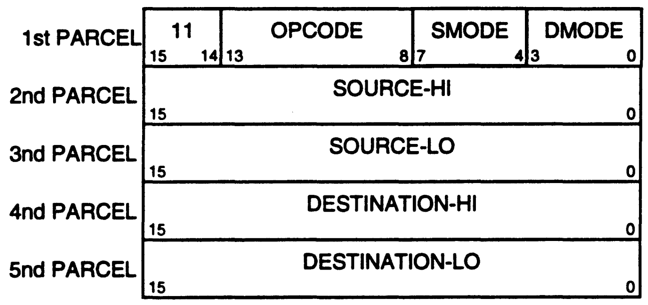


Table 1-20 details the five-parcel encoding.

Table 1-20 Encoding (Five-Parcel)

opcode[5:3]	opcode[2:0]							
	000	001	010	011	100	101	110	111
000	unimp*	ORI	ANDI	ADDI	MOVA	UREM	MOV	DQM
001	unimp*	unimp*	unimp*	unimp*	TADD	TSUB	unimp*	unimp*
010	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*	unimp*
011	unimp*	unimp*	unimp*	unimp*	unimp*	CMPGT	CMPHI	CMPEQ
100	SUB	OR	AND	ADD	XOR	REM	MUL	DIV
101	unimp*	unimp*	unimp*	unimp*	SHR	USHR	SHL	UDIV
110	SUB3	OR3	AND3	ADD3	XOR3	REM3	MUL3	DIV3
111	unimp*	unimp*	unimp*	unimp*	SHR3	USHR3	SHL3	unimp*

The source and destination addressing mode field for three-parcel and five-parcel instructions are encoded the same way (see Table 1-21 thru Table 1-24).

Table 1-21 General Addressing Mode Encoding

Code	Mode	Description
0x0	*$addr:B	Byte absolute
0x1	*$addr:UB	Unsigned byte absolute
0x2	*$addr:H	Half-word absolute
0x3	*$addr:UH	Unsigned half-word absolute
0x4	Roffset:B	Byte stack offset
0x5	Roffset:UB	Unsigned byte stack offset
0x6	Roffset:H	Half-word stack offset
0x7	Roffset:UH	Unsigned half-word stack offset
0x8	*Roffset:B	Byte stack offset indirect
0x9	*Roffset:UB	Unsigned byte stack offset indirect
0xA	*Roffset:H	Half-word stack offset indirect
0xB	*Roffset:UH	Unsigned half-word stack offset indirect
0xC	*$addr:W	Word absolute
0xD	Roffset:W	Word stack offset
0xE	*Roffset:W	Word stack offset indirect
0xF	$data	Immediate

Table 1-22 CPU Modified Addressing Mode Encoding

Code	Mode	Description
0x7	register	CPU prefixed
0xC	*$addr:W	Word absolute
0xD	Roffset:W	Word stack offset
0xE	*Roffset:W	Word stack offset indirect
0xF	$data	Immediate

Table 1-23 CALL/JMP Addressing Mode Encoding

Code	Mode	Description
0xC	**$addr	Absolute indirect
0xD	*Roffset	Stack offset indirect
0xE	Label	Program counter relative
0xF	*$addr	Absolute

Table 1-24 Source/Destination Register Encoding

Code	Register
0x1	MSP
0x2	ISP
0x3	SP
0x4	CONFIG
0x5	PSW
0x6	SHAD
0x7	VB
0x8	STB
0x9	FAULT
0xA	ID
0xB	TIMER1
0xC	TIMER2
0xD	unimp
0xE	unimp
0xF	unimp

1.7 Operand Addressing Modes

The ATT92010 *Hobbit* Microprocessor architecture uses seven addressing modes for accessing data.

- **Immediate** — mode addressing allows a constant to be embedded in the instruction itself. Values up to 32-bits in length are permitted. Shorter values are appropriately sign or 0 extended before use.
- **Absolute** — addressing uses an absolute address in the instruction to access data. Absolute addressing is typically used to reference global variables.
- **Stack Offset** — addressing develops the address of an operand by adding a constant offset in the instruction to the address of the current stack pointer (CSP). For negative offsets, off-chip stack accesses are performed and cache coherency is not maintained. This addressing mode is used to access local variables, temporaries and incoming and outgoing arguments.
- **Stack Offset Indirect** — addressing adds a constant offset in the instruction to the address of the current stack pointer (CSP). The word at this address is fetched and then used as an address to obtain the data operand. The offset must be word aligned. An alignment fault (0x4) is executed if the offset is not word aligned.
- **Absolute Indirect** — addressing stores the operand's address in the instruction. This mode is used for the JMP (Jump), CALL and LDRAA (Load Relative Address into Accumulator) instructions. The operand value should be an instruction address that is parcel (half-word) aligned.
- **Program Counter Relative** — addressing adds a signed, two's complement offset stored in the instruction to the address of the instruction to obtain the operand value. This mode is used only with the JMP (Jump), CALL and LDRAA (Load Relative Address into Accumulator) instructions.
- **Register** — addressing precedes the instruction with a CPU instruction. The CPU instruction is never directly executed but rather it modifies the next instruction's addressing modes for both operands. Code 0x7 allows access to the internal register for use as data. The register number is specified in the operand (source/destination field). Only bits 3:0 are considered for determining the register number. The upper bits are ignored but should be 0 for compatibility.

  At most, one register may be read per instruction. If register 0x0 or 0xD through 0xF is specified, an unimplemented register exception sequence, exception ID 0x6, is performed. Registers can be read in user mode, but if there is a register write in user mode, a privilege violation exception sequence, exception ID 0x5, is performed.

The arithmetic logic unit (ALU) operations generally permit any of the first four addressing modes (Immediate, Absolute, Stack Offset and Stack Offset Indirect) to be used with either operand. Any mode not explicitly mentioned for a given instruction should not be used.

The operand can also have a suffix. The suffixes indicate the size of data operands while a missing suffix implies signed word operands.

- :B — signed byte
- :UB — unsigned byte
- :UH — signed half-word
- :W — word

1.8 Integer Arithmetic Operation

The ATT92010 *Hobbit* Microprocessor offers seven arithmetic instructions:

- ADD a, b ;add a to b
- DIV a, b ;divide b by a, signed
- MUL a, b ;multiply b by a
- REM a, b ;calculate the remainder of signed division of b by a
- SUB a, b ;subtract a from b
- UDIV a, b ;divide b by a, unsigned
- UREM a, b ;calculate the remainder of unsigned division of b by a

REM and UREM are defined in terms of DIV and UDIV, respectively. Operands a and b may be referenced using a variety of addressing modes, with sign interpretation given for byte and half-word arguments.

For the instructions above, the result is stored in b. ADD, DIV, MUL, REM, and SUB as well as other instructions also have a 2 1/2 address version (denoted by a trailing 3) where the result is stored in the accumulator (R4).

1.8.1 Carry Bit

The program status word (PSW) carry bit indicates the occurrence of a borrow during unsigned subtraction or of overflow during unsigned addition or multiplication. Unsigned overflow arises when a result exceeds unsigned (0xFFFFFFFF). In these operations, the PSW carry bit is set when:

```
Unsigned(b) – Unsigned(a) < 0
```

or unsigned overflow on an addition or multiplication:

```
Unsigned(b) {+ or *} Unsigned(a) > Unsigned(0xFFFFFFFF)
```

Unsigned overflow cannot occur in UDIV and UREM.

In the ADD operation, the adder computes the sum of a and b; the word result is delivered and, if carry-out occurs, the PSW carry bit is set. In the SUB operation, the two's complement of a is added to b, and the PSW carry bit is set only if no carry-out occurs.

1.8.2 Overflow Bit

The program status word (PSW) overflow bit signals the occurrence of signed overflow of the word result of an arithmetic operation; this is a result outside the interval:

```
[Signed(0x80000000) to Signed(0x7FFFFFFF)]
```

In terms of the operations above, the PSW overflow bit is set unless:

```
Signed(0x80000000)  (Signed(b) {+, –, or*} Signed(a))  Signed(0x7FFFFFFF)
```

Signed overflow cannot occur in REM. Signed overflow does arise in DIV in precisely the case of 0x80000000 divided by –1, that is 0xFFFFFFFF.

1.8.3 Division and Remainders

Unsigned overflow does not apply to UDIV because its dividend is at most unsigned (0xFFFFFFFF) and its divisor is no less than 1 (except for a zero divisor, which triggers a divide-by-zero exception), so its result is no greater than its dividend. A similar argument applies to DIV, except for the case of overflow.

Like UDIV, unsigned overflow does not apply to UREM. UD and UR are the word results of the UDIV and UREM operations, respectively. Apply these results to operands a and b. UDIV and UREM are related by the formula:

$b = (UD*a) + UR$, where $0 \quad UR < a$

with all values unsigned. UR is no greater than a and therefore no greater than unsigned (0xFFFFFFFF); consequently overflow cannot occur. A similar argument applies to REM.

1.8.4 Tagged Integer Arithmetic

These instructions are useful in object-oriented languages where a given variable may represent different data types at different times during program execution.

- TADD a, b ;tagged add a into b
- TSUB a, b ;tagged subtract a from b

The tagged instructions ensure that the low 2 bits, called tags, of both operands are zero (0), prior to performing the arithmetic operation. If either tag is nonzero, the program status word (PSW) flag bit is set to one (1), and the result is not stored. If both tags are zero, the result is stored only if the operation doesn't result in an arithmetic overflow. If the arithmetic overflow occurs, the PSW flag bit is set to one (1) and the result is not stored.

1.9 Fast Calling Sequence

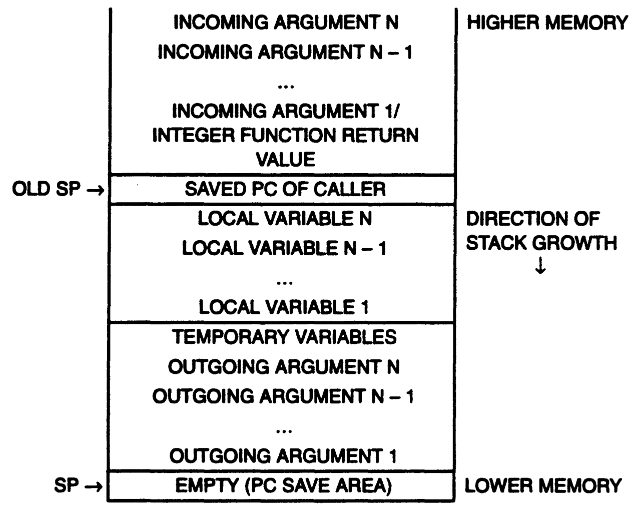
The ATT90210 *Hobbit* Microprocessor provides an efficient procedure calling sequence. Outgoing arguments are moved onto the stack frame. For word arguments, the first argument is stored at current stack pointer (CSP) + 4, the second at CSP + 8, and so one.

The CALL instruction performs an atomic move and jump operation, saving the return point at the CSP and loading the program counter (PC) with the address of the first instruction of the called function. The first instruction of the called function is usually ENTER which adjusts the CSP to allocate its new stack frame.

The last instruction of the called function, RETURN, readjusts the CSP to deallocate its stack frame and then branches to the address pointed to by the CSP. Customarily, a CATCH follows the RETURN in user mode or when the user stack is enabled to refill the stack cache.

This function call overhead—call, allocate, deallocate, and return—can be as little as four clock cycles. Figure 1-5 shows a typical stack frame from the called function's point of view.

Figure 1-5 Stack Frame



The stack grows downward in memory with the stack pointer (SP) always pointing to the top of the stack. The program counter (PC) is stored in this free slot on a function call (or unimplemented instruction exception). This avoids having to adjust the current stack pointer (CSP) to save or restore the PC. The PC is the only machine register implicitly saved during a function call.

Above the saved PC slot in the stack frame is a large area to store outgoing arguments for any call from the current function. Above the outgoing arguments temporary values and local variables are stored. This permits outgoing arguments to be calculated in place with stack offset addressing modes. This statically allocated stack frame allows the CSP to be updated only on function entry and function return.

Traditional PUSH or POP instructions that automatically adjust the CSP are intentionally avoided. POPN is provided to deallocate from the stack frame and is useful in tail recursion. Side effects to the CSP are nearly eliminated and operand address generation for subsequent instructions can smoothly proceed in a pipeline implementation.

1.10 Prefetching Strategy

The ATT92010 *Hobbit* Microprocessor has two types of instruction fetching. Both are selectable through the CONFIG prefetch mode bit.

- Aggressive Prefetching
- Demand Fetching

When aggressive prefetching is enabled (CONFIG prefetch mode bit = 1), the microprocessor prefetch unit fetches text (not been previously fetched and stored in the prefetch buffer memory), in quad-word pieces consisting of two double-word I/O requests.

Text is prefetched sequentially until a branch (predicted jump, unconditional jump, CALL, CRET, KCALL, KRET, or RETURN) is decoded. If the target of the branch is encoded in the instruction (non-indirect), prefetching continues from the target (if it is not already in the prefetch buffer). If the target is indirect, prefetching stops and waits for a demand fetch request from the execution unit.

A demand fetch is requested if the execution unit takes a unpredicted or indirect branch and the target has not been previously decoded. If at any time while the prefetch unit is prefetching sequential code and following predicted branches a demand fetch is requested, any I/O requested by the unit will complete, and prefetching begins anew from the execution unit requested target.

1.10.1 Branch Prediction and Branch Folding

Branches break the flow of instruction execution and may degrade the performance of a pipelined microprocessor. More important, the target of a conditional jump is not known until the instruction is executed. The ATT90210 *Hobbit* Microprocessor solves these problems in two ways:

- Static Branch Prediction
- Prefetch Decode Unit (PDU)

The instruction format provides a static branch prediction field. The field is set at compile time and can indicate whether it is likely to take the conditional branch or not. The prefetch decode unit (PDU) continues prefetching along the predicted path of a conditional jump, the instructions can be issued and executed into the pipeline without any discontinuity.

Second, the PDU assigns a next-PC (program counter) and alternate-next-PC field for each decoded instruction.

1.10.2 Conditional Branches

Conditional branches are specified by first setting the program status word (PSW) flag bit using one of the compare instructions (CMPEQ, CMPGT, CMPHI) or using a miscellaneous instruction (TESTC or TESTV). Then, finish with a conditional jump instruction (JMPTY, JMPTN, JMPFY, or JMPFN).

The jump doesn't need to be the next instruction after the flag is set. The pipeline runs more efficiently if three instructions, which do not reference off-chip memory, are sandwiched between the compare instructions and the jump.

The Y or N at the end of the conditional jump instruction is the prediction of the branch (Y-jump, N-continue).

1.10.3 Tracing

Instruction tracing is supported by the program status word *trace basic block* or *trace instruction* bits. These bits control when tracing is enabled. If an instruction is traceable, a trace exception is taken after the instruction completes execution. The program counter (PC) saved on the interrupt stack is the next instruction PC.

Instructions before folded branches cannot be traced. For example, if a jump is folded into the previous instruction, the trace will occur after the jump. To avoid jumps being folded, all jumps must be encoded as three-parcel.

Event sequences are nontraceable, including exceptions and interrupts. The unimplemented instruction sequence is traceable if the trace bits are not altered. CRET, KCALL, and KRET are always non-traceable.

1.11 Event Processing

There are several sequences that trigger the ATT92010 *Hobbit* Microprocessor that are not invoked by the regular instruction set. These events include, in order of priority:

- Reset
- Interrupt
- Exception

The sequences executed by the ATT92010 *Hobbit* Microprocessor for each of these events are discussed in the following sections. In all cases, interrupts are inhibited while an event processing sequence (the sequence that initiates the event handler) is in progress.

The processing of exceptions and interrupts includes saving the program counter (PC) and the program status word (PSW) on the interrupt stack. For instructions that change the PC, the *current PC* is defined as one of the following.

- CALL and JUMP — If the location pointed to by the instruction cannot be referenced, a fetch-fault results and the PC stored on the interrupt stack is the target PC, not the PC of the instruction. If the indirection word of an indirect instruction cannot be referenced, a read-fault results and the PC stored on the interrupt stack is that of the instruction.
- KCALL — If the location pointed to by the KCALL PC entry in the vector cannot be referenced, a fetch-fault results and the PC stored on the interrupt stack is the target PC, not the PC of the original KCALL.
- CRET, KRET, and RETURN — If the location pointed to by the new PC value cannot be referenced, a fetch-fault results and the PC stored on the interrupt stack is the new PC value, not the address of the instruction.

1.11.1 Reset

The ATT92010 *Hobbit* Microprocessor enters the reset sequence when:

- The external reset pin ($\overline{\text{HRESET}}$) is asserted.
- A memory fault, which is signaled either externally or by the memory management unit (MMU), occurs
  —when attempting to read or write the interrupt stack during any event processing sequence.
  —when attempting to read from the vector table during any event processing sequence.

The reset sequence is:

```
Disable interrupts
Flush the PFB and IC
if reset
    SHAD = 0x0
else
    SHAD = PSW
PSW = 0x0
CONFIG = 0x0
PC = 0x0
Enable NMI interrupts
```

The shadow register (SHAD) is set either to 0x0 or the current program status word (PSW) depending on the reset type. Independent of the reset type, the prefetch buffer (PFB) and instruction cache (IC) are flushed and the PSW, CONFIG, and program counter (PC) are initialized to 0x0. Initialization of the PSW register sets:

- the execution level to kernel mode
- physical addressing to enable
- tracing to disable
- interrupts are inhibited
- the interrupt stack pointer (ISP) as the current stack pointer (CSP)

Initialization of the CONFIG register sets:

- disables all on-chip caches
- disables timer interrupts
- and selects demand prefetching

Initialization in the PC register starts executing instructions at physical address 0x0.

Note If the reset sequence was initiated by the external reset pin, the stack pointer (SP) and the maximum stack pointer (MSP) are undefined. The caches should not be enabled until these registers are assigned values since the range check circuitry would not know whether an address should access the on-chip stack cache or off-chip memory.

1.11.2 Interrupt

An interrupt is signaled when an external device requests service on the interrupt request input lines (IL[2:0]) or either Timer1 or Timer2 overflows with the interrupts enabled.

The three input lines associated with external interrupts and the timer interrupts, which are asserted at level 1, are compared with the program status word (PSW) interrupt priority level (IPL) field. If the interrupt request is less than the IPL field, the interrupt can be serviced. An IPL field of 7 allows interrupts at levels 0 through 6. An IPL field of 0 inhibits interrupts 1 through 6 and allows interrupts at level 0 only. This is referred to as a nonmaskable interrupt (NMI). Table 1-25 list the interrupt levels.

Table 1-25 Interrupt Levels

IL[2:0]	Interrupt Level
000	NMI
001	Level 1
010	Level 2
011	Level 3
100	Level 4
101	Level 5
110	Level 6
111	No interrupt

The interrupt request input lines IL[2:0] must be asserted with the same value for at least two cycles before an interrupt is recognized by the ATT92010 *Hobbit* Microprocessor. The interrupt should remain asserted until the interrupt handler clears it. If the interrupt is accepted, the request enters at the top of the execution unit pipeline. Then all further interrupts are disabled until completion of the interrupt sequence. The ATT92010 *Hobbit* Microprocessor does not indicate when it is servicing an interrupt other than the I/O caused by the interrupt handler.

Nonmaskable Interrupt

A nonmaskable interrupt (NMI) is generated by setting IL[2:0] to 0x0. An interrupt at level 0 is *edge sensitive*, that is, it must be deasserted for at least two cycles before another interrupt at any level is recognized. When an interrupt enters the execution pipeline, all interrupts are disabled, including NMI. After the interrupt sequence completes, the NMI will be serviced if it is still asserted.

Most instructions complete execution before the interrupt request enters the top of the execution unit pipeline. CATCH, ENTER, MUL[3], DIV[3], REM[3], UDIV, and UREM are interruptible. The CATCH portion of CRET is interruptible. The PC stored on the interrupt stack is the address of the interrupted instruction for transparently resuming execution. CATCH, ENTER, and the *CATCH* portion of CRET continues (as opposed to restarting).

Interrupt Sequence

When the interrupt is serviced, the sequence is:

```
Disable interrupts
if (CSP == ISP) ISP = SHAD
else SP =SHAD
*(ISP–8) = PC of interrupted instruction    /* Becomes R8 with respect to new ISP */
*(ISP–4) = PSW                              /* Becomes R12 with respect to new ISP */
ISP –= 16
SHAD = ISP
PC = *(VB + 16 + (4 x interrupt level))
PSW &= 0xFFFF0000
Enable NMI interrupts
```

Where *interrupt level* is the value of the IL[2:0] lines producing the interrupt. Note that the interrupt sequence is almost the same as the KCALL sequence (the event frame left on the interrupt stack is the same). Consequently, a KRET instruction is sufficient for returning from an interrupt; interrupts are disabled during this processing.

1.11.3 Exceptions

Exceptions signal an error in a program. The ATT92010 *Hobbit* Microprocessor recognizes the exceptions listed in Table 1-26.

Table 1-26 Exception Identifier

Code	Exception
0x1	Integer zero-divide
0x2	Trace
0x3	Illegal instruction
0x4	Alignment fault
0x5	Privilege violation
0x6	Unimplemented register
0x7	Fetch fault
0x8	Data read fault
0x9	Data write fault
0xA	Memory access I/O bus fault
0xB	MMU table walk bus fault

The exception handler must always be present.

Exception Sequence

The sequence is similar to the KCALL sequence. If the target address of a CALL, CRET, JMP, KCALL, KRET, or RETURN instruction, or of an interrupt, causes a memory fault, the PC saved on the interrupt stack is the target PC, not the address of the current instruction. The sequence is:

```
Disable interrupts
if (CSP == ISP) ISP = SHAD
else SP = SHAD
* (ISP–12) = exception identifier      /* Becomes R4 with respect to new ISP */
*(ISP–8) = PC of faulted instruction   /* Becomes R8 with respect to new ISP */
*(ISP–4) = PSW                         /* Becomes R12 with respect to new ISP */
ISP –= 16
SHAD = ISP
PC = *(VB + 4)
PSW &= 0xFFFF0000
Enable NMI interrupts
```

In exception IDs 0x8 and 0x9, the 32-bit operand aligned virtual address of faulted access is saved in the Fault Register.

For a text fetch bus error or a data read bus error, the program counter (PC) placed on the interrupt stack is the address of the instruction with the faulting address.

For a data write bus error, the PC placed on the interrupt stack is not the PC of the instruction associated with the faulted access. Because of the *unhinged* nature of the stores in the ATT92010 *Hobbit* Microprocessor, the PC stored is the PC of the instruction which was at the bottom of the execution pipeline when the fault occurred, and not the PC of the instruction with which the faulted store is associated.

1.11.4 Unimplemented Instruction

An attempt to execute an unimplemented opcode results in an unimplemented instruction sequence. This sequence is faster than the exception sequence for software emulation of extended instructions. Since an unimplemented instruction can occur in either execution mode, the unimplemented instruction handler should be in both the user and kernel address space.

If an unimplemented instruction has an addressing mode that is illegal for that instruction class, it is considered an illegal instruction (exception ID 0x3). Specifically:

- An unimplemented monadic instruction is illegal if it has a nonword addressing mode (<0xC).
- An unimplemented instruction is illegal if it follows a CPU instruction and contains an illegal addressing mode, or combination of modes.
- A RETURN instruction with a negative operand.

There are no tests performed on the addressing modes of unimplemented dyadic instructions which do not follow CPU instructions.

Unimplemented Instruction Sequence

The sequence is:

```
*(CSP) = PC of unimplemented opcode
PC = *(VB + 12)
```

Where current stack pointer (CSP) is either stack pointer (SP) or interrupt stack pointer (ISP), depending on the state of the PSW current stack pointer bit.

Trapped Niladic Exception

An attempt to execute a one-parcel niladic with an opcode in the range 0x200 through 0x3FF results in a variant of an unimplemented instruction sequence known as a *trapped niladic exception.* This sequence is the same as the unimplemented instruction sequence except VB + 8 is used for the vector. The trapped niladic handler should be in both the user and kernel address space. The sequence is:

```
*(CSP) = PC of unimplemented opcode
PC = *(VB + 8)
```

Where current stack pointer (CSP) is either stack pointer (SP) or interrupt stack pointer (ISP), depending on the state of the PSW current stack pointer bit.

1.11.5 Event Processing Priority

Since several event requests can be generated simultaneously, an event processing priority has been established. The priorities assigned to each event type request are:

1. Reset
2. Interrupts
3. Trace
4. Instruction fetch faults
5. Illegal instructions
6. Unimplemented instructions/trapped niladic
7. Unimplemented registers
8. Alignment faults
9. Data read and write and read bus error faults
10. Privilege violation
11. Divide by zero

The high-priority events (reset and interrupts) occur independently of an instruction execution. All other events are associated with a particular instruction. During some internal sequences, interrupts are disabled. Many events are mutually exclusive of each other and cannot occur at the same time or within the same instruction.

Chapter 2

Memory Management

The ATT92010 *Hobbit* Microprocessor has an on-chip memory management unit (MMU), which can translate virtual addresses, as seen by a programmer, into physical addresses. The two methods for address translation are:

- Paged segments
- Nonpaged segments

The 32-bit virtual address space is divided into 1,024 segments, each representing 4 MB of virtual addresses with a 4 MB alignment. Paged segments are further divided into 1024-word pages (see Figure 2-1). Nonpaged segments provide a variable-sized contiguous segment of memory (see Figure 2-2). In paged segment address translation, each page can be mapped anywhere in the 32-bit physical address space.

2.1 Address Translation

Address translation is enabled by setting the program status word (PSW) virtual/physical bit to 1 (VP-1). To speed paged segment address translation, the ATT92010 *Hobbit* Microprocessor has two translation lookaside buffers (TLBs)—one for text addresses and one for data addresses. Each TLB has 32 entries and is fully associative. Two nonpaged segment registers (NPSRs), one for a text address and one for a data address, speed nonpaged segment address translation.

Additionally, to provide a physical prefetch buffer, a micro-TLB is provided for text references in the present page. This micro-TLB contains the last translation used by the prefetch unit and provides zero-cycle address translation. If the micro-TLB misses, one cycle is required for update if the address translation hits in the text TLB or text NPSR.

If an address is not contained in the appropriate TLB or NPSR, the on-chip MMU automatically fetches the appropriate entry by walking the memory management tables.

Figure 2-1 Paged Segment Address Mapping

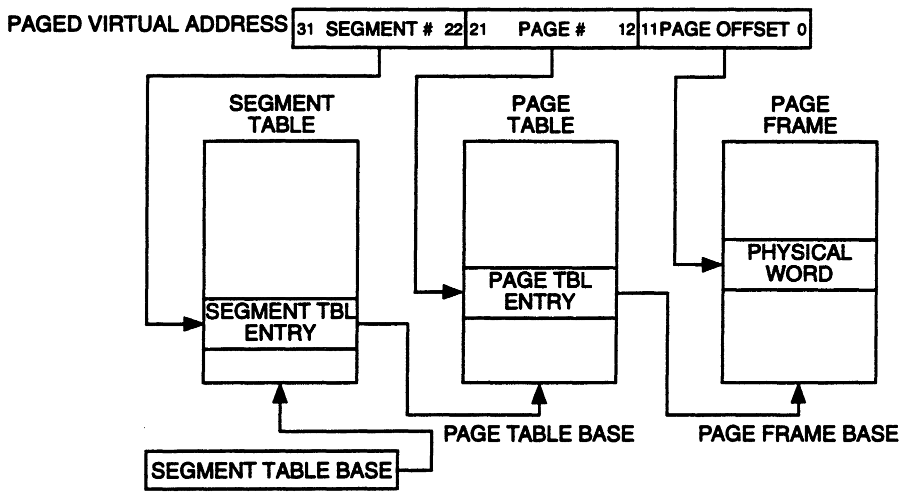
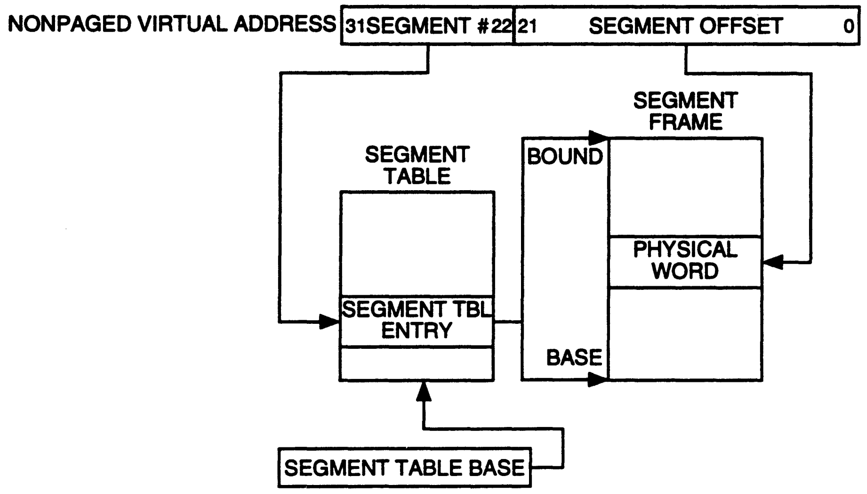


Figure 2-2 Nonpaged Segment Address Mapping



2.2 Address Mapping

All addresses in the ATT92010 *Hobbit* Microprocessor are translated by walking a series of map tables. All map tables in the ATT92010 memory mapping scheme are 4,096 bytes long (one-page frame). All addresses contained within a memory management table are physical addresses, so address translation is not recursive.

Address mapping checks the validity of virtual addresses and translates them into physical addresses. A virtual address is flagged as illegal if one of the following happens:

- There is no valid physical mapping
- User execution level code attempts to access kernel execution level addresses
- A store is attempted to read-only data
- Any violation is signaled as a memory fault:

  Fetch fault — If, during an address translation for text, there is no physical mapping or an attempt is made to access a kernel only page while in user mode, this fault is signaled. Note that a fetch fault is generated only on demand fetches and only stops fetching, until a demand fetch, if aggressive fetching is enabled by the PSW prefetch bit.

  Read fault — If, during an address translation for reading data, there is no physical mapping or an attempt is made to access a kernel only page while in user mode, this fault is signaled. This fault can be ignored if the read was requested because of a mispredicted branch.

  Write fault — If, during an address translation for either writing data or while executing one of the stack manipulation instructions, there is no physical page, an attempt is made to access a kernel only page in user mode, or an attempt is made to write to a nonwritable page, this fault is signaled.

2.2.1 Paged Segment Addresses

A page frame is a contiguous region of 4,096 bytes, beginning at an address evenly divisible by 4,096 (the low 12-bits of the address are all 0). Because all page frames begin on page boundaries, additions are not necessary to calculate addresses. When paged segment translation is in use, virtual addresses are divided into the following three fields:

- Segment number
- Page number
- Page offset

2.2.2 Nonpaged Segment Addresses

When nonpaged segment translation is in use, virtual addresses are divided into the following two fields:

- Segment number
- Segment offset

2.3 Segment Tables

The segment number selects one entry from 1,024 entries in the segment table—a 4 KB table located in one page frame in physical memory. Each segment table entry is 4 bytes long and contains the base address of a page table or the base address and size of a nonpaged segment. The base address of the segment table is contained in the segment table base (STB) register.

The address of a segment table entry is formed by concatenating the upper 20 bits of the segment table base register with the upper 10 bits of the virtual address: the base address field in the segment table base defines the beginning of a segment table in physical memory, and the segment number field of the virtual address defines a word within the segment table.

There are two possible formats for a segment table entry. Paged segments have referenced and modified bits for enhanced memory management. Nonpaged segments only require the segment table to resolve references.

2.3.1 Paged Segment Table Entries

The segment table for paged segments defines 1,024 segments, each 1,024 pages long (for a total of 4,294,967,296 bytes). Segments are defined as a series of pages, so there may be holes in a segment's address space. There is no length specification for a segment: the validity of constituent pages defines a segment's extent. Each paged segment table entry defines the base of a page table.

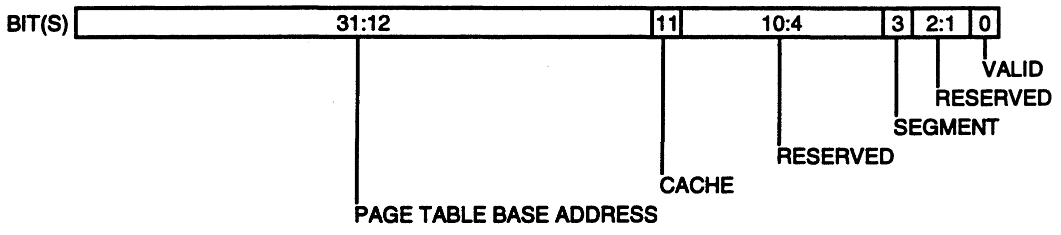


Table 2-1 Paged Segment Table Entry

Bit(s)	Name/Description
31:12	**Page Table Base Address.** The base address in physical memory of the page table.
11	**Cache.** If 1, $\overline{\text{NCACHE}}$ is deasserted when fetching page table entries.
10:4	**Reserved.**
3	**Segment.** 0 for paged segment translation.
2:1	**Reserved.**
0	**Valid.** If 1, the entry is valid.

2.3.2 Nonpaged Segment Table Entries

The segment table for nonpaged segments defines the base and bound of a segment.

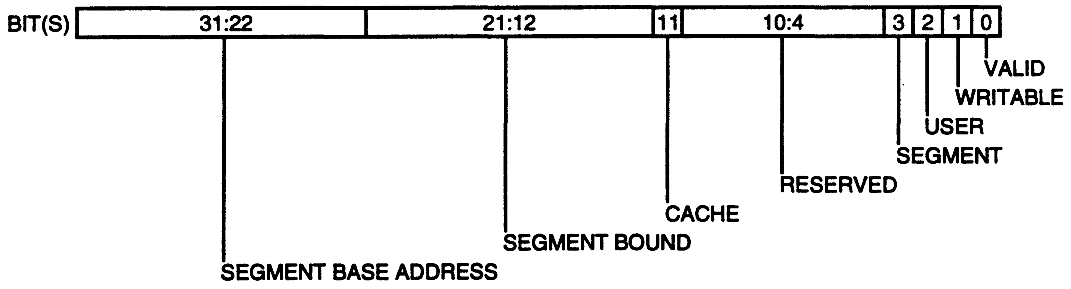


Table 2-2 Nonpaged Segment Table Entry

Bit(s)	Name/Description
31:22	**Segment Base Address.** These bits contain the base address of the segment in physical memory.
21:12	**Segment Bound.** These bits contain the size of the segment, ranging from 4,096 bytes (0x0) to 4 MBs (0x3FF) in increments of 4,096 bytes.
11	**Cache.** If 0, $\overline{\text{NCACHE}}$ is asserted when accessing this segment. Text fetches will not be cached in the prefetch buffer cache, but they will be cached in the decoded instruction cache. If 1, $\overline{\text{NCACHE}}$ is deasserted when accessing nonpaged segments. This bit has no effect on the use of the stack cache.
10:4	**Reserved.**
3	**Segment.** A 1 for nonpaged segment translation.
2	**User.** If 1, the segment can be accessed at user execution level (all valid segments can be accessed at kernel level).
1	**Writable.** If 1, the segment can be written (all valid segments can be read).
0	**Valid.** If 1, the segment is valid.

The segment offset field of the virtual address defines the byte within the segment frame in which the virtual address is mapped. The physical address consists of the segment base address from the segment table entry concatenated with the segment offset field of the virtual address. If a protection violation is detected, no memory access is made and a memory fault exception is executed.

2.3.3 Mixed Paged and Nonpaged Segment Tables

Since the segment bit in the segment table entry controls if the segment table entry is paged or nonpaged, a segment table can contain both paged and nonpaged entries.

2.4 Page Tables

The address of a page table entry is formed by concatenating the upper 20 bits of the segment table entry with bits 21:12 of the virtual address (the page number).

A page table entry defines the physical address corresponding to the virtual address and provides protection information and other data available for paging algorithms. The reference and modified bits are automatically set by the on-chip MMU, but they must be cleared by software when needed.

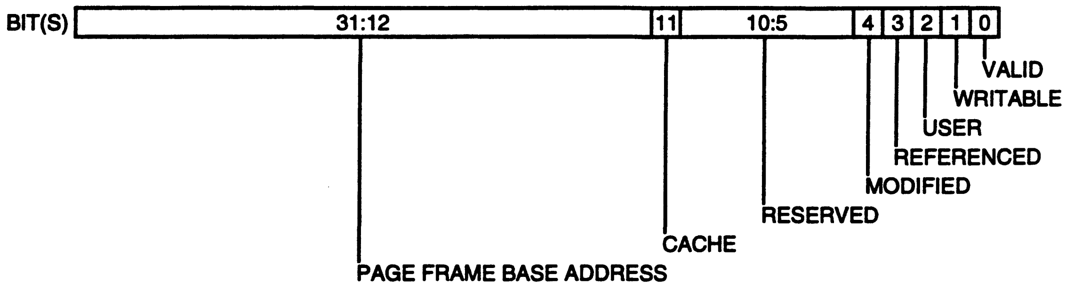


Table 2-3 Page Table Entry

Bit(s)	Name/Description
31:12	**Page Frame Base Address.** These bits contain the base address in physical memory of the page frame.
11	**Cache.** If 0, $\overline{\text{NCACHE}}$ is asserted when accessing this page. Text fetches will not be cached in the prefetch buffer cache, but they will be cached in the decoded instruction cache. If 1, $\overline{\text{NCACHE}}$ is deasserted when accessing this page. This bit has no effect on the use of the stack cache.
10:5	**Reserved.**
4	**Modified.** Set to 1 when a write occurs within the page. On subsequent writes to this page, the memory copy of the page table entry is not accessed to set this bit again. If a direct write to the memory copy of the page table entry changes this bit, the entry should be flushed from the TLB using the FLUSHPTE instruction.
3	**Referenced.** Set to 1 when a page is first referenced. On subsequent references to this page, the memory copy of the PTE is not accessed to set this bit again. If a direct write to the memory copy of the PTE changes this bit, the entry should be flushed from the TLB using the FLUSHPTE instruction.
2	**User Bit.** If 1, the page can be accessed at user execution level (all valid pages can be accessed by the kernel).
1	**Writable.** If 1, the page can be written (all valid pages can be read).
0	**Valid.** If 1, the page is valid.

The page offset field of the virtual address defines the byte within the page frame in which the virtual address is mapped. The physical address consists of the page frame base address from the page table entry concatenated with the page offset field of the virtual address. If a protection violation is detected, no memory access is made and a memory fault exception is executed.

2.5 Memory Management Operations

Both translation lookaside buffers (TLBs) and nonpaged segment registers (NPSRs) are completely flushed whenever the ATT92010 *Hobbit* Microprocessor is reset (either by asserting the external reset pin, or the detection of an internal event that causes the ATT92010 to reset). The TLBs and NPSRs are also flushed whenever the segment table base register is written.

Individual TLB and NPSR entries may be flushed using the FLUSHPTE instruction. If the effective address in the FLUSHPTE instruction is cached in either the translation lookaside buffer or the nonpaged segment register, the TLB or NPSR entry is marked invalid. Any subsequent access of that virtual address will be translated by the full memory map table walk.

The FLUSHPTE instruction is not privileged, so a user process may flush any or all entries in the on-chip TLBs or NPSRs. Although this may degrade the performance of the process, it does not affect correctness, since the memory management tables in physical memory define the address mapping and the FLUSHPTE instruction does not alter the tables in memory.

$\overline{\text{LOCK}}$ is asserted when page table entries are fetched. If the R and M bits of the entry are current, $\overline{\text{LOCK}}$ is cleared. If either R or M bits must be updated, the page table entry is written back to memory with $\overline{\text{LOCK}}$ still asserted. $\overline{\text{LOCK}}$ is deasserted when the write completes.

If there is an external bus error signaled during the memory management table walk, the ATT92010 *Hobbit* Microprocessor will take an exception.

2.6 MMU Performance

Table 2-4 details the performance of address translation. These performance numbers do not include the time required to access the actual item. In the table an *A* represents the I/O delay for a single word access.

Table 2-4 Address Translation Performance

Condition	Penalty
Text reference, micro-TLB miss, TLB/NPSR miss, paged segment walk, R bit modified	3A + 3
Text reference, micro-TLB miss, TLB/NPSR miss, paged segment walk, R bit previously set	2A + 3
Text reference, micro-TLB miss, TLB/NPSR miss, nonpaged segment walk	A + 1
Text reference, micro-TLB miss, TLB/NPSR hit	1
Text reference, micro-TLB hit	0
Data read, TLB/NPSR miss, paged segment walk, R bit modified	3A + 3
Data read, TLB/NPSR miss, paged segment walk, R bit previously set	2A + 3
Data read, TLB/NPSR miss, nonpaged segment walk	A + 3
Data read, TLB/NPSR hit	0
Data write, TLB/NPSR miss, paged segment walk, R and/or M bit modified	3A + 3
Data write, TLB/NPSR miss, paged segment walk, R and M bit previously set	2A + 3
Data write, TLB/NPSR miss, nonpaged segment walk	A + 3
Data write, TLB/NPSR hit and M bit previously set	0

Chapter 3 Instruction Set

The instruction set falls into eight categories: Arithmetic, Compare, Logical, Move, Program Control, Shift, Tagged, and Other. There are two special notations used within these categories: [] and (|). For example, ADD[3] indicates that both the ADD and ADD3 instructions apply. JMP (F|T)(Y|N) indicates that JMPFY, JMPFN, JMPTY, and JMPTN instructions exist.

3.1 Format

The general instruction format is:

```
Instruction source, destination
```

where the instruction can contain a 3 indicating that the destination is the accumulator (R4). Otherwise, the destination is the second operand.

Table 3-1 list the instructions, in alphabetic order, identifies the type and function.

Table 3-1 Instructions *(Sheet 1 of 2)*

Instruction	Function	Type
ADD[3]	Add	Arithmetic
ADDI	Add interlocked	Arithmetic
AND[3]	Bitwise logical AND	Logical
ANDI	Bitwise logical AND interlocked	Logical
CALL	Call subroutine C	Program Control
CATCH	Fill stack cache	Program Control
CLRE	Clear PSW enter guard bit	Other
CMPEQ	Equality comparison	Compare
CMPGT	Signed greater than comparison	Compare
CMPHI	High comparison (unsigned greater than)	Compare
CPU	Register access escape	Other
CRET	Return from kernel with context	Program Control

Table 3-31 Instructions *(Sheet2 of 2)*

Instruction	Function	Type
DIV[3]	Divide	Arithmetic
DQM	Double-word or Quad-word move	Move
ENTER	Enter subroutine	Program Control
FLUSHD	Flush data cache	Other
FLUSHDCE	Flush data cache entry	Other
FLUSHI	Flush the decoded instruction cache	Other
FLUSHP	Flush the prefetch buffer	Other
FLUSHPBE	Flush an entry in prefetch buffer	Other
FLUSHPTE	Flush a page table entry in the TLBs	Other
JMP	Unconditional Jump	Program Control
JMP(FIT) (YIN)	Conditional jump based on PSW flag bit	Program Control
KCALL	Kernel call	Program Control
KRET	Return from kernel	Program Control
LDRAA	Load relative address into the accumulator	Move
MOV	Move	Move
MOVA	Move address	Move
MUL[3]	Multiply	Arithmetic
NOP	No operation	Other
OR[3]	Bitwise logical OR	Logical
ORI	Bitwise logical OR interlocked	Logical
POPN	Free *n* entries from stack space	Program Control
RETURN	Return from subroutine	Program Control
REM[3]	Remainder	Arithmetic
SHL[3]	Left shift	Shift
SHR[3]	Arithmetic right shift	Shift
SUB[3]	Subtract	Arithmetic
TADD	Tagged addition	Tagged
TESTC	Test program status word carry	Other
TESTV	Test program status word overflow	Other
TSUB	Tagged subtraction	Tagged
UDIV	Unsigned divide	Arithmetic
UREM	Unsigned remainder	Arithmetic
USHR[3]	Logical right shift	Shift
XOR[3]	Bitwise logical exclusive OR	Shift

3.2 Pipeline Considerations

Certain combinations of instructions may produce unexpected results because of the pipelining within the microprocessor. Most of these cases are noted in the descriptions of each instruction that follows this section. The following is a summary of these combinations.

- There must be at least two instructions separating an instructions that sets the Carry and Overflow bits (such as ADD or MUL) and an instruction that explicitly reads the program status word, using the CPU prefix. The intervening instructions are not necessary if the Carry and Overflow bits are queried with the TESTC or TESTV instructions.
- An ENTER cannot immediately follow the invalidation of the page into which it enters. There should be two instructions between the invalidation of the page and the ENTER instruction to allow the memory table to update.
- If an ADD, SHL, or MUL instruction with a destination size of byte or half-word results in a number that overflows the destination size, but can fit in a 32-bit word, a subsequent instruction may use the 32-bit version of the result, rather than a truncated 8- or 16-bit result. The non-truncated result may affect the computation if the MUL, USHR, or ADD overflows its byte or half-word destination and

  —the following instruction is a divide or a right shift and it uses the destination of the first instruction as one of its operands, or

  —the destination of the second instruction is larger than the destination of the first instruction.

  Using the truncated version of the result can be forced by interposing two instructions between the MUL, SHL, or ADD and the following instruction. For example:

```
MUL       $0x&F,R4:BMUL$0x7F, R4:B$$
USHR      $4,R4:B →  instr
          instr
          USHR$4,R4:B

MUL       $0x7F,R4:BMUL$0x&F,R4:B
MOV       R4:B, R8:L →instr
          instr
          MOVR4:B, R8:L
```

- An instruction that reads the SHAD register cannot be executed immediately after ENTER or RETURN. Two NOPs should be placed between such instructions to permit the writing of the SHAD register. For example:

```
ENTER         R-16ENTERR-16
MOV           $new,%SHADNOP
              NOP
              MOV$new,%SHAD
CALL          routineCALLroutine
ADD           $16,%SHADNOP
              NOP
              ADD$16,%SHAD
```

3.3 Descriptions

The following pages contain detailed descriptions of the instruction set. Abbreviations used in the following pages are defined in Table 3-2.

Table 3-2 Abbreviations

Abv.	Description
abs32	A 32-bit value with any of the two-word operand addressing modes: PC-relative or absolute
fgen[n]	Any of the following modes with a value that can fit in *n*-bits: absolute, immediate, stack offset or stack offset indirect.
flow32	A 32-bit value with any of the four-word operand addressing modes (modes ≥ 0xC): absolute, absolute indirect, PC-relative or stack offset indirect mode.
gen[n]	Any of the following modes with a value that can fit in *n*-bits: absolute, immediate, stack offset or stack offset indirect. Note that the CPU prefix instruction modifies the meaning of theses addressing modes.
imm[n]	A two's complement constant in the range -2^{n-1} through $2^{n-1}-1$
istk5	An Indirect Stack Offset mode of type word with the offset a number divisible by four in the range 0 through 124.
pvtrl10	A PC-offset mode where the offset is a number divisible by 2 in the range -1024 through -1022
stk5	A Stack Offset mode with the offset number in the range 0 through 124 and is divisible by four, (the operand size of word).
stk8	A Stack Offset mode which is operand size of word.
stk32	A Stack Offset mode with the offset any 32-bit number.
uimm[n]	An unsigned constant in the range 0 through 2^n-1
wai[n]	An unsigned constant in the range 0 through 2^n-1 which is multiplied by 4 (word-aligned).
word32	A 32-bit value with any of the four word operand addressing modes (modes > = 0x C): absolute, immediate modes, stack offset or stack offset indirect.

Name: **ADD—Addition**

Format: ADD[3] src, dst

Operation: ADD:

dst += dst
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

ADD3:

Acc = dst + src
"unsigned overflow" ? PSW.C = 1 : PSW.C = 0
"signed overflow" ? PSW.V = 1 : PSW.V = 0

Description:

The source operand is added to the destination operand and the sum is placed in either the destination (ADD) or the Accumulator (ADD3).

The PSW C-bit is set to 1 on unsigned overflow and the PSW V-bit is set to 1 on signed overflow, otherwise the PSW C- and V-bits are set to 0 (zero).

Encodings:

length	opcode	instruction	src	dst
2	0x0D	ADD3	wai5,	stk5
2	0x14	ADD	imm5,	stk5
2	0x15	ADD3	imm5,	stk5
2	0x16	ADD	stk5,	stk5
2	0x17	ADD3	stk5	stk5
6	0x23	ADD	gen16,	gen16
6	0x33	ADD3	gen16,	gen16
10	0x23	ADD	gen32,	gen32
10	0x33	ADD3	gen32,	gen32

Name: **ADDI—addition interlocked**

Format: ADDI src, dst

Operation:

```
hidden dst
dst += src
Acc = hidden
```

Description:

The source operand is added to the destination operand, and the sum is placed in the destination. $\overline{\text{LOCK}}$ is asserted during the fetch of dst if dst is in memory and not in the stack cache. $\overline{\text{LOCK}}$ is deasserted at the completion of the final store to dst. No other accesses are done between the fetch and store of dst. The original value of dst is placed in the accumulator. If the accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

The PSW carry and overflow bits are not affected by ADDI.

Encodings:

length	opcode	instruction	src	dst
6	0x03	ADDI	gen16	gen16
10	0x03	ADDI	gen32	gen32

Notes

Pipeline bypass hazards associated with semaphore operations are avoided in the ATT92010 *Hobbit* Microprocessor by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4; hence, no operation is performed.

If the accumulator is not in the stack cache, CSP == MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator must not fault in any manner; ADDI is not restartable from this point of the operation.

Name: **AND—bitwise logical AND**

Format: AND[3] src, dst

Operation:
AND:
dst & = src

AND3:
Acc = dst & src

Description:

A bitwise logical AND operation is performed on the source and destination operands. The result is placed in either the destination (AND) or the accumulator (AND3).

Encodings:

length	**opcode**	**instruction**	**src**	**dst**
2	0x0E	AND3	imm5,	stk5
2	0x0F	AND	stk5	stk5
6	0x22	AND	gen16	gen16
6	0x32	AND3	gen16	gen16
10	0x22	AND	gen32	gen32
10	0x32	AND3	gen32	gen32

Name: **ANDI—bitwise logical AND interlocked**

Format: ANDI src, dst

Operation:

```
hidden = dst
dst &= src
Acc = hidden
```

Description:

A bitwise logical AND operation is performed on the source and destination operands, and the result is placed in the destination. $\overline{\text{LOCK}}$ is asserted during the fetch of dst if dst is in memory and not in the stack cache. $\overline{\text{LOCK}}$ is deasserted at the completion of the final store to dst. No other accesses are done between the fetch and store of dst. The original value of dst is placed in the accumulator. If the accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

Encodings:

length	opcode	instruction	src	dst
6	0x02	ANDI	gen16	gen16
10	0x02	ANDI	gen32	gen32

Notes:

Pipeline bypass hazards associated with semaphore operations are avoided in the *Hobbit* microprocessor by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4; hence, no operation is performed.

If the accumulator is not in the stack cache, CSP == MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator must not fault in any manner; ANDI is not restartable from this point of the operation.

Name:	**CALL—subroutine C**
Format:	CALL src
Operation:	* (CSP) = next PC /*save return PC in R0*/ PC = src

Description:

The next program counter (PC) value (return address) is stored at the location indicated by the stack pointer (SP) or the interrupt stack pointer (ISP), whichever is the current stack pointer (CSP). The source operand (subroutine entry point) becomes the new PC value.

Encodings:

length	opcode	subcode	instruction	src
2	0x01	—	CALL	pcrel10
6	0x00	0x1	CALL	flow 32

Notes:

If the location pointed to by CALL cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC, not the PC of the original CALL. The address of the original CALL instruction is not saved. In the event of an indirect CALL, if the ATT92010 *Hobbit* Microprocessor cannot reference the indirection word, a read-fault results and the PC stored on the interrupt stack is that of the indirect CALL. In either case, fetch-fault or read-fault, the correct return PC is saved in R0.

Name: **CATCH—fill stack cache**

Format: CATCH src

Operation:

```
if (CSP == SP)
{
    while ((MSP < (CSP + src)) && ((MSP – SP)< SCSIZE))
    {
        stack_cache [MSP] = memory [MSP]
        stack_cache [MSP + 4] = memory [MSP + 4]
        stack_cache [MSP + 8] = memory [MSP + 8]
        stack_cache [MSP + 12] = memory [MSP + 12]
        MSP = MSP + 16
    }
}
```

Description:

If the CSP is SP, the stack cache is filled to the extent indicated by the source operand. The semantics of CATCH are somewhat different depending upon the address mode of src.

- If the source operand is defined with a stack offset mode (Roffset), the address is formed by adding the offset to the SP to determine the target value for the MSP (MSP = SP + offset).
- If the source operand is defined with an immediate mode ($data), the immediate value is used as the target for the MSP (MSP = data).
- If the source operand is defined with a stack offset indirect mode (*Roffset), the target value for the MSP is fetched from memory (or the stack cache) at the address formed by adding the offset to SP (MSP = *(offset + SP)).
- If the source operand is defined with an absolute mode ($addr), the target value for the MSP is fetched from memory (or the stack cache) at the address specified in the absolute address (MSP = *(addr)).

In no case will the MSP be incremented beyond the size of the on-chip stack cache. If the CSP is the ISP, CATCH is a no-op.

Encodings:

length	opcode	subcode	instruction	src
2	0x02	0x1	CATCH	stk8[a]
6	0x00	0x8	CATCH	word32

[a]. The 8-bit stack offset is zero-extended and multiplied by sixteen, providing an effective range of 0-4080 in quad-aligned increments.

Notes:

The MSP must be greater than or equal to the SP when CATCH executes; otherwise, instruction operation depends upon context and is therefore unpredictable.

If virtual addressing is enabled, and the MSP is updated, the new value is checked to verify that stores are valid at the current execution level. If the address is not valid, either a read fault, exception ID 0x8, or a MMU Table Walk Fault, exception ID 0xB, is flagged for CATCH.

Since the lower 4 bits of the SP do not exist, cache filling is done in 16-byte blocks. If the source operand to CATCH is not divisible by 16, the cache is filled to the next multiple of 16.

Name: **CLRE—clear PSW E- bit**

Format: CLRE

Description:

CLRE clears the PSW enter guard bit. The PSW enter guard bit is set by ENTER which has successfully completed execution.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0xA	CLRE

Name: **CMP—compare**

Format: CMPrel src1, src2

Operation: src1 *rel* src2 ? PSW.F = 1: PSW.F = 0

Description:

The program status word (PSW) flag bit is set to 1 if the comparison between the two source operands is true. If the comparison is false, the PSW flag bit is set to 0. *Rel* is one of the following:

EQ—equal to
GT—signed greater than
HI—higher (unsigned greater than)

Encodings:

length	opcode	instruction	src1	src2
2	0x10	CMPEQ	imm5	stk5
2	0x11	CMPGT	stk5	stk5
2	0x12	CMPGT	imm5	stk5
2	0x13	CMPEQ	stk5	stk5
6	0x1D	CMPGT	gen16	gen16
6	0x1E	CMPHI	gen16	gen16
6	0x1F	CMPEQ	gen16	gen16
10	0x1D	CMPGT	gen32	gen32
10	0x1E	CMPHI	gen32	gen32
10	0x1F	CMPEQ	gen32	gen32

Notes:

src1 is specified in the source operand field. src2 is specified in the destination operand field.

CMPEQ can test either = or ≠ , CMPGT can test signed >, ≥, <, ≤ and CMPHI can test unsigned >, ≥, <, ≤ . In the latter case, it is a matter of ordering the operands properly and testing the correct sense of the PSW flag bit.

Name: **CPU—register access escape**

Format: CPU

Description:

CPU is a prefix that changes the meaning of the instruction that follows the CPU instruction. Specifically, it changes the definition of address modes to enable access to the internal registers. All word-sized address modes remain the same, while mode 0x7 becomes the register addressing mode. The register number is stored in the operand field.

The low 4 bits of the operand are used as the register number; the high-order bits are ignored, but should be zero. Accessing the undefined register 0 results in an unimplemented instruction exception.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x0	CPU

Notes:

The instruction following CPU is considered part of the CPU instruction. If an exception or interrupt occurs, the program counter (PC) saved on the interrupt stack is the PC of the CPU instruction. In the prefetch and decode section of the ATT92010 *Hobbit* Microprocessor, the PC is incremented by four or six parcels, depending on whether the instruction following the CPU instruction is three or five parcels.

Caution:

The CPU is an interlocked instruction in that no other instruction is started until the CPU reaches the result register pipeline stage. It is possible to cause a *hazard* between instructions that modify the program status word (PSW) flag, carry, overflow, or enter guard bits. If either of the two instructions proceeding CPU modify the PSW flag, carry, overflow, or enter guard bits, any access of the PSW should be padded by two no-operation instructions (NOPs).

Name: **CRET—context return from kernel**

Format: CRET

Operation:

```
disable interrupts
SP = *(ISP + 0)           /* R0 wrt ISP */
fetch *(ISP + 4)
enable interrupts
CATCH (MSP - SP)
disable interrupts
MSP = *(ISP + 4)          /* R4 wrt ISP */
PC = *(ISP + 8)           /* R8 wrt ISP */
PSW = *(ISP + 12)         /* R12 wrt ISP */
ISP = ISP + 16
if (CSP == ISP)
    SHAD = ISP
else
    SHAD = SP
enable interrupts
```

Description:

A new stack pointer (SP) is loaded from the interrupt stack. The current contents of the stack cache are discarded and an unconditional CATCH is performed filling the stack cache to the maximum stack pointer (MSP). The program status word (PSW) and program counter (PC) values are restored by popping the interrupt stack.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x5	CRET

Notes:

The target MSP is fetched prior to the CATCH portion executing, but the MSP is not updated until the CATCH portion completes. Interrupts are disabled during a portion of CRET. Interrupts are enabled during the CATCH portion of CRET at the level of the restored PSW. The CATCH portion of CRET is performed consistently with the restored PSW current stack pointer and virtual/physical addressing mode bits.

If a memory fault occurs while reading from the interrupt stack, the ATT92010 *Hobbit* Microprocessor resets.

CRET is privileged. If CRET is initiated at the user level, a privilege exception is executed.

CRET cannot be traced.

If the location pointed to by the new PC value cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC value, not the address of CRET.

Name: **DIV—divide**

Format: DIV[3] src, dst

Operation:
DIV:
dst /= dst src
DIV3:
Acc = dst /src

Description:

The destination operand is divided by the source operand, and the quotient is placed in either the destination (DIV) or the accumulator (DIV3). Two's complement division is performed. See Section 3.3 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x27	DIV	gen16	gen16
6	0x37	DIV3	gen16	gen16
10	0x27	DIV	gen32	gen32
10	0x37	DIV3	gen32	gen32

Notes:

Division by zero results in a zero divide exception. Division of 0x80000000 by 0xFFFFFFFF sets the program status word (PSW) overflow bit and returns the result 0x80000000. The overflow bit is cleared in all other cases. The carry bit is unchanged in all cases.

Name: **DQM—double-word or quad-word move**

Format: DQM src, dst

Operation: dst = src

Description:

Double- or quad-word move moves either two or four contiguous words from the source to the destination. The size of the transfer is determined by the destination address mode field.

Double-word data size is encoded in the destination mode field as 0x0, 0x4, or 0x8. Quad-word data size is encoded in the destination mode field as 0xC, 0xD, or 0xE.

If the source mode is 0xF, the constant is replicated either two or four times depending upon the destination mode. If the destination mode is 0xF, an illegal instruction exception is taken. All other addressing modes result in an alignment fault.

Encodings:

length	opcode	instruction	src[a]	dst[a]
6	0x07	DQM	gen16	gen16
10	0x07	DQM	gen32	gen32

a. The limitations given in the description and note apply.

Notes:

Source and destination addresses of quad-word operands must be divisible by 16 (quad-aligned) and addresses of double-word operands must be divisible by 8 (double-aligned). Otherwise, an alignment exception occurs. Only word addressing modes are permitted for the source and the special modes for the destination. Other modes cause an illegal instruction sequence to occur.

Name: **ENTER—enter subroutine**

Format: ENTER src

Operation:

```
if (CSP == ISP)
{
  SHAD = ISP = target
}
if ((CSP == SP) && (src address mode!= stack offset))
{
  /*flush stack cache unconditionally*/
  while (MSP > SHAD)
  {
      memory[MSP – 16] = stack_cache[MSP – 16]
      memory[MSP – 12] = stack_cache[MSP – 12]
      memory[MSP – 8] = stack_cache[MSP – 8]
      memory[MSP – 4] = stack_cache[MSP – 4]
      MSP – =16
  }
  /*force stack cache to be empty*/
  SHAD = MSP = SP = target
}
if ((CSP == SP) && (src address mode == stack offset))
{
  /*flush only as much of the stack cache as is necessary*/
  if (MSP – target > SCSIZE)
  {
      while ((MSP   SHAD) && (MSP – taget > SCSIZE))
      {
          memory[MSP – 16] = stack_cache[MSP–16]
          memory[MSP – 12] = stack_cache[MSP – 12]
          memory[MSP – 8] = stack_cache[MSP – 8]
          memory[MSP – 4] = stack_cache[MSP – 4]
          MSP – =16
      }
      if (MSP > (target + SCSIZE))
          MSP = target + SCSIZE
  }
  SHAD = SP = target
}
PSW.E = 1
```

Description:

The CSP is altered either by adding the source operand (stack offset addressing mode) or replacing it with a new value (all other addressing modes). If the SP is not the CSP, no data traffic between the stack cache and memory is performed, and the MSP is not updated. If the SP is the CSP, the contents of the stack cache are written to memory (if necessary) in quad-word transfers until no more than SCSIZE bytes are held in the cache. The semantics of ENTER are somewhat different depending upon the address mode of src.

1. If the source operand is defined with a stack offset mode (Roffset), the address formed by adding the offset to the CSP is used to determine the target value (MSP = CSP + offset). The bounds of the stack cache are set to encompass the full amount of ENTER, within the limits of SCSIZE.
2. If the source operand is defined with an immediate mode ($data), the immediate value is used as the target value (MSP = data) and the stack cache is set empty at the completion of ENTER.
3. If the source operand is defined with a stack offset indirect mode (*Roffset), the target value is fetched from memory (or the stack cache) using the address formed by adding the offset to the CSP (MSP = *(offset + CSP)) and the stack cache is set empty at the completion of ENTER.
4. If the source operand is defined with an absolute mode (*$addr), the target value for the CSP is fetched from memory (or the stack cache) using the address specified in the absolute address (MSP = *(addr)) and the stack cache is set empty at the completion of ENTER.

Upon successful completion of ENTER, the PSW enter guard bit is set. This bit is cleared with CLRE.

Encodings:

length	opcode	subcode	instruction	src
2	0x02	0x0	ENTER	stk8[a]
6	0x00	0x9	ENTER	word32

a. The 8-bit stack offset is left padded with ones and multiplied by 16 giving it an effective range of -16 to -4096 in quad-aligned decrements.

Notes:

If the 6-byte form of ENTER is used with a stack offset mode for src, the magnitude of the offset must be greater than SCSIZE, and the offset must be less than or equal to 0, or unpredictable results may occur. The MSP must be greater than or equal to the SP when ENTER begins; otherwise, instruction operation depends upon context and, therefore, is unpredictable.

For the stack offset addressing model, only negative stack offsets are legal; positive stack offsets trigger an illegal instruction sequence. This includes ENTER R0.

If virtual addressing is enabled, the target address and the new MSP, if the MSP is updated, are checked to verify that stores are valid at the current execution level. If the addresses are not valid, a read fault exception, exception type 8, or MMU an MMU table walk fault, exception ID 0xB, is flagged for ENTER. The exception is processed after any stack flushing is completed. Since the lower 4 bits of the SP do not exist, the lower 4 bits of the source operand are ignored.

Name: **FLUSHD—flush data cache**

Format: FLUSHD

Description:

The data cache is flushed; all entries are marked invalid.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x6	FLUSHD

Notes:

Since there is no data cache, FLUSHD is not implemented in hardware. An unimplemented instruction sequence is taken.

Name: **FLUSHDCE—flush a data cache entry**

Format: FLUSHDCE src

Description:

The quad-word at src is flushed from the data cache.

Encodings:

length	opcode	subcode	instruction	src
6	0x00	0xD	FLUSHDCE	word32

Notes:

Since there is no data cache, FLUSHDCE is not implemented in hardware. An unimplemented instruction sequence is taken.

o

Name: **FLUSHI—flush decoded instruction cache**

Format: FLUSHI

Description:

The decoded instruction cache is flushed: all entries are marked invalid.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x3	FLUSHI

Name: **FLUSHP—flush prefetch buffer cache**

Format: FLUSHP

Description:

The prefetch buffer cache is flushed: all entries are marked invalid.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x4	FLUSHP

Name: **FLUSHPBE—flush a prefetch buffer entry**

Format: FLUSHPBE src

Description:

The quad-word at src is marked invalid in the PFB. No other caches are affected.

Encodings:

length	opcode	subcode	instruction	src
6	0x00	0xC	FLUSHPBE	word32

Name: **FLUSHPTE—flush a page table entry from the TLBs**

Format: FLUSHPTE src

Description:

If there is a page table entry for the address defined by src, in either the text or data TLBs, the entry is marked invalid. Both the text and data nonpaged segment registers are invalidated.

Encodings:

length	opcode	subcode	instruction	src
6	0x00	0xB	FLUSHPTE	word32

Notes:

For FLUSHPTE, the src operand is an address. Normally, the address is moved into the stack cache and the stack offset indirect addressing mode is used for src.

Name: **JMP—jump**

Format:

```
JMP dst
JMPT(Y|N) dst
JMPF(Y|N) dst
```

Operation:

```
JMP:
    PC = &dst
JMPT:
    if (PSW.F) PC = &dst
JMPF:
    if (!PSW.F) PC = &dst
```

Description:

The jump instructions cause the address of the destination operand to become the new PC value unconditionally (JMP), if the PSW flag bit is 1 (JMPT), or if the PSW flag bit is 0 (JMPF). A branch prediction bit is available for the conditional jumps to indicate that the jump more likely will (Y), or will not (N) be taken. Conditional jumps cannot use indirect addressing modes.

Encodings:

length	opcode	subcode	instruction	src(dst)
2	0x03	—	JMP	pcrel10
2	0x04	—	JMPFN	pcrel10
2	0x05	—	JMPFY	pcrel10
2	0x06	—	JMPTN	pcrel10
2	0x07	—	JMPTY	pcrel10
6	0x00	0x3	JMP	flow32
6	0x00	0x4	JMPFN	abs32
6	0x00	0x5	JMPFY	abs32
6	0x00	0x6	JMPTN	abs32
6	0x00	0x7	JMPTY	abs32

Notes:

If the location pointed to by the jump instruction cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC, not the PC of the original jump. The address of the original jump instruction is not saved. In the event of an indirect jump, if the ATT92010 *Hobbit* Microprocessor cannot reference the indirection word, a read-fault results and the PC stored on the interrupt stack is that of the indirect jump.

Name: **KCALL—kernel call**

Format: KCALL src

Operation:

```
disable interrupts
*(ISP – 12) = src                        /*R4 wrt new ISP*/
*(ISP – 8) = PC of next instruction      /*R8 wrt new ISP*/
*(ISP – 4) = PSW                         /*R12 wrt new ISP*/
ISP – =16
SHAD = ISP
PC = *(VB + 0)
PSW = PSW & 0xFFFF0000
enable interrupts
```

Description:

The PSW, PC (return point), and src operand values are saved on the interrupt stack as quad-words. The new PC value is read from the memory location pointed to by the vector base register. The low-order 16 bits of the PSW are set to 0, which selects kernel execution level, selects the ISP as the CSP, disables tracing, and inhibits interrupts. The PSW virtual physical addressing mode bit does not change.

Encodings:

length	opcode	subcode	instruction	src
2	0x00	—	KCALL	imm10*
6	0x00	0x0	KCALL	word32

* The 10-bit immediate value is zero-extended and multiplied by four giving it an effective range of 0 through 4092 in increments of 4.

Notes:

Interrupts are disabled while KCALL is processing. If a memory fault occurs while writing to the interrupt stack or reading from the table pointed to by the vector base, the *Hobbit* microprocessor resets.

If the location pointed to by the KCALL PC entry in the vector table cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the target PC (the value in the location pointed to by the VB), not the PC of the original KCALL instruction.

Name: **KRET—kernel return**

Format: KRET

Operation:

```
disable interrupts
PC =*(ISP + 8)                    /*R8 wrt ISP*/
PSW= *(ISP + 12)                  /*R12 wrt ISP*/
 ISP + =16
if (CSP == ISP)
        SHAD = ISP
else
        SHAD = SP
enable interrupts
```

Description:

The PSW and PC values are restored from the interrupt stack.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x1	KRET

Notes:

Interrupts are disabled while KRET is processing. If a memory fault occurs while reading from the interrupt stack the ATT92010 resets. KRET is privileged. If KRET is executed at the user level, a privilege exception is executed.

KRET cannot be traced.

If the location pointed to by the new PC value cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC value, not the address of KRET.

Name: **LDRAA—load relative address into accumulator**

Format: LDRAA dst

Operation: Acc = &dst

Description

The destination address is calculated as if a JMP instruction was being executed and stored in the accumulator.

Encodings:

length	opcode	subcode	instruction	src
6	0x00	0xA	LDRAA	flow32

Name: **MOV—move**

Format: MOV src, dst

Operation: dst = src

Description:

The value of the source operand is stored in the destination.

Encodings:

length	opcode	instruction	src	dst
2	0x0A	MOV	wai5,	stk5
2	0x18	MOV	stk5,	stk5
2	0x19	MOV	istk5,	stk5
2	0x1A	MOV	stk5,	istk5
2	0x1B	MOV	istk5,	istk5
2	0x1C	MOV	imm5,	stk5
6	0x06	MOV	gen16,	gen16
10	0x06	MOV	gen32,	gen32

Name: **MOVA—move address**

Format: MOVA src, dst

Operation: dst = &src

Description:

The address of the source operand is calculated and stored in the destination.

Encodings:

length	opcode	instruction	src	dst
2	0x1D	MOVA	stk5,	stk5
6	0x04	MOVA	gen16*,	gen16
10	0x04	MOVA	gen32*,	gen32

* The source operand must use a word addressing mode (that is modes ≥ 0xC) except for immediate as already noted. Any other mode causes an illegal instruction exception.

Notes:

If the size of the destination is byte or half-word, the calculated address is truncated (or sign-extended) to 8 or 16 bits. An immediate source operand as well as a register source or destination causes an illegal instruction exception.

Name:	**MUL—multiply**
Format:	MUL[3] src, dst
Operation:	MUL: dst * =src "unsigned overflow" ? PSW.C = 1 : PSW.C = 0 "signed overflow" ? PSW.V = 1 : PSW.V = 0 MUL3: Acc = dst * src "unsigned overflow" ? PSW.C = 1 : PSW.C = 0 "signed overflow" ? PSW.V = 1 : PSW.V = 0

Description:

The source operand is multiplied by the destination operand and the product is placed in either the destination (MUL) or the accumulator (MUL3). The PSW carry bit is set to 1 if the product of the operands as unsigned values overflows the destination (or accumulator); similarly, the PSW overflow bit is set to 1 if the product of the operands as signed values overflows the destination (or accumulator). Otherwise, the PSW carry and overflow bits are set to 0. See Section 3.3 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x26	MUL	gen16,	gen16
6	0x36	MUL3	gen16,	gen16
10	0x26	MUL	gen32,	gen32
10	0x36	MUL3	gen32,	gen32

Name: **NOP—no-operation**

Format: NOP

Description:

No operation is performed.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x2	NOP

Name: **OR—bitwise logical OR**

Format: OR[3] src, dst

Operation:

OR:
dst | = src

OR3:
Acc = dst | src

Description:

A bitwise logical OR is performed on the source and destination operands, and the result is placed in either the destination (OR) or the accumulator (OR3).

Encodings:

length	opcode	instruction	src	dst
6	0x21	OR	gen16,	gen16
6	0x31	OR3	gen16,	gen16
10	0x21	OR	gen32,	gen32
10	0x31	OR3	gen32,	gen32

Name: **ORI—bitwise logical OR interlocked**

Format: ORI src, dst

Operation:

```
hidden = dst
dst | = src
Acc = hidden
```

Description:

A bitwise logical OR operation is performed on the source and destination operands and the result is placed in the destination. $\overline{\text{LOCK}}$ is asserted during the fetch of dst if dst is in memory and not in the stack cache. $\overline{\text{LOCK}}$ is deasserted at the completion of the store to dst. No other accesses are done between the fetch and store of dst. The original value of dst is placed in the accumulator. If the accumulator is not in the stack cache, a store is made after the interlocked I/O completes.

Encodings:

length	opcode	instruction	src	dst
6	0x01	ORI	gen16,	gen16
10	0x01	ORI	gen32,	gen32

Notes:

Pipeline bypass hazards associated with semaphore operations are avoided in the *Hobbit* microprocessor by clearing the pipeline before an interlocked instruction enters the first pipeline stage. No other instruction is allowed into the pipeline until the executing interlocked instruction completes.

If R4 is the destination, after the interlocked instruction completes, R4 is the previous value of R4; hence, no operation is performed.

If the accumulator is not in the stack cache, CSP == MSP, an I/O access is made to update the accumulator after the interlocked accesses complete. The access to the accumulator must not fault in any manner, since ORI is not restartable from this point of the operation.

Name: **POPN—pop *n* entries from stack cache**

Format: POPN src

Operation:

```
disable interrupts
SHAD = CSP = CSP + src
if ((CSP == SP) && (CSP > MSP))
MSP SP
enable interrupts
```

Description:

The src operand is fetched, and added to the CSP and SHAD. If the CSP is SP, and the new SP value exceeds the MSP, the MSP is also updated to the new value. If the CSP is ISP, the MSP is not updated.

Encodings:

length	opcode	subcode	instruction	src
2	0x02	0x3	POPN	stk8*
6	0x00	0xF	POPN	stk32

* The 8-bit stack offset is zero extended and multiplied by 16 giving it an effective range of 0 through 4080 in quad-aligned increments.

Notes:

Only the stack offset addressing mode is legal; any other mode results in an illegal instruction exception sequence. Negative stack offsets are illegal.

Name: **REM—remainder**

Format: REM[3] src, dst

Operation:
REM:
dst % = src
REM3:
Acc = dst % src

Description:

The destination operand is divided by the source operand, and the remainder is placed in either the destination (REM) or the accumulator (REM3). Two's complement division is performed. See Section 3.3 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x25	REM	gen16,	gen16
6	0x35	REM3	gen16,	gen16
10	0x25	REM	gen32,	gen32
10	0x35	REM3	gen32,	gen32

Notes:

Division by zero results in a zero divide exception. The PSW overflow bit is always cleared by REM, and the carry bit is unchanged in all the cases.

Name: **RETURN—return from subroutine**

Format: RETURN src

Operation:

```
disable interrupts
PC = PC * (CSP + src)
SHAD = CSP = CSP + src
if ((CSP == SP) && (CSP > MSP))
MSP = SP
enable interrupts
```

Description:

The src operand is fetched and used as the new PC value. If the CSP is SP, and the new SP value exceeds the MSP, the MSP is also updated to the new value. If the CSP is ISP, the MSP is not updated.

Encodings:

length	opcode	subcode	instruction	src
2	0x02	0x2	RETURN	stk8*
6	0x00	0x2	RETURN	stk32

* The 8-bit stack offset is zero extended and multiplied by 16 giving it an effective range of 0 through 4080 in quad-aligned increments.

Notes:

Only the stack offset addressing mode is legal; any other mode results in an illegal instruction exception sequence. Even though the lower 4 bits of the SP do not exist, RETURN can obtain a new PC for a word-aligned register offset that is not a multiple of 16; but when adjusting the SP, the lower 4 bits of the offset are ignored. For example:

```
RETURN R4
```

obtains the new PC from R4, but the SP does not change. Similarly,

```
RETURN R20
```

obtains the new PC from R20, but the SP only increments 16. Only positive offsets are legal. Negative offsets result in an illegal instruction exception sequence. If the location pointed to by the new PC value cannot be referenced, a fetch-fault results. In this case, the PC stored on the interrupt stack is the new PC values, not the address of RETURN.

Name: **SHL—shift left**

Format: SHL[3] src, dst

Operation:
SHL:
dst << = Unsigned(src)
SHL3:
Acc = dst << Unsigned(src)

Description:

The destination operand is shifted left by the number of bits indicated by the source operand. Zeros replace the bits shifted out of the least significant bit of dst. Only the low-order 5 bits of src are used for the shift amount. The upper bits are ignored. For SHL3, the result is placed in the accumulator and the destination is left unchanged.

Encodings:

length	opcode	instruction	src	dst
2	0x1E	SHL3	uimm5,	stk5
6	0x2E	SHL	gen16,	gen16
6	0x3E	SHL3	gen16,	gen16
10	0x2E	SHL	gen32,	gen32
10	0X3E	SHL3	gen32,	gen32

Name: **SHR—arithmetic shift right**

Format: SHR[3] src, dst

Operation: SHR:
dst >> = src
HR3:
Acc = dst >> src

Description:

The destination operand is shifted right by the number of bits indicated by the source operand. The sign bit of the destination is copied as bits are shifted rightward. Only the low-order 5 bits of src are used for the shift amount. The upper bits are ignored. For SHR3, the result is placed in the accumulator and the destination is left unchanged.

Encodings:

length	opcode	instruction	src	dst
2	0x1F	SHR3	uimm5,	stk5
6	0x2C	SHR	gen16,	gen16
6	0x3C	SHR3	gen16,	gen16
10	0x2C	SHR	gen32,	gen32
10	0X3C	SHR3	gen32,	gen32

Name: **SUB—subtract**

Format: SUB[3] src, dst

Operation:

```
SUB:
    dst – = src
    "unsigned borrow" ? PSW.C = 1 : PSW.C = 0
    "signed borrow" ? PSW.V = 1 : PSW.V = 0
SUB3:
    Acc = dst – src
    "unsigned borrow" ? PSW.C = 1 : PSW.C = 0
    "signed borrow" ? PSW.V = 1 : PSW.V = 0
```

Description:

The source operand is subtracted from the destination operand, and the difference is placed in either the destination (SUB) or the accumulator (SUB3). The PSW carry bit is set on unsigned overflow and the PSW overflow bit is set on signed overflow; otherwise, the PSW carry and overflow bits are set to 0.

Encodings:

length	opcode	instruction	src	dst
6	0x20	SUB	gen16,	gen16
6	0x30	SUB3	gen16,	gen16
10	0x20	SUB	gen32,	gen32
10	0x30	SUB3	gen32,	gen32

Name: **TADD—tagged addition**

Format: TADD src, dst

Operation:

```
if ((src[1:0] != 0x0) || (dst[1:0] != 0x0))
    PSW.F = 1
else
    {
    dst + src
    "unsigned overflow" ? PSW.C = 1 : PSW.C = 0
    "signed overflow" ? PSW.V = 1 : PSW.V = 0
    PSW.F = PSW.V
    }
if (PSW.F == 0)
    dst = dst + src
```

Description:

The source operand is added to the destination operand, and the sum is placed in the destination if the PSW flag bit is set to 0. The PSW flag bit is set to 1 if the low 2 bits of either the source and destination operands are non zero or the PSW overflow bit is set to 1. The PSW carry bit is set to 1 on unsigned overflow, and the PSW overflow bit is set to 1 on signed overflow; otherwise, the PSW carry and overflow bits are set to 0. See Section 3.3 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x0C	TADD	gen16,	gen16
10	0x0C	TADD	gen32,	gen32

Name:	**TESTC—test PSW carry**
Format:	TESTC

Operation:	PSW.F = PSW.C
	PSW.C = 0

Description:

The PSW carry bit is copied into the PSW flag bit, and the PSW carry bit is set to 0.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x9	TESTC

Name: **TESTV—test PSW overflow**

Format: TESTV

Operation:

```
PSW.F = PSW.V
PSW.V = 0
```

Description:

The PSW overflow bit is copied into the PSW flag bit, and the PSW overflow bit is set to 0.

Encodings:

length	opcode	subcode	instruction
2	0x0B	0x8	TESTV

Name: **TSUB—tagged subtraction**

Format: TSUB src, dst

Operation:

```
if ((src[1:0] != 0x0) || (dst[1:0] != 0x0))
    PSW.F = 1
else
    {
    dst – src
    “unsigned borrow” ? PSW.C = 1 : PSW.C = 0
    “signed borrow” ? PSW.V = 1 : PSW.V = 0
    PSW.F = PSW.V
    }
if (PSW.F == 0)
    dst = dst – src
```

Description:

The source operand is subtracted from the destination operand, and the difference is placed in the destination if the PSW flag bit is set to 0. The PSW flag bit is set to 1 if the low 2 bits of either the source and destination operands are non-zero or the overflow bit is set to 1. The PSW carry bit is set to 1 on unsigned overflow, and the PSW overflow bit is set to 1 on signed overflow; otherwise, the PSW carry and overflow bits are set to 0. See Section 3.3 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x0D	TSUB	gen16,	gen16
10	0x0D	TSUB	gen32,	gen32

Name:	**UDIV—unsigned divide**
Format:	UDIV src, dst
Operation:	dst += dst src

Description:

The destination operand is divided by the source operand and the quotient is placed in the destination. Unsigned division is performed.

Encodings:

length	opcode	instruction	src	dst
6	0x2F	UDIV	gen16,	gen16
10	0x2F	UDIV	gen32,	gen32

Notes:

Division by zero results in a zero divide exception. The PSW carry bit is always cleared by UDIV, and the overflow bit is unchanged in all the cases.

Name: **UREM—unsigned remainder**

Format: UREM src, dst

Operation: dst % = src

Description:

The destination operand is divided by the source operand, and the remainder is placed in the destination. Unsigned division is performed. See Chapter 1 for a description of integer arithmetic.

Encodings:

length	opcode	instruction	src	dst
6	0x05	UREM	gen16,	gen16
10	0x05	UREM	gen32,	gen32

Notes:

Division by zero results in a zero divide exception. The PSW carry bit is always cleared by UDIV, and the PSW overflow bit is unchanged in all the cases.

Name: **USHR—unsigned shift right**

Format: SHR[3] src, dst

Operation:

```
USHR:
   dst >> = Unsigned(src)
USHR3:
   Acc = dst >> Unsigned(src)
```

Description:

The destination operand is shifted right by the number of bits indicated by the source operand. Zeros replace the bits shifted out of the most significant bit of destination operand. Only the low 5 bits of the source operand are used for the shift amount. The upper bits are ignored. For USHR3, the result is placed in the accumulator and the destination is left unchanged.

Encodings:

length	opcode	instruction	src	dst
6	0x2D	USHR	gen16,	gen16
6	0x3D	USHR3	gen16,	gen16
10	0x2D	USHR	gen32,	gen32
10	0x3D	USHR3	gen32,	gen32

Name: **XOR—bitwise logical exclusive OR**

Format: XOR[3] src, dst

Operation:

```
XOR:
    dst ^ = src
OR3:
    Acc = dst ^ src
```

Description:

A bitwise logical exclusive OR operation is performed on the source and destination operands, and the result is placed in either the destination (XOR) or the accumulator (XOR3).

Encodings:

length	opcode	instruction	src	dst
6	0x24	XOR	gen16,	gen16
6	0x34	XOR3	gen16,	gen16
10	0x24	XOR	gen32,	gen32
10	0x34	XOR3	gen32,	gen32

Chapter 4 Performance

The bottom line in performance is the length of time required to execute a program. This is usually factored into three areas:

- the total number of instructions required (for the program)
- the average number of cycles per instruction
- the clock rate

Because of the highly pipelined nature of the ATT92010 *Hobbit* Microprocessor, it is difficult to measure how long it takes to execute a single instruction. Many instructions can be executed at the rate of one per cycle, since pipelining allows the execution of instructions to overlap. This section provides performance data based on parameters detailed in the next section.

4.1 Execution Time

In order to describe the time it takes to execute an instruction, execution times are specified assuming the following conditions:

- Instruction fetches must access in the instruction cache
- Only the stack offset, immediate, register, absolute (for jumps and calls only), or program counter relative addressing modes are used
- All stack offset accesses are captured in the stack cache
- No data hazards occur between instructions

There are a some pipeline delays that may cause an instruction to take longer to execute. These delays (see section 4-2) should be added to the base execution time.

Throughout this section abbreviations are used, in tables, to represent performance cycles. These abbreviation are:

IC — decoded instruction cache

PDU — prefetch and decode unit

A — memory access time for a single word access

D — memory access time for a double word access

Q — memory access time for a quad word access

M — memory

N — number of valid entries in the stack cache

SC — stack cache

E — ENTER size

EU — Execution unit

For the purpose of estimating execution time, instructions fall into five groups:

- Simple
- Multi-Cycle Arithmetic
- DQM
- Conditional Jump
- Miscellaneous

Delays are also grouped into four types:

- Fetch and Empty Pipeline
- Operand Access
- Data Type
- Miscellaneous

Listings of execution times for each instruction group are shown in Table 4-1 through Table 4-5. Delays are listed in Table 4-6 through Table 4-9.

Table 4-32 Simple Instruction

Instruction	Min	Max	Instruction	Min	Max
ADD	1	1	OR	1	1
ADD3	1	1	OR3	1	1
ADD1	2	2	ORI	2	2
AND	1	1	RETURN	2	2
AND3	1	1	POPN	2	2
ANDI	2	2	SHL	1	1
CALL	1	1	SHL3	1	1
CLRE	1	1	SHR	1	1
CMP	1	1	SHR3	1	1
CPU	0	0	SUB	1	1
FLUSHI	1	1	SUB3	1	1
FLUSHP	1	1	TADD	1	1
FLUSHPBE	1	1	TESTC	1	1
FLUSHPTE	1	1	TESTV	1	1
JMP	0	1	TSUB	1	1
LDRAA	1	1	USHR	1	1
MOV	1	1	USHR3	1	1
MOVA	1	1	XOR	1	1
NOP	1	1	XOR3	1	1

Note that RETURNs are always indirect, do not add delay for indirection. If an unconditional jump is folded into the previous instruction, it takes no time to execute; otherwise, it takes one cycle.

The execution times for multi-cycle arithmetic instructions are data dependent. For these instructions, instruction fetch, operand access and data type delays are possible.

Table 4-33 Multi-Cycle Arithmetic Instructions

Instruction	Min	Max
DIV	38	38
DIV3	38	38
MUL	3	20
MUL3	3	20
REM	38	38
REM3	38	38
UDIV	38	38
UREM	38	38

For DQM type, instruction fetch delays are possible.

Table 4-34 DQM Instructions

Type of DQM	Cycles
Constant to SC double-word	3
Constant to SC quad-word	5
Constant to M double-word	1 + D
Constant to M quad-word	1 + Q
SC double-word to SC double-word	4
SC quad-word to SC quad-word	8
SC/M double-word to M/SC double-word	2 + D
SC/M quad-word to M/SC quad-word	4 + D
M double-word to M double-word	2 • D
M quad-word to M quad-word	8 • S

For conditional jump instructions, fetch delays are possible if the branch is not folded.

Table 4-35 Conditional Jump Instruction

Instruction	Cycles
Correct prediction, folded	0
Correct prediction, unfolded	1
Incorrect prediction, jump after compare	3
Incorrect prediction, jump 2 instructions after compare	2
Incorrect prediction, jump 3 instructions after compare	1
Incorrect prediction, unfolded, 4 or more instructions after compare	1
Incorrect prediction, folded, 4 or more instructions after compare	0

For the remaining miscellaneous instructions, instruction fetch and miscellaneous delays are possible.

Table 4-36 Miscellaneous Instruction

Instruction	Cycles
CATCH	1
CRET	11 + Q
ENTER	1 + (Q • E)
KCALL	8 + D + A
KRET	10 + D
Unimplemented opcode	7 + A
Exception	9 + D + A

4.2 Delays

Instruction fetch delays occur when the instruction is not immediately available for execution by the execution unit (EU). The instruction misses the IC and the EU reset the PDU to fetch the desired instruction. If the EU is using I/O, the PDU is accomplishing a unrelated memory access at the time of reset, or if the PDU is handling a previously received fault, the delays may be extended. Table 4-6 lists the Fetch and Empty Pipeline Delays.

Table 4-37 Fetch and Empty Pipeline Delays

Condition	Penalty
IC miss, instruction contained in prefetch buffer	3
IC miss, instruction contained in single double-word in memory	5 + D
IC miss, instruction contained in single quad-word memory	5 + 2D
IC miss, instruction contained in 2 quad-words in memory	8 + 2D
IC miss and instruction is a CPU-prefix operation	1
EU pipeline empty	2

Operand accesses may also take longer than predicted (using these tables) because of the possibility of a data hazard or internal contention for I/O. Data hazards occur when a previous instructions tires to write to a memory location that overlaps the location being read by a subsequent instruction.

I/O contention occurs when the EU wants to make an external memory access while the PDU is in the middle of an access. The Operand Access Delays are listed in Table 4-7 followed by the Data Type and Miscellaneous Delay tables.

Table 4-38 Operand Access Delays

Condition	Penalty
One operand in memory	1 + A
Two operands in memory	1 + 2A
One or two operands indirect, both pointers in stack cache	1
One or two operands indirect, one pointer in memory	1 + A
Two operands indirect, both pointers in memory	1 + 2A
Destination in memory	A

Table 4-39 Data Type Delays

Condition	Penalty
One operand not word type	1
Two operands not word type	2

Table 4-40 Miscellaneous Delays

Condition	Penalty
One or two operands indirect, both pointers in stack cache	1
One or two operands indirect, one pointer in memory	1 + A
Two operands indirect, both pointers in memory	1 +2A

4.3 Branch Folding

The ATT92010 *Hobbit* Microprocessor provides a *next address* field with each decoded instruction. When the PDU detects a non-branching operation followed by a branch, it *folds* the two instructions to form a single instruction/branch operation.

As a result, branches are rarely explicitly executed because they are folded and executed along with other instructions. A one-parcel branch will fold into a previous one- or three-parcel instruction and execute together except when the previous instruction is:

- another jump of any kind
- any one-parcel instruction with a five-bit opcode in the range 00000->00111
- any three-parcel monadic instruction (for example, opcode equals 000000)

Appendix A Bus Arbitration and Electrical Characteristics

To facilitate multiple bus masters, the bus arbitration protocol does not make the ATT92010 the default bus master. A centralized arbiter selects the current bus master and controls transactions over the bus. A synchronous bus protocol is used to exchange ownership of the bus from one master to another. The central bus arbiter must execute this protocol, asserting and negating $\overline{\text{BGRANT}}$ to the various bus masters in a consistent manner.

A.1 Bus Protocol

The signals involved in this protocol generated by the central bus arbiter are $\overline{\text{HRESET}}$, $\overline{\text{BGRANT}}$, and $\overline{\text{RETRY}}$. There is a $\overline{\text{BGRANT}}$ for each bus master, with the other signals shared among bus masters.

The signals involved in this protocol generated by the bus masters are $\overline{\text{BREQ}}$, $\overline{\text{START}}$, IOC[1:0], and $\overline{\text{LOCK}}$. There is a $\overline{\text{BREQ}}$ for each bus master, with the other signals shared among bus masters. Finally, the device being accessed generates $\overline{\text{DTACK}}$.

Upon reset of the system, which must be synchronous, the arbiter selects one of the requesting bus masters as current bus master by asserting its $\overline{\text{BGRANT}}$. Having received $\overline{\text{BGRANT}}$, the master takes ownership of the bus. The bus arbiter monitors the bus, keeping track of the state of the bus. The asserts $\overline{\text{BREQ}}$ when an I/O transaction is pending (upon reset, the ATT92010 *Hobbit* Microprocessors want to start execution at address 0x0).

The arbiter selects a new bus master by deasserting $\overline{\text{BGRANT}}$ to the current bus master and asserting $\overline{\text{BGRANT}}$ to the next bus master at the end of any outstanding bus transactions. If the current bus master loses $\overline{\text{BGRANT}}$ with an outstanding transaction on the bus, that master $\overline{\text{LOCK}}$ remains on the bus until $\overline{\text{DTACK}}$ is asserted with IOC[1:0] equal to zero and $\overline{\text{LOCK}}$ is deasserted.

The new bus master takes ownership of the bus at the beginning of the next bus cycle after receipt of $\overline{\text{BGRANT}}$. The arbiter must assert $\overline{\text{BGRANT}}$ in a manner which inserts a dead cycle between the end of the previous bus owner's $\overline{\text{BGRANT}}$ and the beginning of the next bus owner's $\overline{\text{BGRANT}}$.

The ATTT92010 asserts $\overline{\text{BGACK}}$ to indicate that it has bus ownership, and it deasserts$\overline{\text{BGACK}}$ to indicate that it has relinquished the bus. In Figure A-1, bus cycles 1 through 4 show a typical bus request and acquisition.

A.2 Surrendering the Bus

The arbiter signals the ATT92010 *Hobbit* microprocessor to relinquish the bus by deasserting $\overline{\text{BGRANT}}$. When $\overline{\text{BGRANT}}$ is deasserted, the ATT92010 will relinquish ownership of the bus and deassert $\overline{\text{BGACK}}$. If the microprocessor is running a bus transaction and $\overline{\text{BGRANT}}$is deasserted, ownership of the bus will be relinquished after receipt of $\overline{\text{DTACK}}$ with IOC[1:0] equal to zero and $\overline{\text{LOCK}}$ is deasserted.

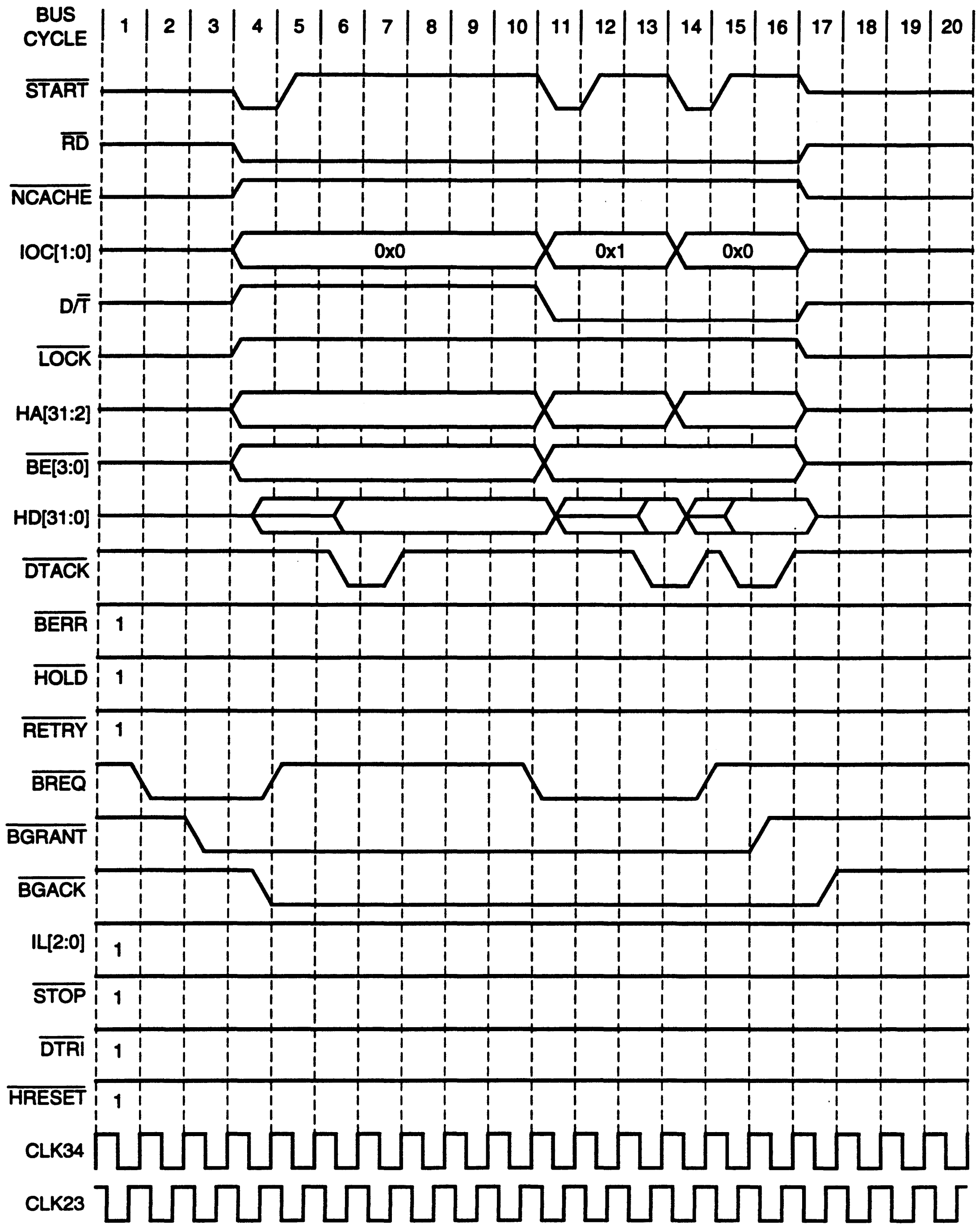
If the microprocessor is not running a bus transaction and $\overline{\text{BGRANT}}$ is deasserted, ownership of the bus will be relinquished at the beginning of the next bus cycle. $\overline{\text{BGACK}}$ is deasserted by the microprocessor in the same bus cycle that ownership of the bus is being relinquished.

Most arbitration protocols will want to continue to grant the bus to the current bus master if it continues to request the bus by asserting its $\overline{\text{BREQ}}$.

In Figure A-1, bus cycles 15 through 17 show a typical release of the bus.

Figure A-1 represents a cacheable single-word data read followed by a double-word text read. The accesses are not interlocked and don't produce bus errors.

Figure A-2 Read BUS Cycles with BUS Arbitration



A.3 Bus Transaction Types

Normal bus transfers begin with the assertion of $\overline{\text{START}}$ and end with the assertion of $\overline{\text{DTACK}}$. In case of an error during a bus transfer, the transaction may be ended by the assertion of $\overline{\text{HRESET}}$ or $\overline{\text{BERR}}$ with $\overline{\text{DTACK}}$. Interlocked bus transfers end with the deassertion of $\overline{\text{LOCK}}$ following a $\overline{\text{DTACK}}$. Multiple word transfers end when IOC[1:0] = 0 with assertion of $\overline{\text{DTACK}}$. Sub-word accesses are the same as single-word accesses with the exception that only the appropriate byte enables are asserted.

A .3 .1 Read Transactions

Read transactions may fetch text or data. Text reads are always double-word transfers. Data reads are either single-, double- or quad-word transfers. After completion of a read transaction, a loopback is performed if the microprocessor remains owner of the bus and there are no pending bus transactions. See Figure A-1 for the following example.

Bus cycles 4 through 6 show a typical read transaction. In bus cycles 7 through 10, a loopback cycle is performed. In bus cycles 11 and 15, a double-word transaction is performed. In bus cycle 16, another loopback cycle is performed. The ATT92010 holds all bus signals at their previous values and loops back the data read on the previous cycle.

Note The bus transaction may be ended by $\overline{\text{HRESET}}$ or $\overline{\text{BERR}}$ with $\overline{\text{DTACK}}$ to signal an error.

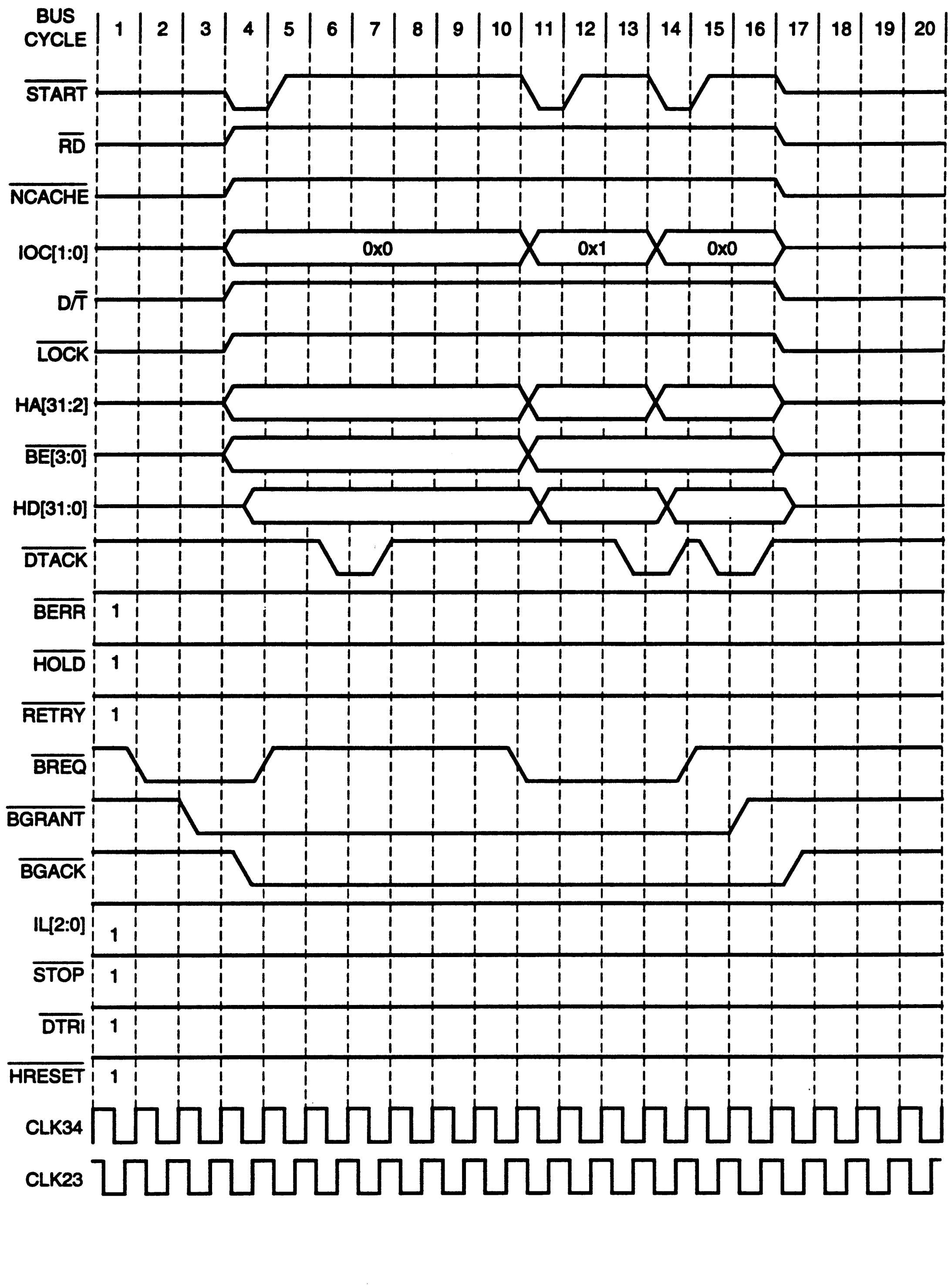
A .3 .2 Write Transactions

Write transactions are either single-, double-, or quad-word transfers. Refer to Figure A-2 for the following example.

Bus cycles 4 through 7 show a typical write transaction. In bus cycles 8 through 10, the microprocessor maintains the previous bus cycles values on most signals. In bus cycles 11 through 13 and bus cycles 14 through 15, two more write transactions are performed.

Note The bus transaction may be ended by $\overline{\text{HRESET}}$ or $\overline{\text{BERR}}$ with $\overline{\text{DTACK}}$ to signal an error.

Figure A-3 Write BUS Cycles with BUS Arbitration



A .3 .3 Interlocked Bus Transfer

This is a read-modify-write type bus operation. This sequence of operations is not interruptible. The bus remains locked through the write. If $\overline{\text{BGRANT}}$ is de-asserted during an interlocked operation, the operation is completed and transfer of bus ownership is delayed a clock cycle. Refer to Figure A-3 for the following example.

Bus cycles 2 through 4 show the read portion of the RMW operation. Bus cycles 7 through 8 show the write portion of the RMW operation. In bus cycle 9, $\overline{\text{LOCK}}$ remains asserted by the microprocessor adding a dead cycle. In bus cycle 11, the next bus cycle begins.

A .3 .4 Block Data Transfer

The block transfer sizes that are supported are double- and quad-word. The block transfer looks like a series of single-word bus transfers with the microprocessor incrementing address bits HA[3:2] and decrementing IOC[1:0] for each access. Block transfers are not interruptible. Block transfers may be retried with the transfer resuming where it was aborted when $\overline{\text{RETRY}}$ is deasserted.

A.4 Exception Handling

The exception/error signals provide a means by which external devices can inform the ATT92010 of an unusual condition which requires the processor to deviate from its normal execution.

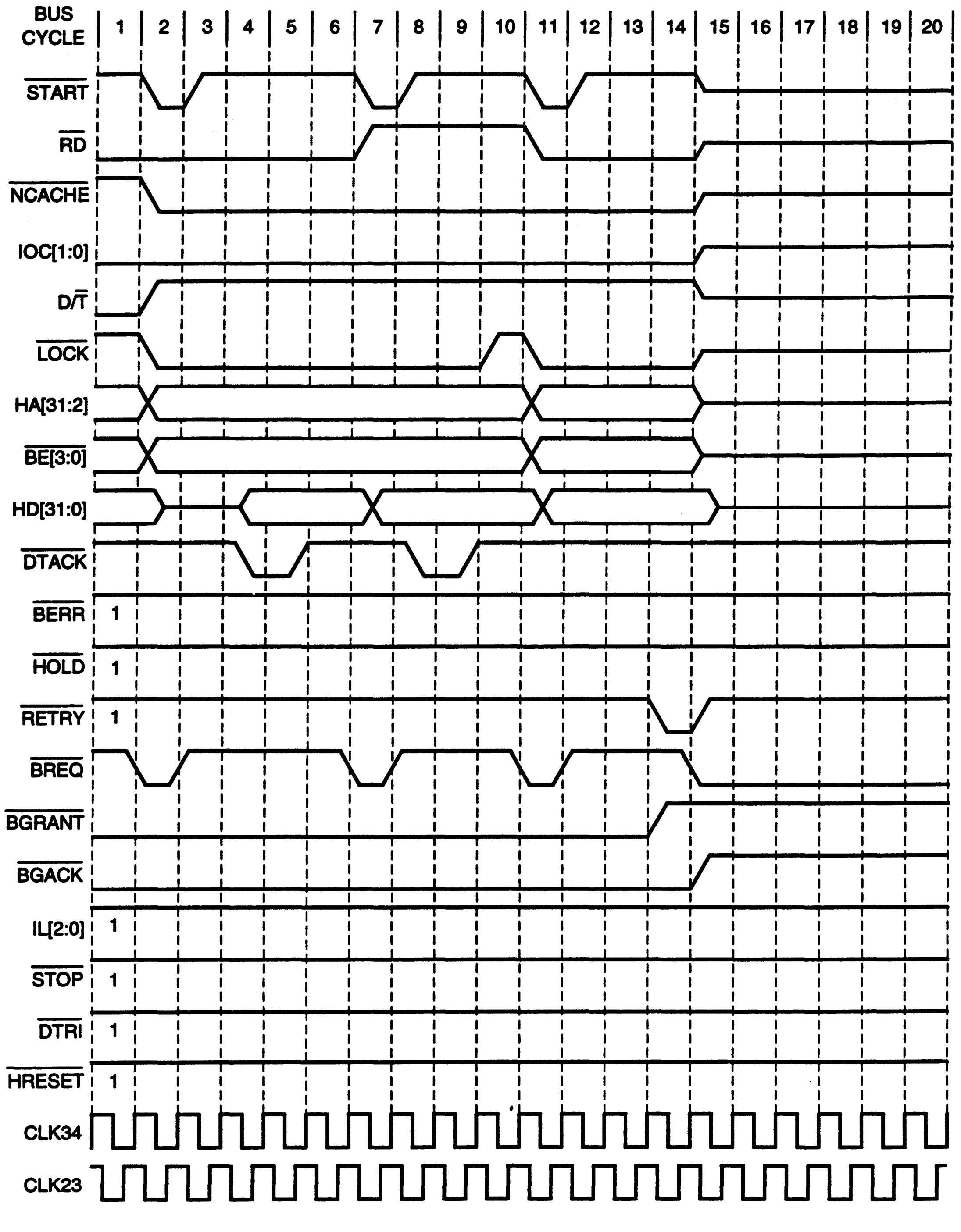
A .4 .1 Bus Retry

$\overline{\text{RETRY}}$ is asserted to retry the current bus transaction. When $\overline{\text{RETRY}}$ is asserted during a valid bus transaction, the ATT92010 *Hobbit* Microprocessor aborts the current bus transfer and masks the $\overline{\text{DTACK}}$ input. After $\overline{\text{RETRY}}$ is deasserted, the bus transaction is rerun after the microprocessor obtains ownership of the bus as $\overline{\text{RETRY}}$ is orthogonal to bus arbitration. In systems with gateways through which two buses communicate with each other, the retry feature is required to break deadlock conditions when the two buses have simultaneous requests for their respective counterpart bus.

A .4 .2 Bus Error

The assertion of $\overline{\text{BERR}}$ indicates an error in a bus transaction of any type. An internal I/O fault is generated when $\overline{\text{BERR}}$ is asserted and a $\overline{\text{DTACK}}$ is received. When $\overline{\text{BERR}}$ is asserted and $\overline{\text{DTACK}}$ received, the exception taken depends upon the type of bus transaction being terminated.

Figure A-4 Interlocked BUS Transfer with and without Entry



A .4 .3 Reset

If $\overline{\text{HRESET}}$ is asserted, the ATT92010 *Hobbit* Microprocessor is reset and any current bus cycle is aborted.

Signal	Priority Level
$\overline{\text{HRESET}}$	Highest
$\overline{\text{RETRY}}$	Ø
$\overline{\text{DTACK}}$	Lowest

Appendix B Electrical Characteristics

This appendix presents reference information on the electrical characteristics of the ATT92010 *Hobbit* Microprocessor. This information is presented in table format as well as diagrams.

B.1 Absolute Maximum Ratings

Stresses in excess of the Absolute Maximum Ratings can cause permanent damage to the device. These are absolute stress ratings only. Functional operation of the device is not implied at these or any other conditions in excess of those given in the operational sections of the data sheet. Exposure to Absolute Maximum Ratings for extended periods can adversely affect device reliability.

Parameter	Symbol	Min	Max	Unit
Idc Supply Voltage	V_{DD}	–0.5	7.0	V
Ambient Operating Temperature	T_A	0	70	C
Storage Temperature	T_{stg}	–40	125	C

B.2 Handling Precautions

All MOS devices must be handled with certain precautions to avoid damage due to the accumulation of static charge. Although input protection circuitry has been incorporated to minimize the effect of statics buildup, proper cautions should be taken to avoid exposure to electrostatic discharge (ESD) during handling and mounting. AT&T employs a human-body model (HBM) for ESD susceptibility testing.

Since the failure voltage of electrostatic devices is dependent on the current and voltage and, hence, the resistance and capacitance, it is important that standard values be employed to establish a reference by which to compare test data. Values of 100 pF and 1500Ω are the most common and are the values used in the AT&T HBM test circuit. The breakdown voltage for the ATT92010 *Hobbit* Microprocessor is 1,000V, according to the HBM, and it is 2,000V according to the charged-device model (CDM).

Table B-1 20 MHz Recommended Operating Conditions

(V_{DD} = 3.3 V 10%; CLK34 and CLK23 = 20 MHz)

Parameter	Symbol	Min	Typ	Max	Unit
Input High Voltage	V_{IH}	2.2	—	V_{DD} + 0.3	V
Input Low Voltage	V_{IL}	–0.3	—	0.6	V
Output High Voltage I_{OH} = 5.7 mA (pins HD[31:0]) I_{OH} = 2.5 mA (all pins except HD[31:0])	V_{OH}	2.5	—	—	V
Output Low Voltage I_{OL} = 5.7 mA (pins HD[31:0]) I_{OL} = 2.5 mA (all pins except HD[31:0])	V_{OL}	—	—	0.3	V
TDI Input Low Current	I_{TDI}	—	—	–1.73	mA
TMS Input Low Current	I_{TMS}	—	—	–0.87	mA
TCK Input Low Current	I_{TCK}	—	—	–0.87	mA
$\overline{\text{TRST}}$ Input High Current	I_{TRST}	—	—	–1.16	mA
Input Leakage Current 0 V ≤ V_{IN} ≤ V_{DD}	I_I	–0.01	—	0.01	mA
3-Stated Output Leakage Current	I_{OTI}	–0.01	—	0.01	mA
Supply Current Output Load = 10 pF Output Load = 50 pF	 I_{DD} I_{DD}	 — —	 75 125	 95 150	 mA mA
Standby Current	I_{SB}	0	6	30	uA

Table B-2 30 MHz Recommended Operating Conditions

(V_{DD} = 5.0 V ±10%; CLK34 and CLK23 = 30 MHz)

Parameter	Symbol	Min	Typ	Max	Unit
Input High Voltage	V_{IH}	3.2	—	V_{DD} + 0.4	V
Input Low Voltage	V_{IL}	–0.4	—	0.8	V
Output High Voltage I_{OH} = 8 mA (pins HD[31:0]) I_{OH} = 3.5 mA (all pins except HD[31:0])	V_{OH}	3.6	—	—	V
Output Low Voltage I_{OL} = 8 mA (pins HD[31:0]) I_{OL} = 3.5 mA (all pins except HD[31:0])	V_{OL}	—	—	0.4	V
TDI Input Low Current	I_{TDI}	—	—	–2.63	mA
TMS Input Low Current	I_{TMS}	—	—	–1.31	mA
TCK Input Low Current	I_{TCK}	—	—	–1.31	mA
$\overline{\text{TRST}}$ Input High Current	I_{TRST}	—	—	–1.75	mA
Input Leakage Current 0 V ≤ V_{IN} ≤ V_{DD}	I_I	–0.01	—	0.01	mA
3-Stated Output Leakage Current	I_{OTI}	–0.01	—	0.01	mA
Supply Current Output Load = 10 pF Output Load = 50 pF	 I_{DD} I_{DD}	 — —	 175 285	 220 340	 mA mA
Standby Current	I_{SB}	—	9	50	uA

All timing is based on a 70 pF load under worst-case conditions, although the device is capable of driving heavier loads.

Table B-3 Test Loading and Output Derating Factors

Output Signal	Max Load (pF)	Test Load (pF)	Output Derating (ns/pF	
			VDD=3.3V ±10%	VDD=5.0V ±10%
HA[31:2]	100	50	0.09	0.06
$\overline{\text{BGACK}}$	100	50	0.09	0.06
$\overline{\text{BE[3:0]}}$	100	50	0.09	0.06
$\overline{\text{BREQ}}$	100	50	0.09	0.06
D/$\overline{\text{T}}$	100	50	0.09	0.06
IOC[1:0]	100	50	0.09	0.06
$\overline{\text{LOCK}}$	100	50	0.09	0.06
$\overline{\text{NCACHE}}$	100	50	0.09	0.06
$\overline{\text{START}}$	100	50	0.09	0.06
TDO	100	50	0.09	0.06
$\overline{\text{RD}}$	100	50	0.09	0.06
HD[31:0]	150	50	0.06	0.04

The *Output Derating* factor shown above is used to obtain an approximate increase rate of output valid delay time with increasing load capacitance up to the maximum loading specified.

Clock

Two 1x clocks in quadrature are required by the ATT92010 *Hobbit* Microprocessor. The internal clocks are decoded from these inputs. The internal clocks can be stopped in phase 1 by asserting $\overline{\text{STOP}}$ prior to phase 1 allowing for burst-mode, single-stepping, and suspended operation

Figure B-1 Clock Input Timing

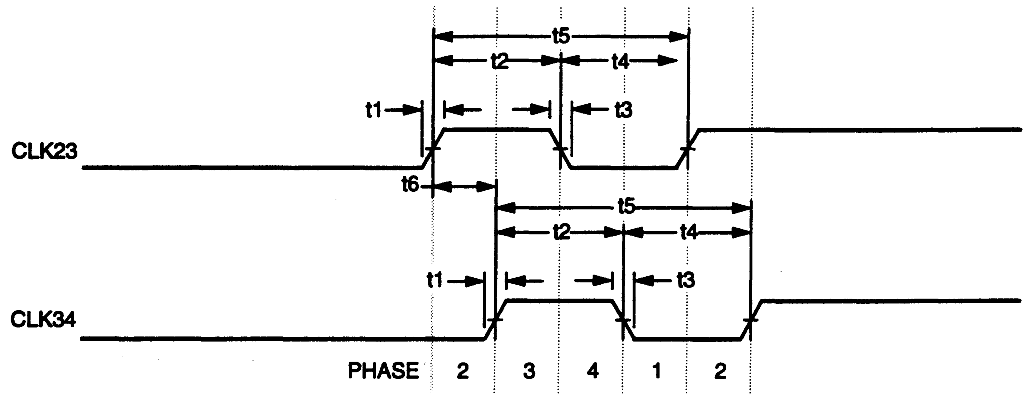


Table B-4 Clock Input Timing Table

Symbol	Parameter	3.3 V		5.0 V		Unit
		Min	Max	Min	Max	
t1	Rise Time	—	3.0	—	3.0	ns
t2	Pulse High	22.5	27.5	14.5	18.5	ns
t3	Fall Time	—	3.0	—	3.0	ns
t4	Pulse Low	22.5	27.5	14.5	18.5	ns
t5	Period	50.0	100	33.2	50.0	ns
t6	Delay	10.5	14.5	6.3	10.2	ns

Figure B-2 Synchronous Input Timing

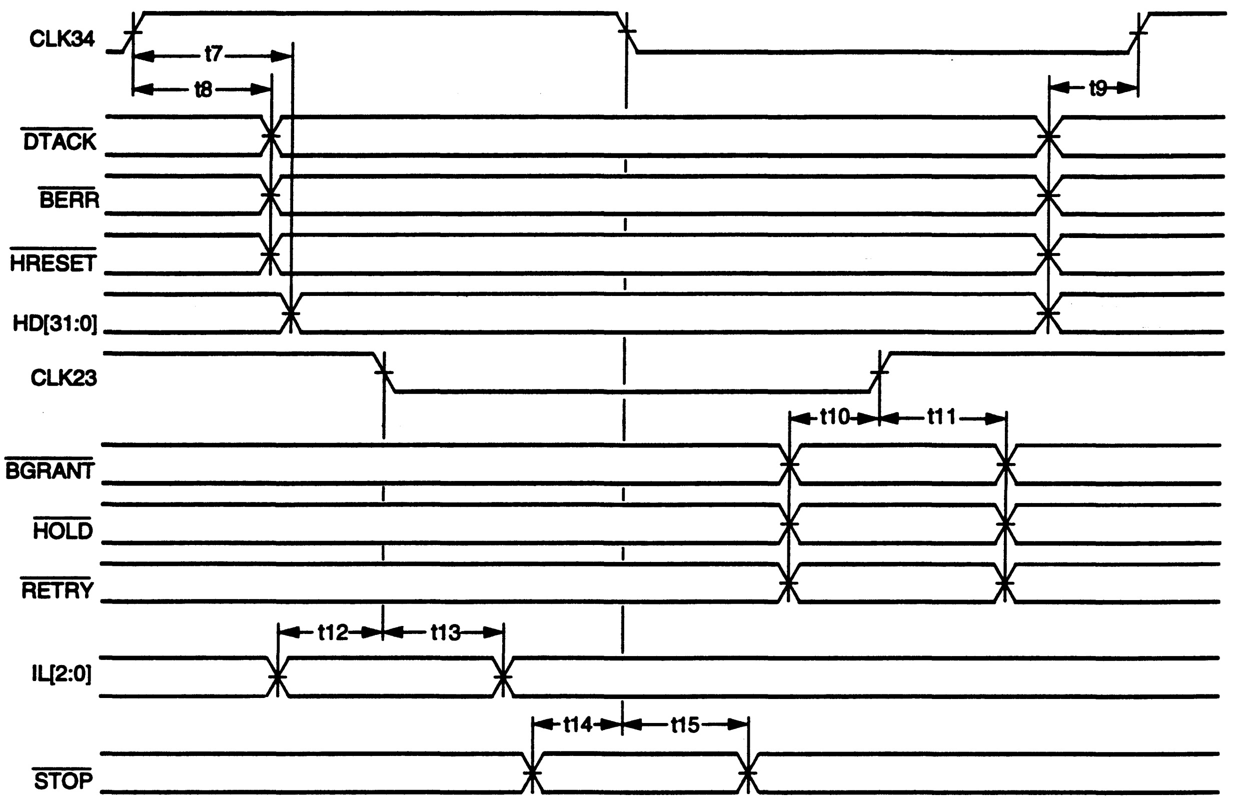


Table B-5 Synchronous Input Timing Table

Symbol	Signal	Type	Reference	3.3 V		5.0 V		Unit
				Min	Max	Min	Max	
t7	HD[31:0]	Input Hold	CLK34 Rise	7	—	6	—	ns
t8	$\overline{\text{DTACK}}$	Input Hold	CLK34 Rise	5	—	4	—	ns
	$\overline{\text{BERR}}$	Input Hold	CLK34 Rise	5	—	4	—	ns
	$\overline{\text{HRESET}}$	Input Hold	CLK34 Rise	5	—	4	—	ns
t9	HD[31:0]	Input Setup	CLK34 Rise	3	—	2	—	ns
	$\overline{\text{DTACK}}$	Input Setup	CLK34 Rise	3	—	2	—	ns
	$\overline{\text{BERR}}$	Input Setup	CLK34 Rise	3	—	2	—	ns
	$\overline{\text{HRESET}}$	Input Setup	CLK34 Rise	3	—	2	—	ns
t10	$\overline{\text{BGRANT}}$	Input Setup	CLK23 Rise	3	—	2	—	ns
	$\overline{\text{HOLD}}$	Input Setup	CLK23 Rise	3	—	2	—	ns
	$\overline{\text{RETRY}}$	Input Setup	CLK23 Rise	3	—	2	—	ns
t11	$\overline{\text{BGRANT}}$	Input Hold	CLK23 Rise	5	—	4	—	ns
	$\overline{\text{HOLD}}$	Input Hold	CLK23 Rise	5	—	4	—	ns
	$\overline{\text{RETRY}}$	Input Hold	CLK23 Rise	5	—	4	—	ns
t12	IL[2:0]	Input Setup	CLK23 Fall	3	—	2	—	ns
t13	IL[2:0]	Input Hold	CLK23 Fall	5	—	4	—	ns
t14	$\overline{\text{STOP}}$	Input Setup	CLK34 Fall	5	—	4	—	ns
t15	$\overline{\text{STOP}}$	Input Hold	CLK34 Fall	3	—	2	—	ns

Figure B-3 Output Timing

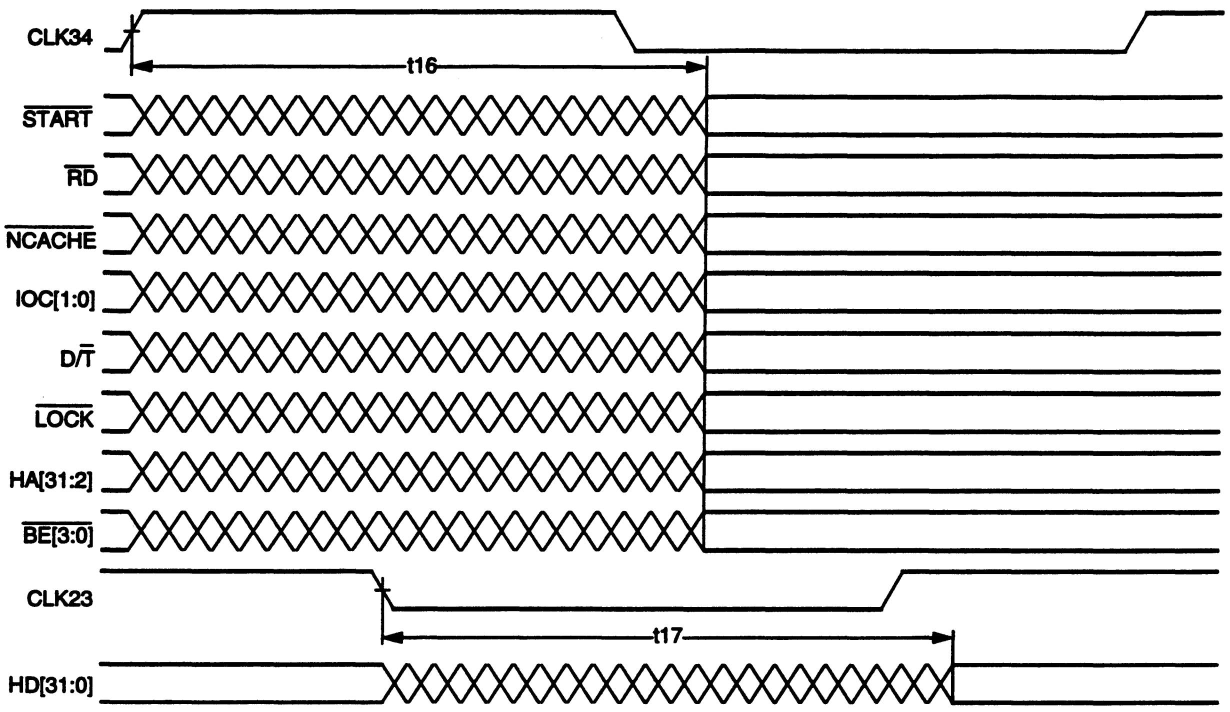


Table B-6 Output Timing Table

Symbol	Signal	Type	Reference	3.3 V		5.0 V		Unit
				Min	Max	Min	Max	
t16	$\overline{\text{START}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
	$\overline{\text{RD}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
	$\overline{\text{NCACHE}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
	IOC[1:0]	Output Valid	CLK34 Rise	—	23	—	18	ns
	D/$\overline{\text{T}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
	$\overline{\text{LOCK}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
	HA[31:2]	Output Valid	CLK34 Rise	—	19	—	15	ns
	$\overline{\text{BE[3:0]}}$	Output Valid	CLK34 Rise	—	23	—	18	ns
t17	HD[31:0]	Output Valid	CLK23 Fall	—	24	—	19	ns

Figure B-4 BUS Relinquish Cycle Output Timing

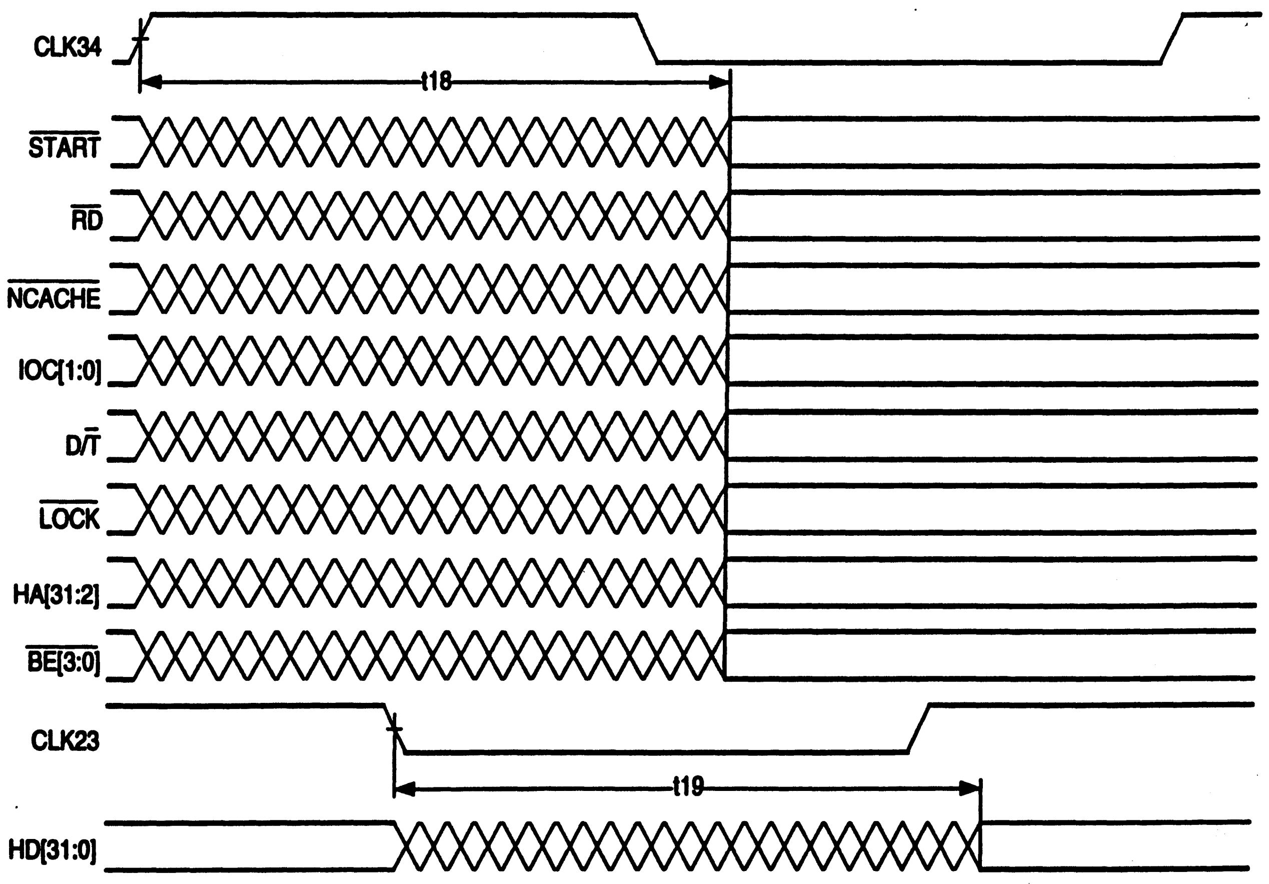


Table B-7 BUS Relinquish Cycle Output Timing Table

Symbol	Signal	Type	Reference	3.3 V		5.0 V		Unit
				Min	Max	Min	Max	
t18	START	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	$\overline{\text{RD}}$	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	$\overline{\text{NCACHE}}$	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	IOC[1:0]	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	D/$\overline{\text{T}}$	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	$\overline{\text{LOCK}}$	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	HA[31:2]	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
	$\overline{\text{BE[3:0]}}$	Output Hi-Z	CLK34 Rise	—	26	—	22	ns
t19	HD[31:0]	Output Hi-Z	CLK23 Fall	—	26	—	22	ns

Figure B-5 $\overline{\text{DTRI}}$ TO Data Output Timing

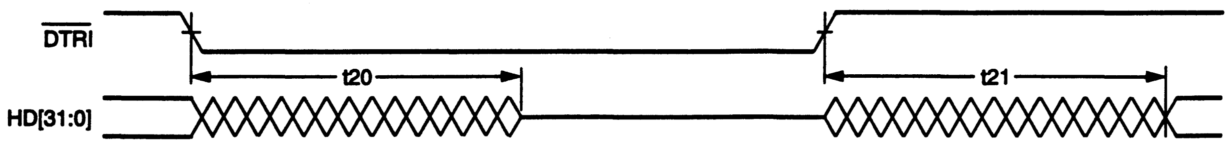


Table B-8 $\overline{\text{DTRI}}$ to Data Output Timing at 50 pF Loading Table

Symbol	Signal	Type	Reference	3.3 V		5.0 V		Unit
				Min	Max	Min	Max	
t20	HD[31:0]	Output Hi-Z	$\overline{\text{DTRI}}$ Fall	—	26	—	22	ns
t21	HD[31:0]	Output Valid	$\overline{\text{DTRI}}$ Rise	—	24	—	19	ns

Figure B-6 $\overline{\text{BREQ}}$ and $\overline{\text{BGACK}}$ Timing Diagram

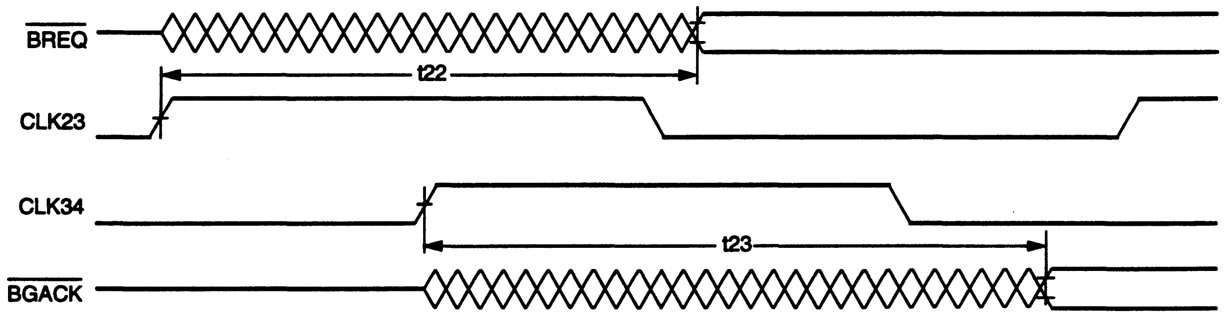


Table B-9 $\overline{\text{BREQ}}$ and $\overline{\text{BGACK}}$ Output Timing at 50 pF Loading Table

Symbol	Signal	Type	Reference	3.3 V		5.0 V	
				Min	Max	Min	Max
t22	$\overline{\text{BREQ}}$	Output Valid	CLK23 Rise	—	23	—	18
t23	$\overline{\text{BGACK}}$	Output Valid	CLK34 Rise	—	32	—	25

Table B-10 JTAG BUS Timing Specifications

Signal	Type	Reference	3.3 V		5.0 V		Unit
			Min	Max	Min	Max	
TCK	Period	—	400.0	—	200.0	—	ns
TCK	Pulse High	—	200.0	—	100.0	—	ns
TCK	Pulse Low	—	200.0	—	100.0	—	ns
TDI	Input Setup	TCK rise	50.0	—	25.0	—	ns
TDI	Input Hold	TCK rise	50.0	—	25.0	—	ns
TMS	Input Setup	TCK rise	50.0	—	25.0	—	ns
TMS	Input Hold	TCK rise	50.0	—	25.0	—	ns
TDO	Output Valid	TCK fall	—	100.0	—	50.0	ns
TDO	Output Hi-Z	TCK fall	—	100.0	—	50.0	ns

Appendix C Testability

The ATT92010 *Hobbit* Microprocessor is a highly testable design providing access to all testability features via the IEEE 1149.1/D5 interface. These features are:

- Single clock delay by-pass
- Boundary-scan of I/O signals
- Embedded memory built-in test (BIT) and scan features
- Embedded programmable logic array (PLA) built-in test features

C.1 Test Access Port (TAP)

The test access port (TAP) consists of five I/O pins and a sequential 16 state controller. The signals in the TAP are defined as follows.

TCK	Test Clock—Input—An externally gated clock signal with a 50% duty cycle. The changes on the TAP input signals (TMS and TDI) are clocked into the TAP controller, instruction register or selected test data register on the rising edge of TCK. Changes at the TAP output signal (TDO) occur on the falling edge of TCK. This signal does not conform to IEE 1149.1/D5 requirement of TCK being a free running clock. TCK must be stopped at one (1). The TCK input has a built-in pull-up resistor to ensure that a high signal is seen on an unconnected input.
TMS	Test Mode Select—Input—A serial control input that is clocked into the TAP controller on the rising edge of TCK. The TMS input has a built-in pull-up resistor to ensure a high signal value is seen on an unconnected input.

TDI	Test Data Input—Input—TDI is clocked into the least significant bit of the selected register, data or instruction, on the rising edge of TCK. The TDI input has a built-in pull-up resistor to ensure a high signal value is seen on an unconnected input.
TDO	Test Data Output—Output—The contents of the MSB of the selected register, data or instruction, is shifted out of the TDO on the falling edge of TCK. TDO is tri-stated except when scanning of data is in progress.
TRST	Test Reset—Active low input—TRST is the reset input to the TAP controller. Assertion of this input forces the TAP controller into the reset state. The TRST input has a built-in pull-down resistor to ensure a low signal values is seen on an unconnected input to force the TAP controller into the reset state.

C.2 TAP Controller (TAPC)

The tap controller (TAPC) is a synchronous finite state machine. This is where, under control of the TMS, sequencing through the various operations of the testability circuitry occurs. A definition of each TAPC state is listed in Table C-1 and a state diagram appears in Figure C-1.

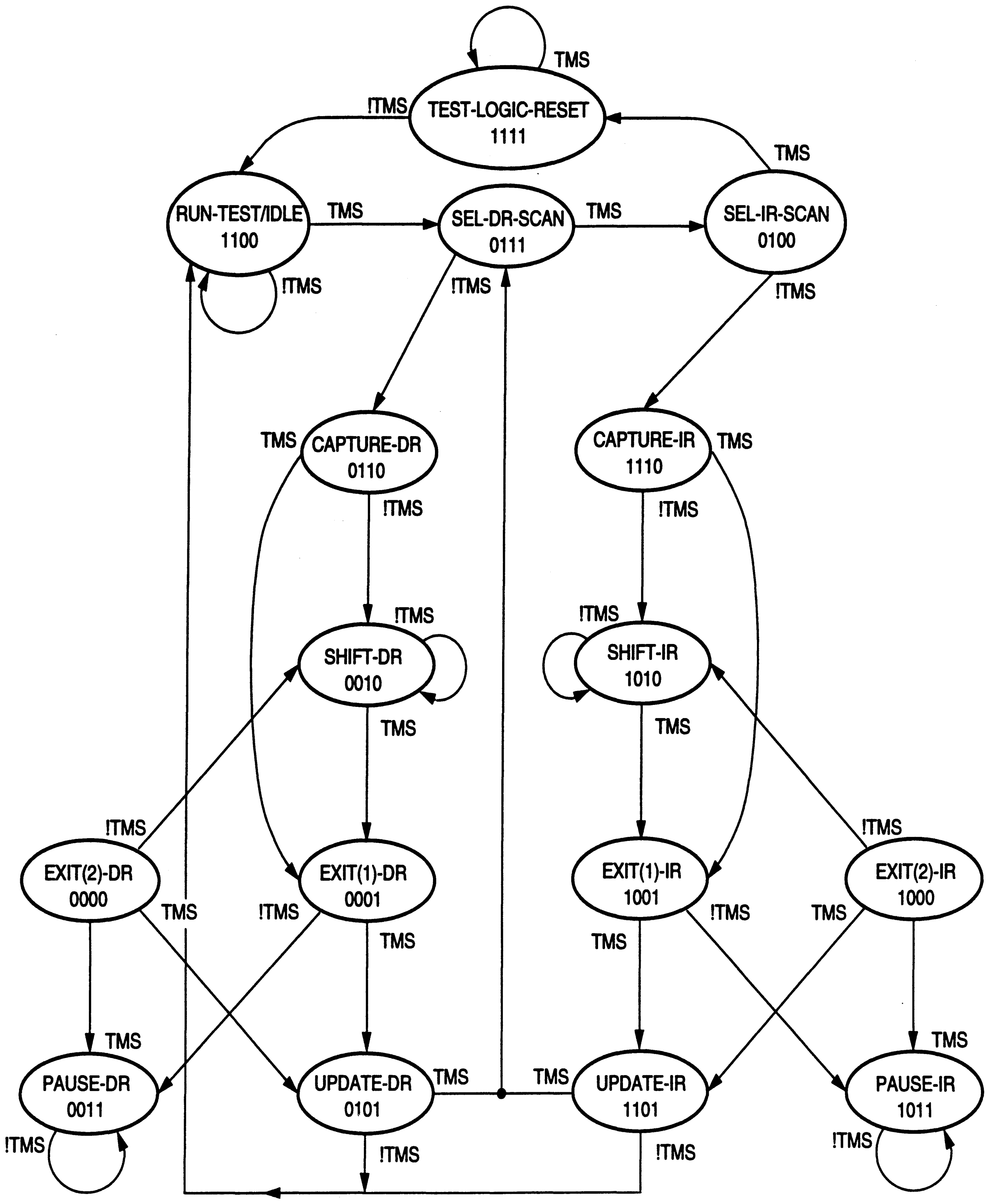
Table C-1 TAP Controller States (*Sheet 1 of 2*)

State	Description
0x0	**Exit(2)-DR.** This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the Shift-DR state.
0x1	**Exit(1)-DR.** This is a temporary controller state. All test data registers and the instruction register retain their previous state. TMS = 1 in this state causes termination of the scanning process; TMS = 0 causes entry into the Pause-DR state.
0x2	**Shift-DR.** In this controller state, the selected data register shifts data one stage towards its serial output on each rising edge of TCK. All registers other than the selected test data register retain their previous state.
0x3	**Pause-DR.** This controller state allows shifting of the selected test data register to be temporarily halted. All test data registers and the instruction register retain their previous state. The controller remains in this state while TMS = 0. When TMS goes high, the controller advances to the Exit(2)-DR state.
0x4	**Select-IR-Scan.** This is a temporary controller state in which all test logic retains its previous state. If TMS = 0 when the controller is in this state, then a scan sequence for the instruction register is initiated.

Table C-1 TAP Controller States (*Sheet 2 of 2*)

State	Description
0x5	**Update-DR.** During this controller state, data is transferred from each shift-register stage into the corresponding parallel output latch (if the selected test data register includes a parallel output latch). All shift-register stages in the selected register retain their previous state.
0x6	**Capture-DR.** In this controller state, data is parallel loaded into the selected test data register. If the register does not have a parallel input, or if capturing is not required for the selected test, the register retains its previous state unchanged.
0x7	**Select-DR-Scan.** This is a temporary controller state in which all test logic retains its previous state. If TMS = 0 when the controller is in this state, then a scan sequence for the selected test data register is initiated.
0x8	**Exit(2)-IR.** This is a temporary controller state. All test data registers and the instruction register retain their previous state. A high signal on the TMS line while in this state causes termination of the scanning process; a low causes entry into the Shift-IR state.
0x9	**Exit(1)-IR.** This is a temporary controller state. All test data registers and the instruction register retain their previous state. If TMS = 1 while in this state, the scanning process is terminated; if 0, the Pause-IR state is entered.
0xA	**Shift-IR.** In this controller state, the instruction register shifts data one stage towards its serial output on each rising edge of TCK.
0xB	**Pause-IR.** This controller state allows shifting of the instruction register to be temporarily halted. All test data registers and the instruction register retain their previous state. The controller remains in this state while TMS = 0. When TMS goes high, the controller advances to the Exit(2)-DR state.
0xC	**Run-Test/Idle.** The controller state between scan operations where an internal test previously selected by setting the instruction register may be executed. Registers not involved in the application of the test retain their previous state. If the data in the instruction register does not indicate that a test should be executed, then all test logic must retain their previous state. Once entered, the controller will remain in the Run-Test/Idle state as long as TMS = 0.
0xD	**Update-IR.** During this controller state, the instruction is transferred from each shift-register stage of the instruction register into the parallel output latch of the instruction register. All shift-register stages in the instruction register retain their previous state.
0xE	**Capture-IR.** In this controller state, data is parallel loaded into the instruction register. If the register does not have a parallel input, or if capturing is not required for the selected test, the register retains its previous state unchanged.
0xF	**Test-Logic-Reset.** While in this state, all test circuitry is disabled. The instruction register (IR) is reset to select the by-pass register. The controller remains in this state as long as TMS = 1 or $\overline{\text{TRST}}$ is asserted.

Figure C -1 TAP Controller State Diagram



C. 2. 1 Instruction Register

The instruction register (IR) allows a test instruction to be shifted into the ATT92010 *Hobbit* Microprocessor. The IR is used to select the test to be performed or the test data register to be accessed. The IR is seven (7) bits in length. Table C-2 identifies the instruction encoding.

Table C-2 TAP Instruction Register Encoding

Instruction	Register Selected	Instruction Mnemonic	Description
0000000	BS	EXTEST	BS selected with BS external test.
0000001	BS	SAMPLE	BS selected with BS sample.
0000010	BS	INTEST	BS selected with BS internal test.
0000011	PPLA	IRPPLA	PPLA selected with PPLA self-test.
0000100	ICD	IRICD	Instruction cache data selected with ICD self-test.
0000101	SC	IRSC	Stack cache selected with SC self-test.
0000110	PFD	IRPFD	Prefetch cache data selected with PFD self-test.
0000111	PFT	IRPFT	Prefetch cache tag selected with PFT self-test.
0001xxx	NA	NA	Reserved.
001xxxx	BP	BP	BP selected with all self-test.
01xxxxx	BP	BP	BP selected and BS sample.
10xxxxx	ID	ID	ID selected and BS sample.
11xxxxx	BP	BP	BP selected and BS sample.

C. 2. 2 By-Pass Register

The by-pass (BP) register provides a single TCK delay path from TDI to TDO. When the BP register is selected, a 0 is loaded on the rising edge of TCK in the Capture-DR controller state. When the Test-Logic-Reset controller state is entered, the BP register retains its last value.

C. 2. 3 Boundary-Scan Register

The boundary-scan register (BS) allows testing of circuitry external to the ATT92010 *Hobbit* Microprocessor. Additionally, BS provides for sampling and examination of the I/O values without impacting the operation of the system logic. Ninety shift elements are in the boundary-scan shift chain. Ninety-one TCKs are required to shift the entire chain from TDI through to TDO. Position is given from TDI to TDO.

Table C-3 Boundary-Scan Shift Chain (*Sheet 1 of 2*)

Position	Name	Description	Position	Name	Description
1	$\overline{\mathrm{HRESET}}$	I	2	CLK23	Sample Only I
3	$\overline{\mathrm{STOP}}$	Sample Only I	4	CLK34	Sample Only I
5	$\overline{\mathrm{DTRI}}$	I	6	3-data	Control for I/O
7	D31	I/O	8	HD30	I/O
9	HA22	3-State O	10	HD4	I/O
11	HA15	3-State O	12	HD3	I/O
13	HA3	3-State O	14	HD29	I/O
15	HA14	3-State O	16	HD28	I/O
17	HA2	3-State O	18	HD27	I/O
19	HA13	3-State O	20	HD26	I/O
21	HA31	3-State O	22	HD25	I/O
23	HA30	3-State O	24	HD24	I/O
25	HA29	3-State O	26	HD23	I/O
27	HA12	3-State O	28	HD22	I/O
29	HA21	3-State O	30	HD21	I/O
31	HA11	3-State O	32	HD20	I/O
33	HA20	3-State O	34	HD7	I/O
35	HA10	3-State O	36	HD6	I/O
37	HA19	3-State O	38	HD5	I/O
39	HA9	3-State O	40	HD19	I/O
41	HA28	3-State O	42	HD18	I/O
43	HA8	3-State O	44	HD17	I/O
45	HA27	3-State O	46	HD16	I/O

Table C-3 Boundary-Scan Shift Chain (*Sheet 2 of 2*)

Position	Name	Description	Position	Name	Description
47	HA26	3-State O	48	HD15	I/O
49	HA25	3-State O	50	HD14	I/O
51	HA7	3-State O	52	HD13	I/O
53	HA18	3-State O	54	HD2	I/O
55	HA6	3-State O	56	HD1	I/O
57	HA17	3-State O	58	HD0	I/O
59	HA5	3-State O	60	HD12	I/O
61	HA16	3-State O	62	HD11	I/O
63	HA4	3-State O	64	HD10	I/O
65	HA24	3-State O	66	HD9	I/O
67	HA23	3-State O	68	HD8	I/O
69	D/$\overline{\text{T}}$	3-State O	70	$\overline{\text{NCACHE}}$	3-State O
71	$\overline{\text{RD}}$	3-State O	72	$\overline{\text{BE0}}$	3-State O
73	$\overline{\text{BE1}}$	3-State O	74	$\overline{\text{BE2}}$	3-State O
75	$\overline{\text{BE3}}$	3-State O	76	IOC1	3-State O
77	IOC0	3-State O	78	$\overline{\text{LOCK}}$	3-State O
79	START	3-State O	80	$\overline{\text{BGACK}}$	O
81	$\overline{\text{BREQ}}$	O	82	3-bus	Control for 3-State O
83	$\overline{\text{BGRANT}}$	I	84	$\overline{\text{RETRY}}$	I
85	$\overline{\text{BERR}}$	I	86	$\overline{\text{HOLD}}$	I
87	$\overline{\text{DTACK}}$	I	88	IL2	I
89	IL1	I	90	IL0	I

C. 2. 4 Identification Register

The ID register is readable by serial shifting through the TAP and through normal register access. This register is described in detail in section 1.5.3.

Glossary

A

ALU

Arithmetic Logic Unit

C

CISC

Complex Instruction Set Computing

CMOS

Complementary Metal-Oxide Semiconductor

CRISP

C-Language Reduced Instruction Set Processor

CSP

Current Stack Pointer

D

DRAM

Dynamic RAM

DQM

Double-word or Quad-word Move

E

EU

Execution Unit

EPROM

Erasable Programmable Read-Only Memory

I

IC

Instruction Cache

IR

Instruction Register

ISP

Interrupt Stack Pointer

J

JTAG

Joint Test Action Group

M

MHBI

Multiplexed *Hobbit* Bus Interface

MIL

Machine Interface Layer

MSP

Maximum Stack Pointer

N

NOP	No-Operation
NPSR	Nonpaged Segment Register

P

PBR	Prefetch Buffer Register
PC	Program Counter
PCMCIA	Personal Computer Memory Card International Association
PDU	Prefetch Decode Unit
PDR	Prefetch Decode Register
PFB	Prefetch Buffer
PIR	Prefetch Instruction Register
PSW	Program Status Word

R

RISC	Reduced Instruction Set Computing
RR	Result Register

S

SC	Stack Cache
SHAD	Shadow
SP	Stack Pointer

T

TAP	Test Access Port
TAPC	Test Access Port Controller
TCK	Test Clock
TLB	Translation Lookaside Buffer
TMS	Test Mode Select

V

VB

Vector Base

VP-1

Virtual/Physical Addressing Bit 1

VRAM

Video RAM

Index

A

Address
- fault 2-3
- mapping 2-3
- nonpaged segment 2-4
- page tables 2-6
- paged segment 2-4
- virtual 2-3

Addressing 1-3

Addressing modes 1-27

Alignment
- half-words 1-4

Architecture 1-1

Arithmetic
- division and remainders 1-30
- instruction
  - integer operation 1-28
- tagged integer 1-30

B

Bits
- numbering 1-3

Block Diagram 1-2

Branches
- conditional 1-33
- prediction
  - folding 1-33

Buffer
- translation look-aside 1-18

Bus
- central arbiter A-1
- cycles A-3
- error A-6
- exception handling A-6
- interlocked transfer A-6
- protocol A-1
- read transactions A-4
- reset A-8
- retry A-6
- surrendering A-2
- transaction types A-4
- write transactions A-4

Bus master A-1

Byte
- Big-endian 1-4
- Little-endian 1-4

Bytes
- numbering 1-3

C

Carry bit
- unsigned overflow 1-29

CISC 1-1

CRISP architecture 1-1

D

Data type 1-3

Delays
- data type 4-7
- direct 4-3
- empty pipeline 4-6
- fetch 4-5
- hazards 4-6
- indirect 4-3
- internal contention 4-6
- miscellaneous 4-7
- operand access 4-6

E

Electrical
- BREQ and BGACK B-10
- bus relinquish cycle B-9
- clock input timing B-5
- DTRI to DATA B-10
- handling precautions B-1
- JTAG bus B-11
- maximum ratings B-1
- operating conditions (20MHz) B-2
- operating conditions (30MHz) B-3
- output derating factors B-4
- output timing B-8
- synchronous input timing B-6
- test loading B-4

Electrical characteristics
- handling precautions B-1

Encoding
- five-parcel 1-25
- general addressing mode 1-25
- one-parcel 1-22
- three-parcel 1-24

Event processing
- exception sequence 1-38
- exceptions 1-38
- interrupt 1-36
- nonmaskable interrupt 1-37
- priority 1-40
- reset 1-35
  - interrupt
    - exception 1-34
- trapped niladic exception 1-40
- unimplemented instruction 1-39
- unimplemented instruction sequence 1-39

Event processing interrupt sequence 1-37

F

Features 1-1

I

Instruction
- LDRAA 1-13

Instruction fetching
- aggressive
  - demand 1-32

Instruction set
- ADD 3-5
- ADDI 3-6
- AND 3-7
- ANDI 3-8
- CALL 3-9
- CATCH 3-10
- CLRE 3-12
- CMP 3-13
- CPU 3-14
- CRET 3-15
- DIV 3-17
- DQM 3-18

Instruction set (continued)
- ENTER 3-19
- FLUSHD 3-22
- FLUSHDCE 3-23
- FLUSHI 3-24
- FLUSHP 3-25
- FLUSHPBE 3-26
- FLUSHPTE 3-27
- format 3-1
- JMP 3-28
- KCALL 3-29
- KRET 3-30
- LDRAA 3-31
- MOV 3-32
- MOVA 3-33
- MUL 3-34
- NOP 3-35
- OR 3-36
- ORI 3-37
- pipeline consideration 3-3
- POPN 3-38
- REM 3-39
- RETURN 3-40
- SHL 3-41
- SHR 3-42
- SUB 3-43
- TADD 3-44
- TESTC 3-44, 3-45
- TESTV 3-46
- TSUB 3-47
- types 3-1
- UDIV 3-48
- UREM 3-49
- USHR 3-50
- XOR 3-51

Instructions
- five-parcel format 1-24
- one-parcel format 1-21
- three-parcel format 1-23
- tracing 1-33
- unimplemented 1-39

Integer accumulator 1-7

M

Memory management, see MMU

MMU
- address translation 2-1
- nonpaged segments 2-1
- operations 2-8
- paged segments 2-1
- performance 2-9
- physical prefetch buffer 2-1
- TLBs 2-1

MSP
- see registers 1-5

O

Overflow bit
- signed 1-29

P

Performance
- branch folding 4-7
- conditional jump 4-5
- delays 4-2
- DQM instructions 4-4
- execution time 4-1
- multi-cycle instructions 4-4

Pipeline 3-3

Procedure
- fast calling 1-30

R

Registers
- configuration 1-9
- control 1-8
- fault 1-10
- ID 1-11
- interrupt stack pointer 1-12

Registers (continued)
- JTAG, see ID
- maximum stack pointer 1-12
  - stack pointer 1-5
- program counter 1-13
- program status word 1-14
- segment table base 1-18
- shadow 1-16
- stack pointer 1-17
- timer1 1-19
- timer2 1-19
- vector base 1-20

RISC 1-1

S

Segment tables 2-4
- formats 2-4
- mixed paged 2-6
- nonpaged 2-6

Sequence
- fast calling 1-30

SP
- see registers 1-5

Stack cache
- instructions 1-6
- maintenance 1-6
- organization 1-5
- precautions 1-8
- register allocation 1-5

Stack frame 1-6, 1-31

T

Test
- access port C-1
- boundary-scan register C-6
- by-pass register C-5
- clock C-1
- controller (TAPC) C-2

Test (continued)
- data input C-2
- data output C-2
- external circuit C-6
- instruction register C-5
- mode select C-1
- reset C-2

Testability
- features C-1

Timing
- see Electrical

TLB
- micro 2-1

Translation lookaside buffers, see TLB

W

Word
- double 3-18
- quad 3-18

For additional information, contact your AT&T Account Manager or the following:

U.S.A.: AT&T Microelectronics, Dept. AL-500404200, 555 Union Boulevard, Allentown, PA 18103
1-800-372-2447, FAX 215-778-4106 (In CANADA: **1-800-553-2448**, FAX 215-778-4106)

ASIA PACIFIC: AT&T Microelectronics Asia/Pacific, 14 Science Park Drive, #03-02A/04 The Maxwell, Singapore 0511
Tel. (65) 778-8833, FAX (65) 777-7495, Telex RS 42898 ATTM

JAPAN: AT&T Microelectronics, AT&T Japan Ltd., 7-18, Higashi-Gotanda 2-chome, Shinagawa-ku, Tokyo 141, Japan
Tel. (81) 3-5421-1600, FAX (81) 3-5421-1700

For data requests in Europe:

AT&T DATALINE: **Tel. (44) 732 742 999**, FAX (44) 732 741 221

For technical inquires in Europe:

CENTRAL EUROPE: **(49) 89 950 860** (Munich), NORTHERN EUROPE: **(44) 344 48711** (Bracknell UK), FRANCE: **(33) 47 67 47 67**,
SOUTHERN EUROPE: **(39) 266 011 800** (Milan) or **(34) 1 807 1441** (Madrid)

January 1993
MN91-043MCP

AT&T
Microelectronics