SPARClite
MB86860 Series
Programming Manual

Edition 1.0 - Jun. 22, 1999

Fujitsu Ltd.

Contents

0. Introduction

MB8686 Series processors are 32-bit RISC processors for installation which have architecture conforming to SPARC Version 8. In this Manual the architecture and actual programming methods in the system of the MB86860 Series processor are discussed. This Chapter concerns system programs and application programs which use MB86860 Series processors.

0.1. Configuration of this Chapter

0.2. Notation

- In this Manual, the MB86860 Series Processor is denoted as the MB8686x Processor. For product characteristic information, the nomenclature is like that of the MB86860 and MB86861.
- Register notation is as follows:

X	- Register Name
X.A,X.B,X.C	- Indicates bit fields inside registers.
X[5:0]	- Indicates register interior broken down by bits (bits 5-0) (X.A=X[31:24], X.B=X[23:16] , X.C=X[15:0])

Register Name : X

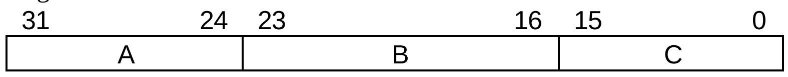


Figure 0-1 Register Notation

- Radix Notation

  Hexadecimal ....: "0x " is used as prefix

  Decimal : No prefix used

  Binary : "0b" is used as prefix

0.3. Attached Program Listings

The program listings attached to this Manual are all drafted with the Cygnus Corporation "GNUPro Toolkit for SPARC86x" as the development environment.

1. Programming Models

MB8686 Series processors are high capacity processors for installation which have architecture conforming to SPARC Version 8. In this Manual implementations in SPARC architecture programming model MB8686 processors are explained.

1.1. Program Modes

Two operating modes are defined in the SPARC architecture- user mode and supervisor mode, and program and data protection functions are supported in a multitask environment. Issuance of privileged instructions is restricted for programs operating in User Mode (Issuance of privileged instructions by user mode programs causes privileged instruction traps to occur). Program Mode moves cause traps, and traps are caused by RETT instructions. (See "*3. Traps*" for details).

1.2. Setting Registers

MB8686 processors, broadly divided, have the following 3 kinds of registers:

- IU r-registers
  32-bit general purpose registers. Consists of 8 global registers and windowed register fields.
- IU Control/Status registers
  Registers used for displaying processor status and for processor control purposes.
- Memory Mapped registers
  Registers mapped in memory spaces. Used as system control or status display registers.

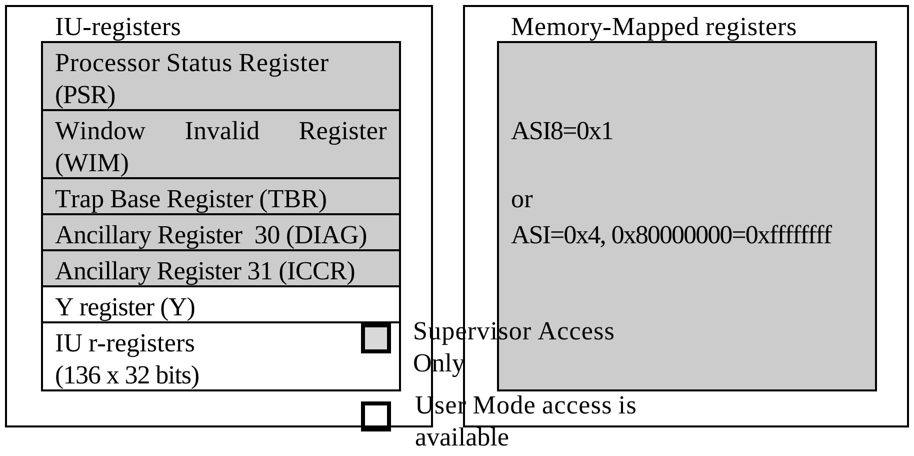

IU-registers
Processor Status Register (PSR)
Window Invalid Register (WIM)
Trap Base Register (TBR)
Ancillary Register 30 (DIAG)
Ancillary Register 31 (ICCR)
Y register (Y)
IU r-registers (136 x 32 bits)
Supervisor Access Only
User Mode access is available
Memory-Mapped registers
ASI8=0x1
or
ASI=0x4, 0x80000000=0xffffffff

Figure 1-2 Setting Registers

1.2.1. IU r-registers

IU r-registers are 32-bit registers which can be used for general purposes. The MB86860 Series processor has 8 global registers stipulated by the SPARC architecture and 128 windowed registers (number of windows=8) built in. 8 global registers and 24 windowed registers can be accessed at any time.

1.2.1.1. Register Windows

The SPARC architecture uses windowed register file models. Figure 1-3 shows register window model drawings. Programs can access the 8 global registers and the 24 r-registers inside register windows at any time. The 24 registers are further classified into 3 kinds of registers:

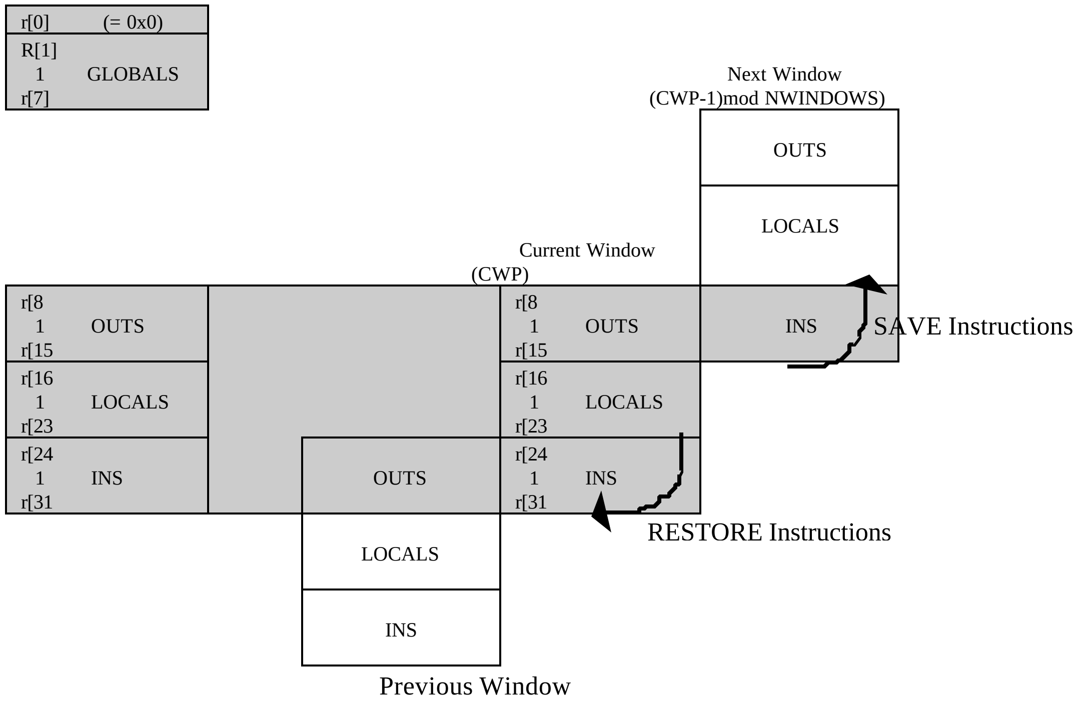
- OUT registers (r[18] - r[15] or o[0]-o[7])
  share next-window IN register.
- LOCAL registers (r[16]-r[23] or 1[0]-1[7])
  characteristic register in current window.
- IN registers (r[24]-r[31] or i[0]-i[7])
  share previous window OUT register.

By taking advantage of this register mechanism and using shared registers, transfer of parameters in function calls can be executed at high speed. CWP (Current Window Pointers) which indicate current window positions are set in PSR and CWP fields. CWPs can be operated by the SAVE and RESTORE instructions.

- SAVE
  CWP is moved to the Next Window. (CWP ← (CWP-1) (moduloNWINDOWS)
- RESTORE or RETT
  CWP is moved to Previous Windows. (CWP ← (CWP+1) (moduloNWINDOWS)

SAVE operations are automatically performed when traps occur.

Here, NWINDOWS shows the number of windows being implemented. In MB8686 processors this is NWINDOWS=8. CWP decrements by SAVE instructions and increments by RESTORE instructions are performed in NWINDOWS modulo units. Thus window(0-1)=window(NWINDOS-1) window(NWINDOWS)=window(0).



((CWP+1) mod NWINDOWS)

Figure 1-3 Register Windows

1.2.1.2. Window Overflow/Window Underflow

When a register is used up, the oldest registers used must be saved in memory. WIM (Window Invalid Masks) must be set to detect these overflows (or underflows). When a SAVE instruction is issued and CWP+1 matches a window set to WIM, it generates a window overflow trap. When a RESTORE or RETT instruction is issued and the next window matches a window set to WIM, it generates an underflow trap. Register save/return processing must properly be performed by window overflow/underflow traps.

[Note] PSR and CWP fields and WIM registers are undefined when the power is turned on. PS and WIM must be set by activation processing programs after the power is turned on.

[Note] An automatic SAVE operation is performed when a trap occurs, but no window overflow checks are performed. It is therefore necessary to check window overflow inside trap handlers.

[Note] It is always possible that traps may occur while programs are running. In preparation for the occurrence of traps 1 window must be kept empty.

1.2.1.3. Special r-Registers

Some of the ways in which they are used have been predetermined inside the r-Registers.

- r[0](g1[0])
  This register is used as a zero register. 0x0 is read when it is used as a source operand, and when it is used as a destination operand, write data is discarded.

- For CALL instructions, the address of the instruction itself is written to r[15[](o[7]).
- When traps occur, program counter PC and nPC are saved respectively in new windows r[17](1[1]) and r[18](1[2]),

1.2.2. IU Control / Status Registers

IU (Integer Unit)

1.2.3. Memory Mapped Registers

1.3. Address Spaces

1.3.1. ASI (Address Space Identifiers)

MB8686 Series processors configure addresses of 1T bytes using ASIs (Address Space Identifiers) indicated by 8 bits together with the 32-bit address spaces which they have. In 86x processors, ASIs are assigned as shown is Table 1-1, and ASI=0x9 and 0xb (supervisor), ASI=0x8 and 0xa (user) are assigned to program and data areas. These spaces can be accessed by ordinary load/store instructions, but for other spaces having ASIs STA and LDA (privileged instructions) must be used. Hence the above ASIs must be assigned to memory devices which store programs and data. Also, ASI=0x4–0x7 can be used as user I/O space. (However, the STA and LDA instructions must be used).

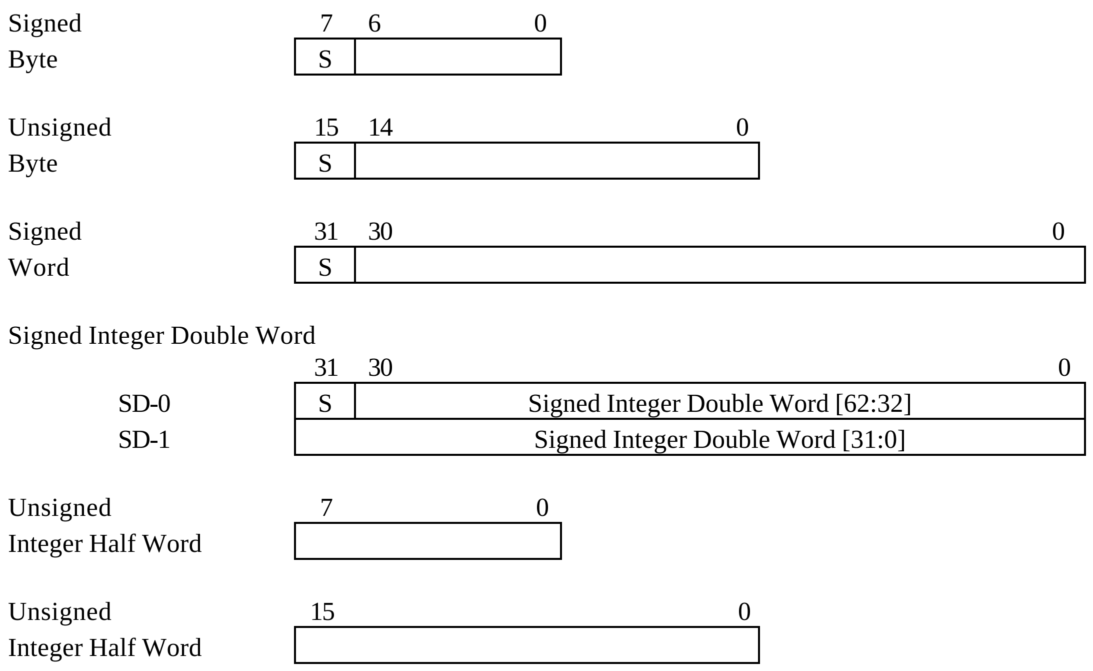
Table 1-1 MB86860/MB86861 ASI Assignments

ASI	Function
0h	reserved
1h	special regs.
2h	reserved
3h	reserved for TLB probe
4h	internal regs. / user I/O
5h - 7h	user i/o regs.
8h	user inst.
9h	supervisor inst.
ah	user data
bh	supervisor data
ch	I Cache TAG diagnostic
dh	I Cache Set diagnostic
ch-fh	reserved
10h-14	I Cache/D Cache line flush IB/RB flush
15h-17	reserved
18h-1b	I Cache line flush IB flush
1ch	D Cache TAG diagnostic
1dh	D Cache Set diagnostic
1ch-30h	reserved
31	Flush entire I Cache/D Cache IB/RB flush
32h-ffh	reserved

▩ reserved
▒ Supervisor access Only
☐ Supervisor /User mode

1.4. Data Types

MB8686 processors support the following data types:


Signed Byte
7 6 0
S
Unsigned Byte
15 14 0
S
Signed Word
31 30 0
S
Signed Integer Double Word
31 30 0
SD-0
S
Signed Integer Double Word [62:32]
SD-1
Signed Integer Double Word [31:0]
Unsigned Integer Half Word
7 0
Unsigned Integer Half Word
15 0

Unsigned Integer Word

31	0

Unsigned Tagged Integer Word

31 2	1 0
	tag

Unsigned Integer Double Word

UD–0

31	0

UD–1

31	0

S :Sign flag
SD–0: Signed Integer Double Word (Address alignment 0mod8 word data)
SD–1: Signed Integer Double Word (Address alignment 4mod8 word data)
UD–0: Unsigned Integer Double Word (Address alignment 0mod8 word data)
UD–1: Unsigned Integer Double Word (Address alignment 4mod8 word data)

1.5. Instruction Format

- Instruction Format

Format 1 (OP=1) :CALL instruction

31 30	29 0
OP	disp30

Format 2 (OP=0) :SETH instruction, BRANCH instruction

31 30	29	28 25	24 22	21 0
OP	rd		OP2	imm22
OP	a	cond	OP2	disp22

Format 3 (OP=2 or 3) :Instructions other than CALL, SETH, BRANCH

31 30	29 25	24 19	18 14	13	12 5	4 0
OP	rd	OP3	rsl	i	asi	rs2
OP	rd	OP3	rsl	i	simm13	
OP	rd	OP3	rsl	opf		rs2

- OP, OP2, OP3

These 3 fields are op codes. Most op code instructions are as shown below. For details, see the *Instruction Set Manual*.

OP	OP2	OP3	Instruction	Format
	000	—	not loaded	
	001	—	not loaded	
	010	—	Bicc	

00	011	—	not loaded	2
	100	—	SETHI	
	101	—	not loaded	
	110	—	FBfcc	
	111	—	CBccc	
01	—	—	CALL	1
1X	—	XXXXXX	Except CALL, SETH, BRANCH(Bicc,FBfcc,CBccc	3

- rd, rs1, rs2

These 3 fields are fields which indicate register addresses of general purpose registers.
rd fields omit showing source operands in ST (Store) instructions and indicate destination operands. All rs1 and rs2 fields indicate source operands except when executing ST instructions.

- disp30, disp22

  These 30-bit and 22-bit fields show PC (Program Counter) relative displacement. These displacements are used word-aligned and sign-expanded.

- a

  This field is contained in the BRANCH instruction, and it specifies cancellation of execution of instructions which exist in delay slots.

- cond

  This field is contained in conditional BRANCH instructions, and they specify the condition codes which test the BRANCH conditions.

- imm22

  This field is contained in the SETH instruction, and it displays constants set to the upper order 22 bits of destination registers.

- i

  This field is contained in arithmetic instructions and load/store instructions, and it is the field which selects the method of specifying the 2nd operand. If I=0, the rs2 field indicates a register address, and for the 2nd operand, general purpose register r[rs2] is selected. If I=1, the simm13 field indicates an immediate value of 13 bits, and an immediate value sign-expanded to 32 bits is the 2nd operand.

- simm13

  This field is contained in arithmetic and load/store instructions where the i field is “1”, and it indicates an immediate value of 13 bits. An immediate value sign-expanded to 32 bits is the 2nd operand of the instruction.

- asi

  This field is contained in load/store alternate instructions and indicates the address spaces to be selected.

- opf

  This is the 4th operand field which encodes floating point instructions and coprocessor instructions. Floating point units and coprocessors are not loaded in the MB8683 Series. Floating point instructions and coprocessor instructions are trapped for software emulation.

1.6. Trap Models

1.6.1. Kinds of Traps

- Strict Traps

  Strict traps are activated by specific instructions, and they occur before conditions which can be recognized by programs are changed by these instructions. If a strict trap occurs, some conditions must be kept in force.

- Delay Traps

- Interrupt Traps

1.6.2. Trap Types

Table 1-2 shows an overview of trap types in MB8686x processors.

Table 1-2 Overview of Trap Types

Trap	Priority	TBR.tt	Causes
Reset	1	—	External system asserted pin RESET#.
Instruction Acces Exception	2	1	Input signal to MEXC# pin was asserted during instruction access to external buses.
Privileged Instruction	3	3	Privileged instruction was executed in User Mode.
Illegal Instruction	4	2	Undefined instructions, UNIMP instructions and instructions giving rise to incorrect processor conditions (incorrect values written to CWP fields of Register PSR etc.) were executed. However, undefined FPop and CPop instructions become FPU absent exception and coprocessor absent exception traps.
FPU Absent Exception	5	4	FPop, FBfcc or floating point load/store instructions were executed.
Coprocessor Absent Exception	5	36	CPop, CBccc or coprocessor load/store instructions were executed.
Window Overflow	6	5	A SAVE instruction gave rise to a CWP pointing to a window invalidated by Register WIM.
Window Underflow	7	6	A RESTORE or RETT instruction gave rise to a CWP pointing to a window invalidated by Register WIM.
Address Misalign	8	7	Load/Store generated addresses which are not properly aligned. JMPL or RETT instructions generated addresses which are not word-aligned.
Data Access Exception	10	9	Input signal was asserted to MEXC# pin during data access to external buses.
Tag Overflow	11	10	At least 1 operand bit is not “0”, or an arithmetic overflow occurs during execution of TADDccTV or SUBccTV instructions.
Trap Instruction	12	128-255*	Ticc instruction was executed and a judgment made that trap condition is true.
Break Point	1.5	255	An instruction or data matches a break point.
Interrupt Level 15-1	14-28	31-17	External interrupt request.

*Specifications which generate 255 in a field are prohibited.

2. Initialization

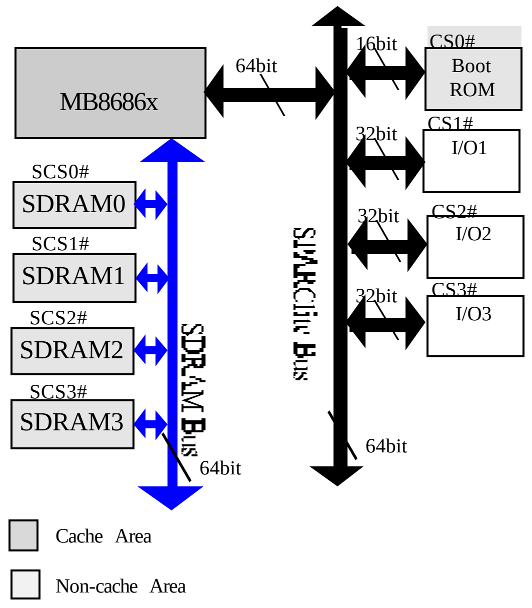
When external reset requests are input to the processor, the processor generates reset traps and starts running programs from 0x0 addresses (CS0# areas). Establishing processor status, assignment of address spaces to all system resources and other initialization processing must be performed by initialization programs which start at 0x0 addresses. In this Chapter, required setting methods for each kind of register and matters requiring attention etc. in initialization programs which are run following reset cancel will be discussed.

2.1. Register Models

This Chapter explains initialization programs for system configurations using SPARClite-S, with the system configuration shown below as an example.

- System Clock Frequency 33MHz, Clock Frequency Multiple 3
- Configuration using 64Mbit (16 bits/word) SDRAM X 4 and 4 chip-select areas.
  (SDRAM capacity) = (8Mbyte) x 4 x 4 = 128Mbyte
- Uses Boot-ROM in 16-bit bus width, and access wait number = 7 system clock cycles.
- Uses I/O1~I/O3 in 32-bit bus width, and all I/O wait numbers = 3 system clock cycles.

Figure 2-4 shows an example of system configuration, and Figure 2-5 shows memory mapping.



ASI=0x8, 9, a, b

CS0#	ROM	0x00000000	ROM Area wait cycle=6 cycle (non-cache Area)
SCS0#	SDRAM0	0x40000000	SDRAM Area (cache area)
SCS1#	SDRAM1	0x42000000	
SCS2#	SDRAM2	0x44000000	SDRAM Area (non-cache area)
SCS3#	SDRAM3	0x44000000 0x47ffffff	

() = MB86860

ASI=0x7

CS1#	I/O1	0x100000000	SDRAM Area wait cycle = 3 cycles (non-cache area) (CS1,CS2,CS3)
CS2#	I/O2	0x20000000	
CS3#	I/O3	0x30000000	

Figure 2-4 Example of System Configuration **Figure 2-5 Memory Maps**

SPARC processors assert CS0# after reset cancel and start running from 0x0 addresses (ASI=0x9). Therefore, boot processing programs must be mapped in 0x0 addresses.

2.2. Initialize Flow

The following settings must be made by initialization programs:

- Processor intialization (TBR, WIM, PSR)

- Address space assignments (ARSR, AMR, WSSR, SDARS, SDAM)
- Stack frame creation
- Cache and buffer initialization

From Section 2.3 onward, individual initialization processing will be explained.

2.3. Processor Initialization

After reset cancel, processors are in the status shown below and start running.

- Supervisor Mode (PSR.S=1)
- Interrupts prohibited (PSR.ET=0)
- Program Counter =0x0 nPC=0x4.
- All other IU control/status register fields keep pre-reset values (undefined when power is turned on).

In boot programs following reset cancel, it is first of all necessary to initialize the IU control/status registers (PSR, TBR, WIM) in order to define processor status. Status register read/write instructions (RD/WRPSR, RD/WRTBR, RD/WRWIM) are used for reads/writes to the IU control/status registers. Status register read/write instructions are provileged instructions.

list2-1 Processor Initialization

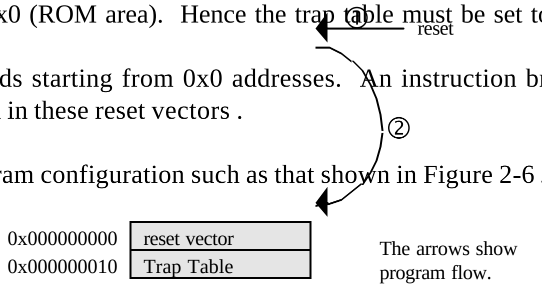
```
/*
 * Initialize processor at reset
 * PSR.PIL   <= 0xf  –  Mask all interrupts except for NMI
 * PSR.S     <= 0x1  –  Supervisor mode
 * PSR.PS    <= 0x0  –  User mode
 * PSR.ET    <= 0x1  –  Enable Traps
 * PSR.CWP   <= 0x7  –  Current Window Pointer = 0x7
 * PSR.WIM   <= 0x1  –  Window Invalid Mask = window 0
 * PSR.TBR       <=  0x0    –         Trap Base Address
 */
 _reset:
            wr    %g0,   0x0fa7,    %psr
            wr    %g0,   0x1,       %wim
            wr    %g0,   0x0,       %tbr
            nop
            nop
            nop
```

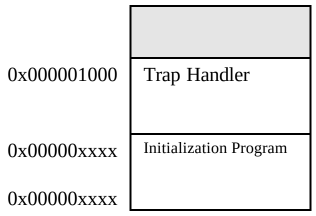
(Note] Read values of registers which perform writes in the 3 instructions following Status register write instructions (WRPSR, WRTBR, WRWIM) are not guaranteed.

[**Program Configuration]**

- In list2-1, settings are made to TBR=0x0 (ROM area). Hence the trap table must be set to 0x00000000-0x00000fff.
- Reset vectors are always set to 4 words starting from 0x0 addresses. An instruction branching to an initialization routine must be contained in these reset vectors .

Boot ROM must have a program configuration such as that shown in Figure 2-6 .





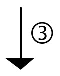


Figure 2-3 Program Configuration

2.4. Processor Type Judgments

In MB8686 Series processors, type judgments between processors can be made using PSR.imp bits, PSR.ver bits and IDR registers (address=0x80000ff0 asi=0x4).

Table 2-1 SPARClite Processor Judgments

PSR.imp	PSR.ver	
0000	except 1111	MB86930Series
	1111	MB86830 Series
0001	1111	MB86860 Series

		IDR(address=0x80000ff0, asi=0x4)
		ID=0x0860xxxx – MB86860
		ID=0x0861xxxx – MB86861
		ID=0x0862xxxx – MB86862

(Lower order 16 bits undefined)

2.5. Memory Area Settings

Address space assignments, bus widths, access methods and other settings for each device must be made in order to establish connections between the processor and all types of resources in the system. The following registers are used to set memory areas:

- SPARClite bus width – ARSR, AMR, WSS
- SDRAM bus width – SDARS, SDAMR

2.5.1. SPARClite Bus Area Settings

2.5.1.1. Address Area Settings

CS Area address range settings use the ARSR registers and the AMR registers. The ARSR and AMR are both set, and, by accessing the appropriate addresses, the corresponding signals are asserted.

[Setting Examples]

[CS1# Areas]

Address Range	0x10000000 – 0x1fffffff
ASI	0x7
Bus Width	32bit
Cacheability	non-cacheable

Set Values:

Bit Field	Set Value	Explanation
ARSR1	0xa0071000	
ARSR1.N	0x1	non-cache
ARSR1.BW	0x1	32-bit bus width
ARSR1.ASI	0x7	ASI=0x7
ARSR1.Address	0x1000	Start=0x10000000

Bit Field	Set Value	Explanation
AMR1	0x00000fff	

AMR1.ASImask	0x0	ASI=0x7
AMR1.Addressmask	0x0fff	address = 0x10000000 - 0x1fffffff

2.5.1.2. Wait State Controller Settings

MB8686 Series processors have built in wait state controllers, and wait states can be set for each CS area. The number of wait states is CNT1, CNT2 (set value+1).

[Setting example]

CS1# Areas:

Number of Wait States	3 cycles
Burst Transfer	not used
Parity Function	not used
Override	not used

Set Values:

Bit Fields	Set Values	Remarks
WSSR1	0x00000850	
WSSR1.CN1	0x2	wait cycle = 2+1
WSSR1.CN2	0x2	wait cycle = 2+1
WSSR1.WE	0x1	wait enable
WSSR1.OVR	0x0	
WSSR1.SCB	0x0	
WSSR1.PE	0x0	

2.5.2. SDRAM Bus Area Settings

The registers shown below are used in setting SDRAM areas. Please note that setting registers and setting methods differ between the MB86860 and the MB86861.

[Setting example]

SDCS0# Area:

Address Range	0x40000000 - 0x43ffffff
ASI	0x8, 9, a, b
Cacheability	Cacheable

MB86860:

Set Values

Bit Fields	Set Values	Remarks
SDARS0	0x00084000	
SDARS0.N	0x0	Cacheable
SDARS0.ASI	0x8	ASI=0x8
SDARS0.Address	0x4000	Start = 0x40000000

Bit Fields	Set Values	Remarks
SDAM0	0x00033fff	
SDAM0.ASImask	0x3	
SDAM0.Addressmask	0x3fff	address= 0x40000000 -

		0x43ffffff

MB86861:

(T. B. D.)

2.5.3. Using ASIs

In SPARClite SS processors, ASIs are regulated as shown below. Use ASI=0x8,9,a,b in program memory and ASI 0x4,5,6,7 in user I/O assignments.

ASI	Function
0x4	Internal registers / User I/O
0x5-7	User I/O
0x8	User instruction
0x9	Supervisor instruction
0xa	User data
0xb	Supervisor instruction

[Note] ASI=0x4 and Address=0x80000000 - 0xffffff areas are reserved as internal register areas. When using ASI=0x4 as I/O areas, use the Address=0x00000000 - 0x7ffffff range.

2.5.4. Sample Programs

SPARClite Bus and SDRAM Bus area mapping examples are shown.

list2-2 Example of SPARClite Bus / SDRAM Bus Area Memory Mapping (MB86860)

```
#define SPL_860_ASI 0x4
#define MB86860     0x860
#define MB86861     0x861

/*  MB86860 memory-mapped register address settings      */
/*  SPARClite-bus area settings */

#define ARSR_BASE   0x80000100
#define AMR_BASE    0x80000200
#define WSSR_BASE   0x80000400
#define SDARS_BASE  0x80000130
#define SDAM_BASE   0x80000230

/*  CS0#area settings */
#define ARSR0_DAT   0x40080000    /* 16bit-bus, cacheable     */
#define AMR0_DAT    0x00030fff    /* adr=0x00000000-0fffffff, */
                                  /* asi=0x8, 9, a, b /*
#defineWSSR0_DAT    0x00001cf0    /* wait cycle=0x7, wait enable */

/*  CS1# area settings */
#define ARSR1_DAT   0xa0071000    /* 32bit-bus, non-cache */
#define AMR1_DAT    0x00000fff    /* adr=0x100000000-1fffffff */
                                  /* asi=0x7 */
#define WSSR1_DAT   0x00001cf0    /* wait cycle=7, wait enable */

/*  CS2# area settings */
#define ARSR2_DAT   0xa0072000    /* 32bit-bus, non-cache */
#define AMR2_DAT    0x00000fff    /* adr=0x200000000-0x2fffffff */
                                  /* asi=0x7 */
#define WSSR2_DAT   0x00001cf0    /* wait cycle=0x7, wait enable */


/*  CS3# area settings */
```

```
#define ARSR3_DAT   0xa0073000    /* 32bit-bus, non-cache */
#define AMR3_DAT    0x00000fff    /* adr=0x300000000-0x3fffffff */
                                  /* asi=0x7 */
#define WSSR3_DAT   0x00001cf0    /* wait cycle=0x7, wait enable */

/*  SDRAM-bus area settings */
/*  SDCS0# area settings */
#define SDARS0_DAT  0x00084000    /* 64bit-bus, cacheable     */
#define SDAM0_DAT   0x000303ff    /* adr=0x40000000-0x43ffffff */
                                  /* asi=0x8, 9, a, b /*
/*  SDCS1# area settings */
#define SDARS1_DAT  0x80084400    /* 16bit-bus, cacheable     */
#define SDAM1_DAT   0x000307ff    /* adr=0x44000000-0x47ffffff*/
                                  /* asi=0x8, 9, a, b /*
/*  register setting macros */
/*      10 - register address */
/*      11 - data to set */

/*  arsr getting macro */
#define SET_ARSR (reg_num. data)                              \
        sethi  %hi (ARSR_BASE+(reg_num)*8),              %10; \
        or     %10,  %lo(ARSR_BASE*(reg_num)8*), %10;    \
        sethi  %hi((data)),  %11  ;                           \
        or     %11,  %lo((data)),  %11;            \
        sta    %11,  [%10] SPL_REG_ASI

/*  amr setting macro /*
#define SET_AMR (reg_num. data)                          \
        sethi  %hi (AMR_BASE+(reg_num)*8), %10;          \
        or     %10,  %lo(AMR_BASE*(reg_num)8*),  %10;    \
        sethi  %hi((data)),  %11  ;                           \
        or     %11,  %lo((data)),  %11;            \
        sta    %11,  [%10] SPL_REG_ASI

/*  wssr setting macro /*
#define SET_WSSR (reg_num. data)                              \
        sethi  %hi (WSSR_BASE+(reg_num)*8),              %10; \
        or     %10,  %lo(WSSR_BASE*(reg_num)8*), %10;    \
        sethi  %hi((data)),  %11  ;                           \
        or     %11,  %lo((data)),  %11;            \
        sta    %11,  [%10] SPL_REG_ASI

/*  sdars setting macro /*
#define SET_SDARS (reg_num. data)                        \
        sethi  %hi (SDARS_BASE+(reg_num)*8),             %10; \
        or     %10,  %lo(SDARS_BASE*(reg_num)8*),        %10; \
        sethi  %hi((data)),  %11  ;                           \
        or     %11,  %lo((data)),  %11;            \
        sta    %11,  [%10] SPL_REG_ASI

/*  sdam setting macro /*
#define SET_SDAM (reg_num. data)                              \
        sethi  %hi (SDAM_BASE+(reg_num)*8),              %10; \
        or     %10,  %lo(SDAM_BASE*(reg_num)8*), %10;    \
        sethi  %hi((data)),  %11  ;                           \
        or     %11,  %lo((data)),  %11;            \
        sta    %11,  [%10] SPL_REG_ASI
```

```
/*  get psr.impl */
#define GETPSR_IMPL(reg)                                \
        rd     %psr, %reg;                              \
        srl    %reg, 28, %reg                           \

GETPSR_IMPL(12)

/*  SPARClite-bus area settings /*
SET_ARSR(0, ARSR0_DAT)/* CS0# area settings       */
SET_AMR(0, AMR0_DAT)
SET_WSSR(0, WSSR0_DAT)

SET_ARSR(1, ARSR1_DAT)/* CS1# area settings       */
SET_AMR(1, AMR1_DAT)
SET_WSSR(1, WSSR1_DAT)

SET_ARSR(2, ARSR2_DAT)/* CS2# area settings       */
SET_AMR(2, AMR2_DAT)
SET_WSSR(2, WSSR2_DAT)
```

```
          SET_ARSR(3, ARSR3_DAT)/* CS3# area settings       */
          SET_AMR(3, AMR3_DAT)
          SET_WSSR(3, WSSR3_DAT)

          subcc   $12, 0x1, %g0
          be      error
          nop

      sdram_860:
          SET_SDARS(0, ARSR0_DAT)     /* SDCS0# area settings */
          SET_SDAM(0, AMR0_DAT)

          SET_SDARS(1, ARSR1_DAT)     /* SDCS1# area settings */
          SET_SDAM(1, AMR1_DAT)

          ba,a    nsxt_stage

      error:

      next-stage:
```

2.6. SDRAM-IF

SDRAM-IF are modules that control data transfers between SDRAM and the processor. SDRAM access is made possible by setting the kinds of SDRAM in the system to match the system configuration and activating the SDRAM-IF. The following procedures must be followed in making these settings in order to use SDRAM-IF.

- Perform SDRAM area mapping using the SDARS and SDAM registers for the MB8686 and the SSAR and SAMR registers for the MB86861 (see *2.5.2. SDRAM Bus Area Settings*).
- Perform SDRAM settings CS by CS using the CSCR register (MB86861 only)
- Make ART register settings — set SDRAM refresh spaces.
- Set SCR Register — set SDRAM Operation Mode and activate SDRAM-IF
- Verify completion of SDRAM initialization — check SS Registers

2.6.1. ART Register

SDRAM refresh spaces are set by setting the ART.TC bit.

(Refresh spaces)=(ART.TC value)/(input clock (CLKIN) frequency)×(clock frequency multiple)

If input clock frequency = 33.3MHz, clock frequency multiple = 3 and SDRAM refresh spaces are set to 15μ sec, then

$$\text{ART.TC} = 15 \times 10^{6} \times (33.3 \times 10^{6} \times 3) = 1498 = \text{0x5da}$$

2.6.2. SCR Register

Make settings in keeping with SDRAM type and SDRAM configuration.

- Set SCR.CL, SDCFG.ST, SDCFG, SDCFG.PRC in accordance with the SDRAM standard in use.
- Set SDRAM bus width to SCR.BS,
- Use SCR.SPC and SDCFG.SP to set parity functions.
- Activate SDRAM-IF by setting SCR.SE = 1.

2.6.3. Verifying Completion of SDRAM Initialization

Following SDRAM activation, SDRAM-IF performs SDRAM initialize operation. Completion of SDRAM initialize operations can be verified by the SSR.SD bit. Be sure to veryfy that SSR.SDI=1 before accessing SDRAM.

SDRAM-IF setting sample codes are shown below.

list2-3 SDRAM Set / Activate (MB86860)

```
#define  SPL_860_ASI   0x4
#define  ART_ADR       0x80000808
#define  ART_DAT       0x000005da
#define  SCR_ADR       0x80000800
#define  SCR_DAT       0x000006dd
#define  SCR_ADR       0x80000810
#define  SDI_FLG       0x1

/* set ART register               */
sethi  %hi(ART_ADR), %l0
or     %l0, lo(ART_ADR), %l0
sethi  %hi(ART_DAT), %l1
or     %l1, %lo(ART_DAT), %l1
sta    %l1, [%l0] SPL_860_ASI

/* set sdram control register        */
sethi  %hi(SCR_ADR), %l0
or     %l0, lo(SCR_ADR), %l0
sethi  %hi(SCR_DAT), %l1
or     %l1, %lo(SCR_DAT), %l1
sta    %l1, [%l0] SPL_860_ASI

/* check ssr register                */
sethi  %hi(SSR_ADR), %l0
or     %l0, lo(SSR_ADR), %l0

/* check ssr.sdi bit                 */
check_ssr:
       lda    [%l0] SPL_860_ASI, %l1
       andcc  %l1, SDI_FLG, %g0
       be     check_ssr
       nop

/* SDRAM initialization completed  !                   */
go_next:
       ....
```

2.7. Stack Frame Creation

It is always possible that traps may occur during user program operations. For this reason, stack pointers must always be set correctly in order safely to save register windows. Moreover, for the same reason, when using register windows, 1 window should be left empty.

The following registers are used as stack pointers in the SPARC architecture:

- %sp(Stack Pointer) — %o6(%14) ... current window stack pointer
- %fp(Frame Pointer) — %i6(r30) ... previous window stack pointer

list2-4 Stack Frame Creation

```
#define  NWINDOWS           8
#define  SDRAM_END          0x48000000
#define  DEFAULT_STACK_FRAME (NWINDOWS*16)

sethi    %hi(SDRAM_END), %fp               /* Initialize frame pointer
*/
```

```
          add    %fp, DEFAULT_STACK_FRAME, %sp                    /* Allocate
          stack frame */

          save   %sp, DEFAULT_STACK_FRAME, %sp
```

2.8. Cache Initialization

In SPARClite SS processors, a 16Kbyte-4 way set associative (D-cache is write through) cache is built in together with I-cache and D-cache. In order validly to take advantage of cache functions inside the core, a data buffer part is provided between BIU (Bus interface Units). When using cache functions, the data buffer part must also be enabled.

2.8.1. Cache Initialization Flow

The following register settings are made for cache/buffer control.

- Cache Control — ICCR (ARSR31) Register
- Buffer Control — BCR (Buffer Control Register)

In Figure 2-7 an example of cache initialization flow, and in list2-5 an example of sample code is shown.

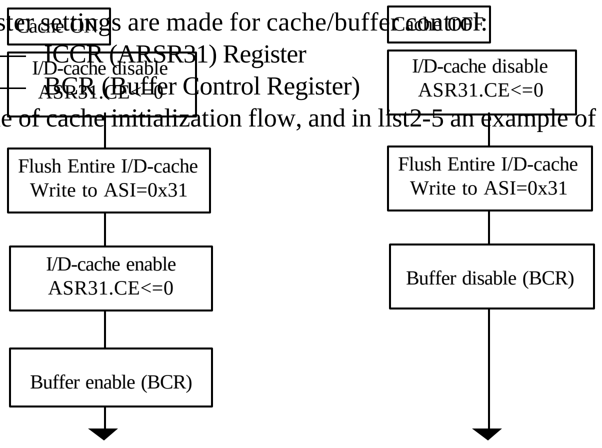


Figure 2-7 Cache Initialization Flow

list2-5 Example of Cache Control Routine

```
#define  BCR_ADR     0x80000000
#define  BCR_DAT     0x8000003f

/* Cache on  */
cache_on:
        wr   %g0, %g0,  %asr31        /* cache off */
        nop
        nop
        nop
        sta    %g0,   [%g0+%g0] 0x31  /* I/D cache flush all */
        nop
        nop

        wr   %g0, 0x3,  %asr31 /* cache on */
        nop
        nop
        nop
        sethi  %hi(BCR_ADR),%g1
        or      %g1, %lo(BCR_ADR), %g1
        or      %g0,          BCR_DAT, %g2
        sta     %g2, [%g1] SPL_860_ASI

        nop

        impl    %o7+8, %g0
        nop

/* Cache off  */
cache-off:
        sethi  %hi(BCR_ADR), %g1
        sta  %g0, [%g1]4            /* Buffer disable */
        nop

        wr   %g0, %g0, %asr31      ! cache off : arsr31 <= 0
        nop                        ! IFTD <= 0, CE <= 0

        impl    %o7+8, %g0
        nop
```

3. TRAPS

When interrupts and traps occur in SPARC processors, a move is made to the control vectors specified by the trap table. The first 4 instructions of each trap handler are contained in the trap table. In this chapter, methods of creating and examples of creating trap tables and trap handlers are shown.

3.1. Operations when Traps are Generated

When PSR.ET=1, the following processor operations are performed by traps:

- Disable Traps (PSR.ET <= 1)
- Preserve the mode at trap time (PSR.S) in PSR.PS. (PSR.PS <= PSR.S)
- Change mode to Supervisor Mode (PSR.PS <= 1)
- Advance register window to new window. (CWP<=((CWP-1)modNWINDOWS)
- Save program counter value when trap occurred to new windows %11 and %12. (r[17]<=PC, r[18]<=nPC)
- Write special values identifying exceptions and interrupt requests to TBR.tt bit. (except reset)

When PSR.ET=0, the processor goes into ERROR mode if a trap occurs, asserts ERROR# signals and stops. External interrupts are igno

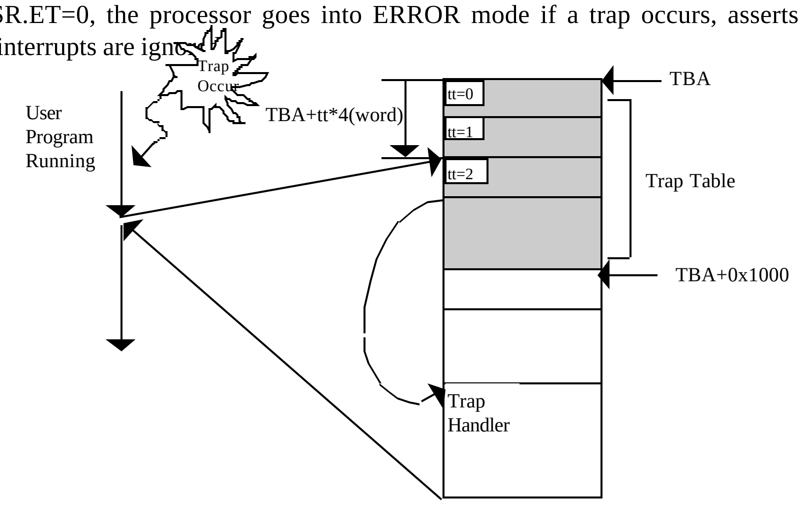


Figure 3-8 Trap Operations

3.2. Trap Tables

Trap tables consist of 4 words × 256 = 1Kword, with the TBA (Trap Base Address) as the top address. Trap vectors corresponding to each trap type contain the first 4 words of the service routine. Control is moved to trap handlers through these vectors.

Figure 3-9 Trap Table Example

```
_start:
_traptbl:
/* tt = undefined */
reset_trap:
       rd     %psr, %10        /* %10=psr at trap time.*/
       rd     %tbr, %13        /* %13=tbr at trap time.*/
       ba     _reset_entry     /* Branch to the reset trap
       handler.*/
       nop
```

```
/* tt = 1 */
instr_access_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _instr_access
/* tt = 2 */
illegal_instr_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _illegal_instr
/* tt = 3 */
privil_instr_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _privil_instr
/* tt = 4 */
fp_disable_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _fp_disable
/* tt = 5 */
win_ovf_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba     _win_ovf
       nop
/* tt = 6 */
win_unf_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba     _win_unf
       nop
/* tt = 7 */
mem_misalign_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _mem_misalign
/* tt = 8 */
fp_exception_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _fp_exception
/* tt = 9 */
data_access_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _data_access
/* tt = 10 */
tag_ovf_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _tag_ovf
/* tt = 11 */
       nop
       nop
       nop
       nop
/* tt = 12 */
       nop
       nop
       nop
       nop
/* tt = 13 */
       nop
       nop
       nop
       nop
/* tt = 14 */
       nop
       nop
       nop
       nop
/* tt = 15 */
       nop
       nop
       nop
       nop
/* tt = 16 */
       nop
       nop
       nop
       nop
/* tt = 17 */
int_1_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _int_1
/* tt = 18 */
int_2_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _int_2
/* tt = 19 */
int_3_trap:
       rd     %psr, %l0        /* %l0=psr at trap time.*/
       rd     %tbr, %l3        /* %l3=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _int_3

       . . .

       . . .
```

```
/* tt = 253 */
Ticc_126_trap:
       rd     %psr, %10        /* %10=psr at trap time.*/
       rd     %tbr, %13        /* %13=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _Ticc_126
/* tt = 254 */
Ticc_127_trap:
       rd     %psr, %10        /* %10=psr at trap time.*/
       rd     %tbr, %13        /* %13=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _Ticc_127
/* tt = 255 */
emulation_bp_trap:
       rd     %psr, %10        /* %10=psr at trap time.*/
       rd     %tbr, %13        /* %13=tbr at trap time.*/
       ba,a   _chk4ovflo       /* check for reg window overflow*/
       ba,a   _emulation_bp
```

[Note]: The chk4ovflo routine checks whether a window overflow has occurred or not when a trap occurs. For window overflow/underflow checks not to be run when traps occur, window overflow checks must be run inside each trap handler.

3.4. Trap Handlers

3.4.1. Processing Inside Trap Handlers

The followig things must be done in all trap handlers:

- Verify whether a window can be used or not when a new trap occurs. (When a trap occurs, the processor automatically saves the interrupted routine window by decrementing the Current Window Pointer.)
- Re-enable traps by setting PS ET bits.
- Process exception conditions which generate traps.
- Disable traps by clearing PS ET bits.
- Execute IMPL/RETT instruction pairs. Return addresses are kept in r[17].
- RETT instructions automatically select ET=1
  The return sequence for returns from traps is as follows:
- When re-running trapped instructions:

```
jmpl     %r17, %g0
rett     %r18
```

- When re-starting execution starting from the next instruction after a trapped instruction:

```
jmpl     %r18, %g0
rett     %r18+4
```

3.5. Trap Handler Examples

3.5.1. Window Overflows/Underflows

In this section, examples are shown of window overflow/underflow handlers in SPARC processors which are essential in normal use.

list3-1 Window Overflow Check Routine

This routine checks whether or not there are window overflows in trap handlers, and if a window overflow happens, saves the oldest window in the register window to memory.

```
/* FUNCTION */
/*     _chk4ovrflo */

/* DESCRIPTION */
/*     This code is branched to before each trap (except reset */
/*     _win_unf, and _win_ovf) handler. */
/*     It checks to see if we have moved into the invalid window */
/*     and performs fixup ala _win_ovf. */

/* INPUTS *7
/*     – $10 = psr at trap time */
/*     – %11 = pc at trap time */
/*     – %12 = npc at trap time */
/*     – %13 = tbr at trap time */

/* INTERNAL DESCRIPTION */

/* RETURNS */
/*     _None. */

_chk4ovflo:
       and    %10, 0x1F. %15        /* get the cwp */
       set    1, $14                /* compare the cwp with the wim
       */
       rd     %wim   %16            /* read the wim */
       nop
       nop
       nop
       sll    $14, %15, %14         /* compare */
       andcc  %14, %16, %g0
       bz     _ret2ttbl             * not valid window, just return
       */
       nop

                                    /* in line version of _win_ovf */
#ifndef INT_MODE1
       or     %10, 0xf20, %17       /* enable traps, disable interrupts
       */
       wr     %17. %g0, %psr
#endif
       or     %g0, %g1, %17         /* Save %g1. */
       srl    %16, 1, %g1           /* Next WIM = %g1 = */
                                    /* rol(WIM, 1, NWINDOW). */
```

```
        sll    %16, NWINDOWS-1, %15
        or     %15, %g1, %g1
        save                           /* Get into window to be saved. */
        wr     %g1, %g0, %wim          /* install new wim */
        nop                            /* must delay three instructions */
        nop                            /*  before  using  these  registers,  so
        */
        nop                            /* put nops in just to be safe. */
                                       /* save all local registers */

        std    %10, [%sp + 0x0 * 4]
        std    %12, [%sp + 0x2 * 4]
        std    %14, [%sp + 0x4 * 4]
        std    %16, [%sp + 0x6 * 4]
        std    %i0, [%sp + 0x8 * 4]
        std    %i2, [%sp + 0xa * 4]
        std    %i4, [%sp + 0xc * 4]
        std    %i6, [%sp + 0xe * 4]

        restore                                     /* Go back to trap
window */
        or     %g0, %17, %g1           /* Restore %g1. */

_ret2ttbl:
        /* It is safe now to allocate a stack frame for this window */
        /* because all overflow handling will have been accomplished */
        /* in the event we trapped into the invalid window */
        /* i.e. all of this window %o regs (next window %i regs) */
        /* will have been safely stored to the stack before we overwrite
%sp */

        add    %fp, DEFAULT_STACK_FRAME, %sp
        or     %13, 0xc, %13 /* add 0xc to %tbr value to obtain */
                             /* ttbl instruction addr we came from + 0x4 */
        jmpl   %13, %g0      /* jump back into table at aforementioned
addr.*/
        nop
```

list3-2 Window Underflow Check Routine

This routine checks to see whether or not a window underflow has occurred when returning from a trap, and if an underflow haa occurred saves the RESTORE destination window to memory.

```
/* FUNCTION */
/*     _chk4uflo */

/* DESCRIPTION */
/*     This code is branched to before each trap (except reset */
/*     _win_unf, and _win_ovf) handlers. */
/*     It  checks  to  see  if  we  are  about  to  move  to  the  invalid
window */
/*     on ensuing rett instruction and performs fixup ala _win_unf.
*/

/* INPUTS */
/*     – $10 = psr at trap time */
/*     – %11 = pc at trap time */
/*     – %12 = npc at trap time */
/*     – %13 = tbr (ord w/ 0xc) at trap time */

/* INTERNAL DESCRIPTION */

/* RETURNS */
/*     _None. */

_chk4uflo:
/* Test that the window we are moving into is valid */
       and    %10, 0x1F. %15         /* get the cwp */
       subcc  %14, (NWINDOWS-11), %g0             /* test for being in
       high window */
       be     win7
       nop
       add    $14, 1, %14   /*   compare   with   the   next   highest
       window*/
       ba     not7
       nop
win7:  or     %g0, %g0, %14
not7:  or     %g0, 0x1, %15          /* compare cwp with the wim. */

       sll    %15, %14, %15          /* if they are the same then */
       rd     %wim   %16             /* read the wim */
       nop
       nop
       nop
       andcc  %15, %16, %g0
       ba     _trap_return
       nop
       ba     _win_unf
       nop
```

3.5.2. Window Overflow Handlers

Window overflow handlers save the oldest windows to memory when window overflows occur.

```
/* FUNCTION*/
/*     _win_ovf*/

/* DESCRIPTION*/
/*     This routine is the trap handler for register overflow trap.*/
/*     Priority: 0x06*/
/*     Upon entry, the cwp points to the trap window, which is 1 less
than*/
/*     the register window that must be saved to the stack.  The stack is*/
/*     organized with %i6 = $o6 - (0x40 + local stack used).  The ins and*/
/*     locals are saved, and the wim is adjusted for the new window.*/

/* INPUTS*/
/*     - %10=psr at trap time*/
/*     - %11=pc at trap time*/
/*     - %12=npc at trap time*/
/*     - %13=tbr at trap time*/

/* INTERNAL DESCRIPTION*/
/*     - Move the invalid window to the next window by rotating the %wim*/
/*          register left by one slot*/
/*      - Get into the previously invalid window, the one that caused the
trap,*/
/*          and save all of the registers in it.*/
/*      - Get back into the previously valid window and let the trapped
routine*/
/*          execute the save again.*/

/* RETURNS*/
/*      - %17 = 1 so execution starts at the trapped instruction.*/

-win-ovf:
       /* Turn on traps again so that we do not go into error state.*/
#ifindef INT_MODE1
       or     %10, 0xf20, %17          /* enable traps, disable interrupts.*/
       wr     %17, %g0, %psr
#endif
       rd     %wim %14                 /* Get wim at trap time.*/
       or     %g0, %g1, %17            /* Save %g1.*/
       srl    %14, 1, %g1              /* Next WIM = %g1 =*/
                                       /* rol(WIM, 1, NWINDOW).*/
       sll    %14, NWINDOWS-1, %15
       or     %15, %g1, %g1
       save                            /* Get into window to be saved.*/
       wr     %g1, %g0, %wim           /* Install new wim.*/
       nop                             /* must delay three instructions */
       nop                             /* before using these registers, so */
       nop                             /* put nops in just to be safe. */
                                       /* save all local registers */
       std    %10, [%sp + 0x0 * 4]
       std    %12, [%sp + 0x2 * 4]
       std    %14, [%sp + 0x4 * 4]
       std    %16, [%sp + 0x6 * 4]
       std    %i0, [%sp + 0x8 * 4]
       std    %i2, [%sp + 0xa * 4]
       std    %i4, [%sp + 0xc * 4]
       std    %i6, [%sp + 0xe * 4]

       restore                                        /* Go back to trap
window.*/
       or     %g0, %17, %g1            /* Restore %g1.*/
       or     %g0, 1, %17              /* Tell _trap_return to rerun
trapped*/
       ba     _trap_return             /* instruction.*/
       nop
```

3.5.3. Window Underflow Handlers

Window underflow handlers save restore destination windows to memory when window underflows occur.

```
/* FUNCTION*/
/*      _win_unf*/

/* DESCRIPTION*/
/*      This routine is the trap handler for register overflow trap.*/
/*      Priority: 0x07*/
/*      Upon entry, the cwp points to the trap window, which is 1 more
than*/
/*      the register window that must be restored from the stack.  The stack
is*/
/*      organized with %i6 = $o6 - (0x40 + local stack used).  The ins and*/
/*      locals are restored, and the wim is adjusted for the new window.*/

/* INPUTS*/
/*      - %10=psr at trap time*/
/*      - %11=pc at trap time*/
/*      - %12=npc at trap time*/
/*      - %13=tbr at trap time*/

/* INTERNAL DESCRIPTION*/
/* RETURNS*/
/*      - %17 = 1 so execution starts at the trapped instruction.*/

-win-unf:
        /* Turn on traps again so that we do not go into error state.*/
#ifindef INT_MODE1
        or      %10, 0xf20, %17         /* enable traps, disable interrupts.*/
        wr      %17, %g0, %psr
#endif
        rd      %wim %14                /* Get wim.*/
        nop
        nop
        nop
        sll     %14, 1, %15             /* Next WIM = rol(WIM. 1, NWINDOW).*/
        srl     %14, NWINDOWS-1, %16
        or      %16, %15, %16
        mov     %16, %wim               /* Install it.*/
        nop                             /* must delay three instructions */
        nop                             /* before using these registers, so */
        nop                             /* put nops in just to be safe. */
                                        /* Back to user window.*/
                                        /* Get into window to be restored.*/
                                        /* restore registers from stack*/

             subcc  %13, 0x6, %g0
              bne   notunf1
              restore
              restore
 notunf1:
        ldd      [%sp + 0x0 * 4], %10
        ldd      [%sp + 0x2 * 4], %12
        ldd      [%sp + 0x4 * 4], %14
        ldd      [%sp + 0x6 * 4], %16
        ldd      [%sp + 0x8 * 4], %i0
        ldd      [%sp + 0xa * 4], %i2
        ldd      [%sp + 0xc * 4], %i4
        ldd      [%sp + 0xe * 4]. %i6
              subcc   %13, 0x6, %g0
               bne  notunf2
        save                              /* Get back to original window.*/
        save
notunf2:
        or      %g0, 1, %17             /* Tell _trap_return to rerun
trapped*/
        ba      _trap_return            /* instruction.*/
        nop
```

3.5.4. Trap Return Routines

Trap return routines return from traps to programs executing processing. Before executing a RETT instruction, interrupt disable status (PSR.ET=1) must be in effect.

```
/* FUNCTION*/
/*      _trap_return*/

/* DESCRIPTION*/
/*      This routine is called by all traps except reset to perform some*/
/*      common setup tasks before returning from a specific handler.  It*/
/*      is a leaf procedure.*/

/* INPUTS*/
/*      - %17 = rerun instruction flag*/
/*      - %10 = psr upon return*/
/*      - %11 = pc at trap time*/
/*      - %12 = npc at trap time*/

/* INTERNAL DESCRIPTION*/
/*      - Determine whether to start exectuion at pc or npc.*/
/*      - Disable traps.*/
/*      - Prepare cache autolock return sequence*/
/*      - Return from trap or pc or npc.*/

/* RETURNS*/
/*      - None.*/

_trap-return
        subcc     %17,1, %g0             /* If trap handler returns 1, then*/
        be        _rerun_trap_instr      /* rerun the trapping instruction*/
        nop                              /* else do not.*/
        ba        _skip_trap_instr
        nop

/* Return routine for executing the trapped instruction like for page
faults.*/[1]

_rerun_trap_instr:
        andn      %10, 0x20, %10         /* Disable traps*/
        wr        %10, %g0, %psr
        /* or        %g0, 0x1, %17 */          /* Set Restore Lock bit,*/
        /* or        %g0, 0x10, %10 */         /* in case an autolock
sequence*/
        /* sta       %17, [%10] 1 */           /* is in effect.*/
        jmpl      %11, %g0                /* Return to instruction at pc.*/
        rett      %12+%g0

/* Return routine for skipping the trapped instruction.*/
```

[1] Translator's Note: All code in this document was in English in the Japanese original and is copied exactly as is. The English contained therein is the programmer's, not the translator's.

```
_skip_trap_instr:
       andn      %10, 0x20, %10         /* Disable traps.*/
       wr        %10, %g0, %psr
       jmpl      %12, %g0               /* Return to instruction at npc.*/
       rett      %12+4
```

4. Low Power Consumption Modes

SPARClite SS processors have 2 operating modes for low power consumption: SLEEP Mode and STOP Mode. In this chapter, the operating sequence when shifting operating modes will be discussed from a program standpoint.

4.1. Operating Modes

Operating Mode status transitions are shown in Figure 4-10

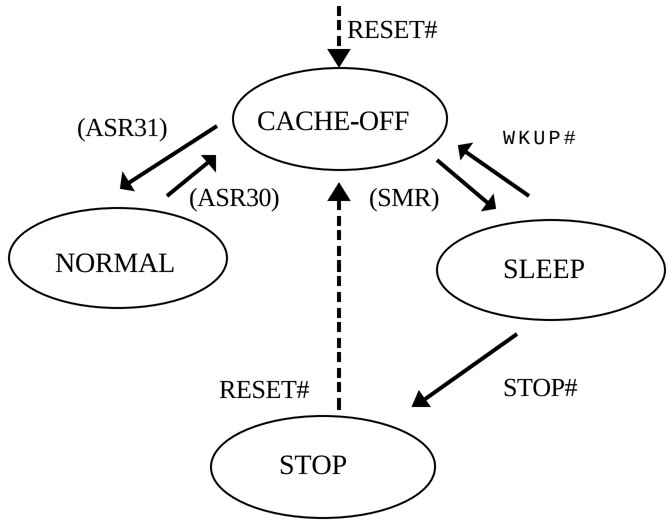


Figure 4-1 SPARClite SS Processor Operating Status Transitions

• CACHE-OFF Mode	—	Operating Mode after a RESET cancel.
• NORMAL Mode	—	Operating Mode during Cache operation.
• SLEEP Mode	—	Stops clocks except PLL.
• STOP Mode	—	Stops all internal clocks RESET# input is required to restart operation.

4.2. SLEEP Mode

In order to shift to SLEEP Mode, the previous processor operating modes, CACHE-OFF and CACHE, must be flushed. Sleep Mode is entered by writing "1" to the SLP bit of the Sleep Mode Register (SMR) (ASI=0x1, address=0x00002004). See Figure 4-11.

4.3. STOP Mode

In Sleep Mode, the processor enters STOP Mode and stops operating by asserting external pin STOP. Operation is only restarted from Stop Mode by a RESET# input .

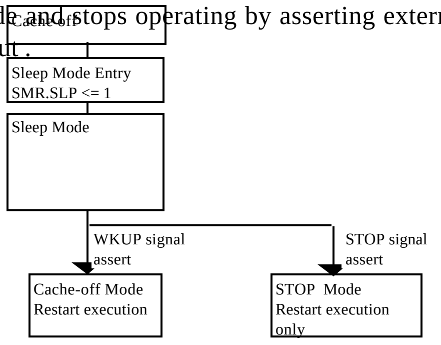


Figure 4-11 Low Power Consumption Mode Operating Flow