SUNPLUS

S^+core 7 Processor Core Technical Reference Manual (for Software Use, Fixed-MMU)

V1.0 – Feb. 22, 2005

19, Innovation First Road, Hsinchu Science Park, Taiwan 300, R.O.C.

886-3-578-6005 886-3-578-4418 www.sunplus.com.tw

0 Table of Content

Figure

Table

1 Document Revision History

Revision	Date	By	Remark
1.0	2005/02/22	June-Yuh Wu	First Edition

2 Introduction

2.1 Introduction

S^{+}core is a 32-bit RISC with Sunplus-owned instruction set architecture (ISA). The ISA has 32/16-bit hybrid instruction mode and parallel conditional execution (patent pending) for high code density, high performance and versatile applications. The microarchitecture includes AMBA bus for SoC integration, coprocessor and custom engine interface for function flexibility, and SJTAG for efficient testing and debugging.

2.2 Key Features

The features of S^{+}core are listed as following:

- 32/16-bit Hybrid Execution Mode
- Parallel Conditional Execution (patent pending)
- Capability for Further Software Security Design (patent pending)
- A Harvard Architecture (I-Cache / D-Cache) Solution
- Fixed-MMU (Fixed mode)
- Compliance to the AMBA Specification (Rev 2.0) for easy integration into SOC implementation
- Vectored Interrupt
- SJTAG (Sunplus JTAG)

2.3 Instruction Set

2.3.1 32-Bit Instructions

S^{+}core 32-bit instructions can be divided into the following functional categories:

- Load and store instructions
- Data processing instructions
  - (1) Arithmetic instructions
  - (2) Logical instructions
  - (3) Shift/rotate instructions
  - (4) Extension instructions
  - (5) Move instructions
- Custom engine instructions
- Jump and branch instructions
- Special instructions

(1) System control instructions

(2) Cache instructions

(3) Debug instructions

(4) Control register instructions

- Coprocessor instructions

(1) Coprocessor register transfer instructions

(2) Coprocessor memory access instructions

(3) Coprocessor operation instructions

2.3.2 16-Bit Instructions

S$^+$core 16-bit instructions can be divided into the following functional categories:

- Load and store instructions
- Data processing instructions

(1) Arithmetic instructions

(2) Logical instructions

(3) Shift instructions

(4) Move instructions

- Jump and branch instructions
- Special instructions

(1) System Control Instructions

(2) Debug Instructions

3 Programming Model

3.1 Data Types

S$^+$core supports the following data types:

- ***Byte:*** 8 bits.
- ***Halfword:*** 16 bits.
- ***Word:*** 32 bits.

Table **3-1** lists the supported data types and ranges.

Table 3-1 Data types

Data Type		Range	Note
Byte	Signed	$-2^7 \sim +2^7-1$	
	Unsigned	$0 \sim +2^8-1$	
Halfword	Signed	$-2^{15} \sim +2^{15}-1$	Must be aligned to halfword boundary
	Unsigned	$0 \sim +2^{16}-1$	
Word	Signed	$-2^{31} \sim +2^{31}-1$	Must be aligned to word boundary
	Unsigned	$0 \sim +2^{32}-1$	

All data processing operations, for example ADD, are performed on word quantities.
Load and store operations can transfer bytes, halfwords and words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded.
The 32-bit instructions are exactly one word and are aligned on a four-byte boundary. The 16-bit instructions are exactly one halfword and are aligned on a two-byte boundary.

3.2 Processor Modes

S+core supports three processor modes:

- ***User Mode:*** Used by application software or operation system software. The CPU normally operates in User Mode until an exception is detected to force it into Kernel Mode. While the processor is in user mode, the program being executed is unable to access some protected system resources.
- ***Kernel Mode:*** Used exclusively by operation system. When the processor enters Kernel Mode by exception, it remains in Kernel Mode until a Return From Exception (RTE) instruction is executed.
- ***Debug Mode:*** Used by debug service routine. In this mode, the program has full access to User/Kernel mode register sets and some other debug registers.

3.3 Register Set

S^+core has the following registers:

- 32 General Purpose Registers
- 2 Custom Engine Registers (CEH/CEL)
- 3 Special Purpose Registers
  - Sr0: Loop Counter Register (CNT)
  - Sr1: Load Combine Register (LCR)
  - Sr2: Store Combine Register (SCR)
- 17 System Control Registers

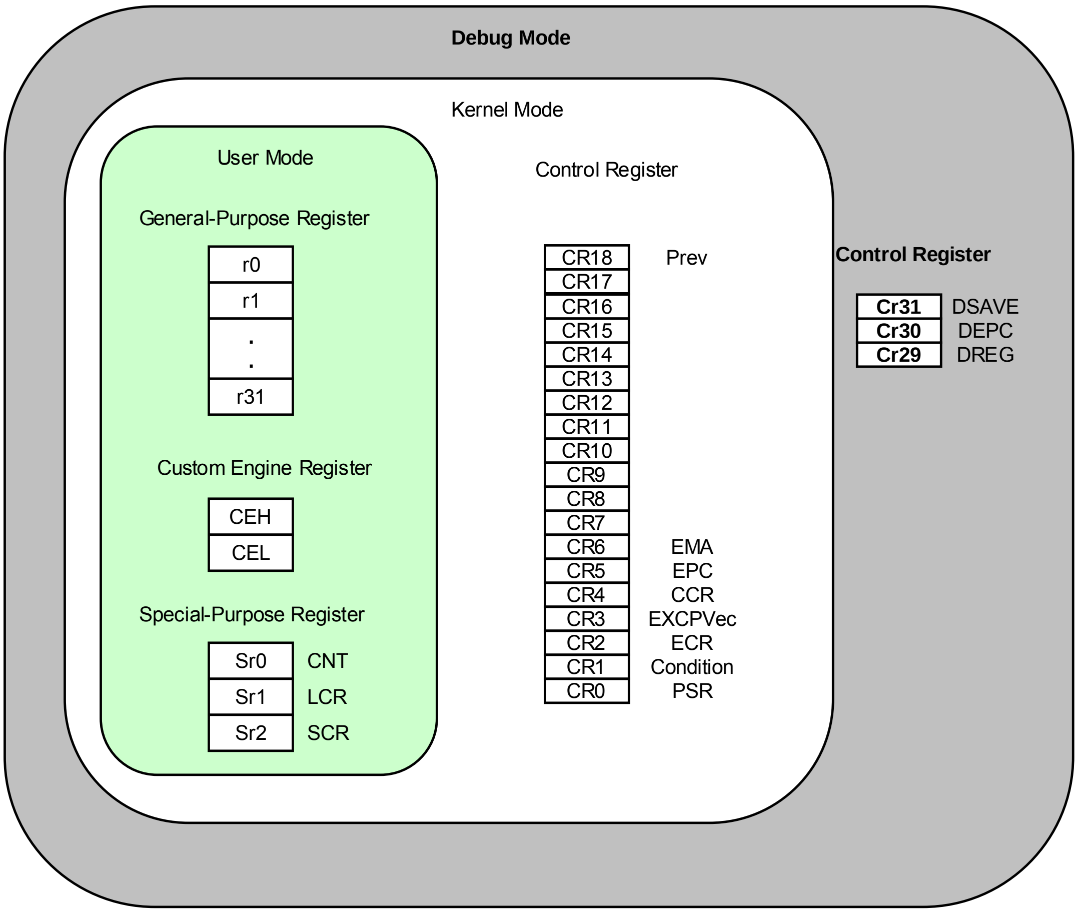


Fig 3-1 S^+core Register Set

3.3.1 Register Set Overview

User Mode

Used by application software or operation system software. In this mode, the program can access 32 GPRs, CEH/CEL registers and three special purpose registers (CNT, LCR, STR).

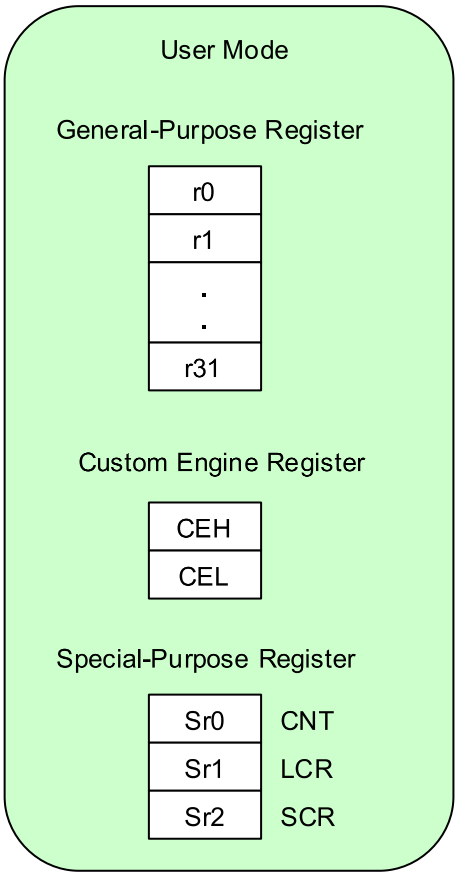


Fig 3-2 User Mode Register Set

Kernel Mode

In this mode, the program can access all User Mode register sets and system registers (CR0~CR18)

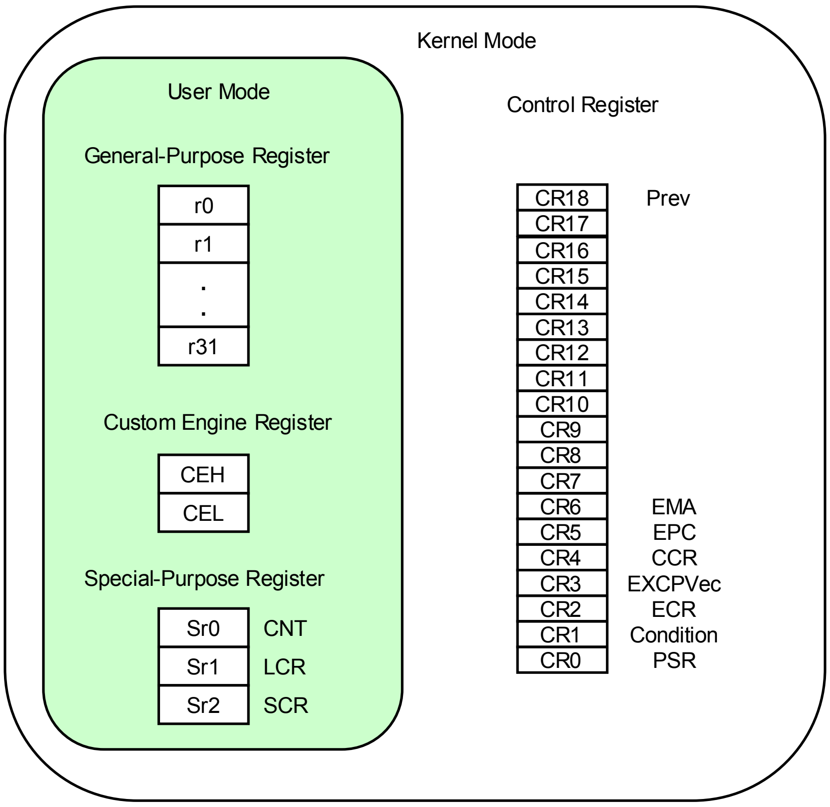


Fig 3-3 Kernel Model Register Set

Debug Mode

In this mode, the program can access User/Kernel mode register sets and three debug registers (DSAVE, DEPC, DREG).

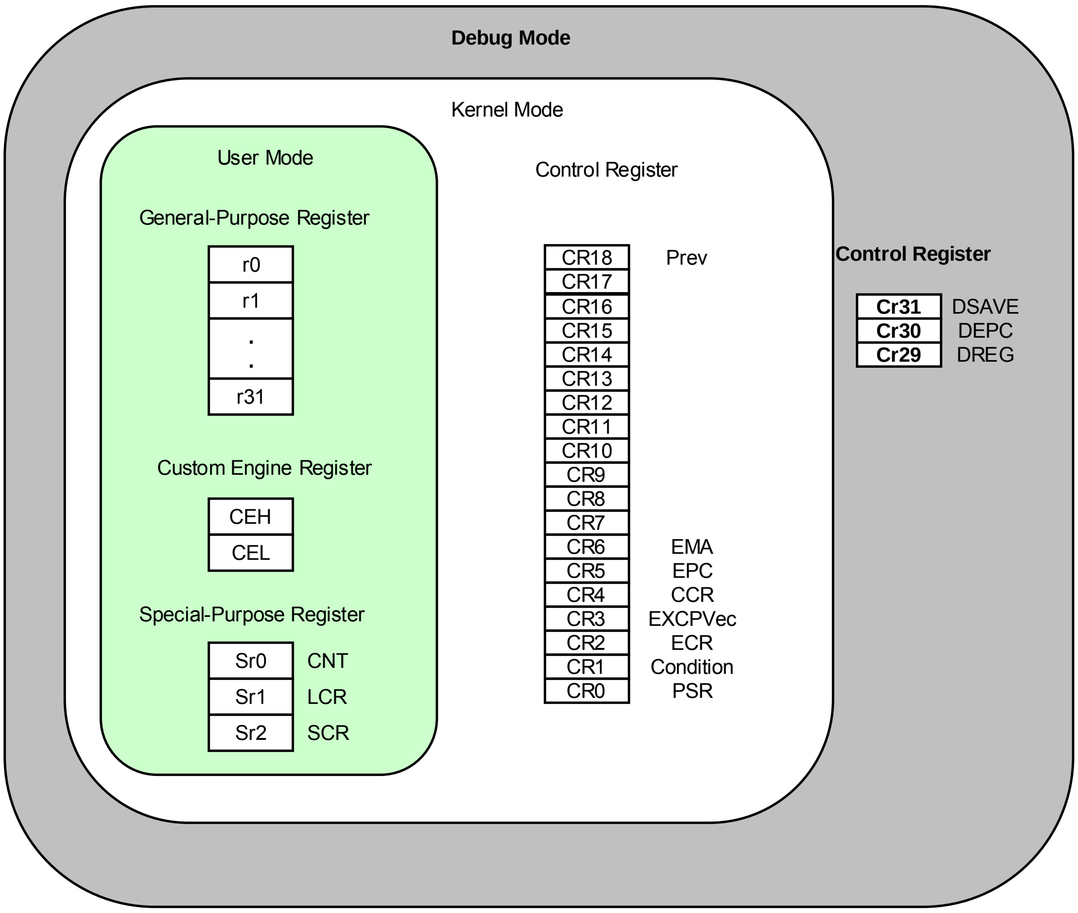


Fig 3-4 Debug Mode Register Set

3.3.2 General Purpose Registers (GPRs)

The S+core processor has thirty-two 32-bit general-purpose registers (r0~r31). In 32-bit instruction mode all of these GPRs can be accessed. Due to the restriction of the instruction encode, normally the 16-bit instruction mode only can access the lower sixteen registers (r0~r15). The general register r3 is used as link register for branch/jump and link instruction.

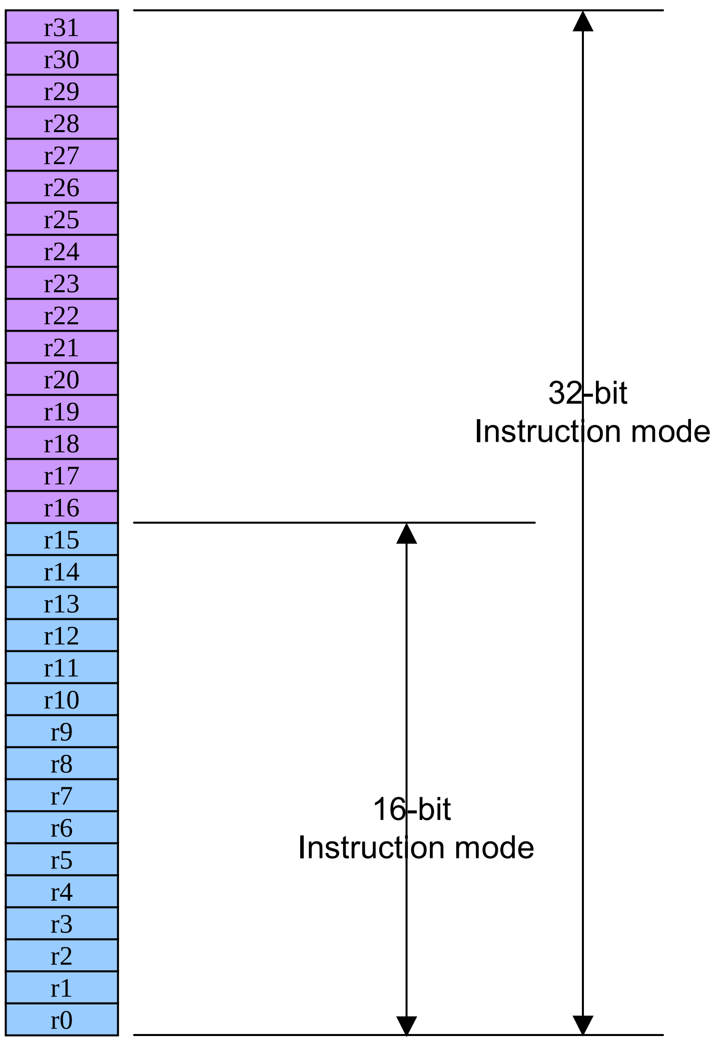


Fig 3-5 General Purpose Registers

3.3.3 Custom Engine Registers

The CEH and CEL registers store the higher and lower word of multiplication and division. After the multiplication operation completes, the high order word of the double result is loaded into CEH register, and the low order word is loaded into CEL register. For division, the quotient and remainder words of the double result are stored into the CEL and CEH registers, respectively.

The content of these two registers can be transfer to or from general purpose registers by MFCEH, MFCEL, MFCEHL, MTCEH, MTCEL and MTCEHL instructions.

3.3.4 Special Purpose Registers

S+core has three special purpose registers: CNT (Sr0), LCR (Sr1) and SCR (Sr2). The counter register (CNT) is a 32-bit register. CNT register holds a loop count that would be decremented during execution of branch instructions that contain an appropriately coded BC field. When executing *bcnz* instruction and the CNT register is not equal to zero, CNT register would be decremented by one and branch to the target address. If CNT register is equal to zero, the *bcnz* instruction behaves like *nop* instruction and CNT register is unchanged. Load combine register (LCR) and store combine register (SCR) are used in unaligned load and store operation.

3.3.5 Control Register (CR)

Table 3-2 System Register

Name	control register No.
PSR	0
Conditional	1
ECR	2
EXCPVec	3
CCR	4
EPC	5
EMA	6
-	-
Prev	18
DREG	29
DEPC	30
DSAVE	31

Program Status Register (CR0)

31 29	28	27 24	23 16	15	14 6	5	4	3	2	1	0
CU[2:0]	CRA	-	IM[7:0]	Endian	-	UMb	IEb	UMs	IEs	UMc	IEc

Fig 3-6 Program Status Register Format

Table 3-3 Program Status Register Field Description

Bits	Name	Description	R/W	Reset
31:29	**CU**	CU[n] = 1 (0) indicates the coprocessor n+1 is usable (unusable) in coprocessor instructions	R/W	0
28	**CRA**	CRA = 1 (0) indicates the control register is accessible (non-accessible) and control register instruction is executable (non-executable).	R/W	0
27:24	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
23:16	**IM**	Interrupt masks for the six priority encoded hardware interrupts (IM[7:2]) and two software interrupts (IM[1:0])	R/W	0
15	**Endian**	Endian = 0 (1), indicates LittleEndian (BigEndian)	R	big_en
14:6	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
5	**UMb**	Backup of saved user mode bit. UM = 1 (0) indicates user (kernel) mode	R/W	0
4	**IEb**	Backup of saved interrupt enable. IE = 1 (0) indicates that sixty-three hardware interrupts and two software interrupts are enabled	R/W	0
3	**UMs**	Saved mode bit. UM = 1 (0) indicates user (kernel) mode	R/W	0
2	**IEs**	Saved interrupt enable. IE = 1 (0) indicates that sixty-three hardware interrupts and two software interrupts are enabled	R/W	0
1	**UMc**	Current mode bit. UM = 1 (0) indicates user (kernel) mode	R/W	0
0	**IEc**	Current interrupt enable. IE = 1 (0) indicates that sixty-three hardware interrupts and two software interrupts are enabled	R/W	0

UMb, IEb, UMs, IEs, UMc, IEc fields form a three-level hardware stack UM/IE signal. The current values are KUc/IEc, the saved previous values are UMs/IEs, and the backups of saved values are UMb/IEb.

The IM[7:2] is a 6-bit encoded value in the range 0, 1, …63. A value 0 indicates that no interrupt request are masked. The value n represents that priority level n and below n is masked. The IM[1:0] is a software interrupt mask, IM[1]/IM[0] = 1 indicates software interrupt 1/0 is masked by IM[1:0].

Condition Register (CR1)

31	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-		Tb	Nb	Zb	Cb	Vb	Ts	Ns	Zs	Cs	Vs	Tc	Nc	Zc	Cc	Vc

Fig 3-7 Condition Register Format

Table 3-4 Condition Register Field Description

Bits	Name	Description	R/W	Reset
31:15	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
14	**Tb**	Backup of saved Conditional parallel execution T flag	R/W	0
13	**Nb**	Backup of saved Negative/Less Than flag	R/W	0
12	**Zb**	Backup of saved Zero flag	R/W	0
11	**Cb**	Backup of saved Carry/Borrow/Extend flag	R/W	0
10	**Vb**	Backup of saved Overflow flag	R/W	0
9	**Ts**	Saved Conditional parallel execution T flag	R/W	0
8	**Ns**	Saved Negative/Less Than flag	R/W	0
7	**Zs**	Saved Zero flag	R/W	0
6	**Cs**	Saved Carry/Borrow/Extend flag	R/W	0
5	**Vs**	Saved Overflow flag	R/W	0
4	**Tc**	Current Conditional parallel execution T flag	R/W	0
3	**Nc**	Current Negative/Less Than flag	R/W	0
2	**Zc**	Current Zero flag	R/W	0
1	**Cc**	Current Carry/Borrow/Extend flag	R/W	0
0	**Vc**	Current Overflow flag	R/W	0

Tb, Nb, Zb, Cb, Vb, Ts, Ns, Zs, Cs, Vs, Tc, Nc, Zc, Cc, Vc fields form a three-level hardware stack T/N/Z/C/V signal. The current values are Tc/Nc/Zc/Cc/Vc, the saved values are Ts/Ns/Zs/Cs/Vs, and the backups of saved values are Tb/Nb/Zb/Cb/Vb.

Exception Cause Register (CR2)

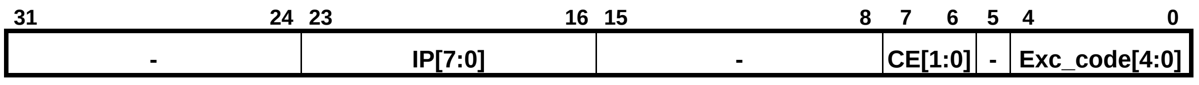


Fig 3-8 Exception Cause Register Format

Table 3-5 Exception Cause Register Field Description

Bits	Name	Description	R/W	Reset
31:24	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
23:16	**IP**	IP[1:0] indicates software interrupt pending (read/write)	R/W	0
		IP[7:2] represents 6-bit encoded priority interrupt request	R	0
15:8	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
7:6	**CE**	When control or coprocessor usable exception occur, this field indicates the number of responsible cause: CE[1:0] = 2'b00 : Control register accessible exception CE[1:0] = 2'b01: coprocessor 1 usable exception CE[1:0] = 2'b10: coprocessor 2 usable exception CE[1:0] = 2'b11: coprocessor 3 usable exception	R	0
5	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
4:0	**Exc_code**	When an exception occurs, the Exc_code field presents the exception cause	R	0

IP[7:2] is a 6-bit encoded priority interrupt request that processor will be servicing. A value 0 indicates that no interrupt request to be serviced. The value 1~63 represents the lowest (1) to the highest (63) priority for the interrupt to be serviced. Normally an external priority encoded interrupt controller is needed. IP[1:0] are software interrupt, and can be written to set or reset software interrupts.

EXCPVec Register (CR3)

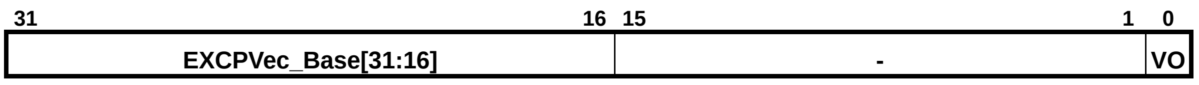


Fig 3-9 EXCPVec Register Format

Table 3-6 EXCPVec Register Field Description

Bits	Name	Description	R/W	Reset
31:16	**EXCPVec_Base**	Indicates all the exception vector address bit 31~16	R/W	0x9F00
15:1	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
0	**VO**	Indicate Vector address offset mode. 0: offset 0x4 1: offset 0x10	R/W	0

CCR Register (CR4)

31	9	8	7	6	5	4	3	2	1	0
	DPFB	IPFB	W-Back	RdBpDis	NOP	BTEN	LDM	LIM	MMU	WB

Fig 3-10 CCR Register Format

Table 3-7 CCR Register Field Description

Bits	Name	Description	R/W	Reset
31:10	**Reserve**	Reserved; read will return zero and write must be zero	R	0
9	**DPFB**	Enable Data Pre-Fetch Buffer	R	0
8	**IPFB**	Enable Instruction Pre-Fetch Buffer	R	0
7	**W-Back**	Enable Write-Back mode of data cache operation	R	0
6	**RdBpDis**	Disable Data read bypass mode of Write-Buffer function	R/W	0
5	**NOP**	(0) Normal; (1) NOP is as “Bubble” in pipe-line	R/W	0
4	**BTEN**	AMBA devices support burst early terminate function	R/W	0
3	**LDM**	LDM = 0 (1), indicates that local data memory interface is disable (enable).	R/W	0
2	**LIM**	LIM = 0 (1), indicates that local instruction memory interface is disable (enable).	R/W	0
1	**MMU**	MMU = 0 (1), indicates that memory management unit disable (enable)	R	0
0	**WB**	WB = 0 (1) indicates write buffer disable (enable). Default the write buffer is disabled.	R	0

EPC Register (CR5)

31	1	0
EPC[31:1]		M

Fig 3-11 EPC Register Format

Table 3-8 EPC Register Field Description

Bits	Name	Description	R/W	Reset
31:1	**EPC[31:1]**	This field records the most significant 31-bits of the program counter when an exception occurs. When a RTE instruction is executed, the value in EPC register would be restored to program counter register.	R/W	0
0	**M**	Indicates the instruction mode (32-bit or 16-bit) of the instruction at which a exception occurs. For exceptions that occur after D stage: 1 : indicates that exception instruction is a pce or 16-bit instruction 0 : indicates that exception instruction is a 32-bit instruction Otherwise, this bit would be undefined.	R	0

The address of an instruction at which an interrupt or exception occurred is saved to EPC. The bit 0 (M) of EPC indicates the exception caused instruction mode (16-bit (1) /32-bit (0) instruction mode). Moreover the bit 1 of EPC indicates the exception caused instruction location (low 16-bit instruction (1) /high 16-bit instruction or 32-bit instruction (0)). When executing RTE to restore EPC to current program counter, the bit 0 (M) will be ignored.

EPC[1:0] =	00	32-bit instruction
	01	High 16-bit instruction/PCE high 16-bit instruction (true)
	10	Illegal
	11	Low 16-bit instruction/ PCE low 16-bit instruction (false)

Exception Memory Address (EMA) Register (CR6)

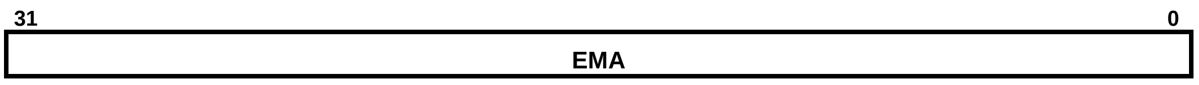


Fig 3-12 Exception Memory Address (EMA) Register Format

Table 3-9 Exception Memory Address (EMA) Register Field Description

Bits	Name	Description	R/W	Reset
31:0	**EMA**	EMA contains the virtual address (instruction or data) which generated an P-EL, AdEL, and AdES exception error.	R/W	0

Prev Register (CR18)

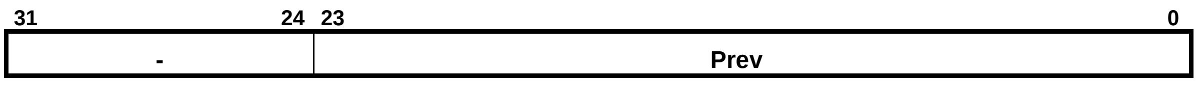


Fig 3-13 Prev Register Format

Table 3-10 Prev Register Field Description

Bits	Name	Description	R/W	Reset
31:24	**Reserve**	Reserved; read will return zero and write must be zero	R0/W0	0
23:0	**Prev**	Prev is a 24-bit read-only register containing the version and revision information of the processor.	R	NORM_ PREV

DREG Register (CR29)

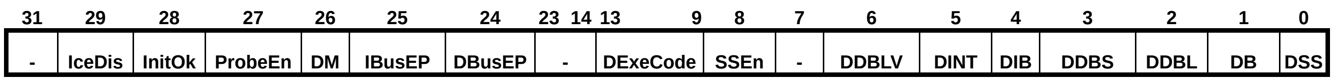


Fig 3-14 Debug register format

Table 3-11 Debug register field description

Bits	Name	Description	R/W	Reset
31:30	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0

Bits	Name	Description	R/W	Reset
29	**IceDis**	Disable SJTAG module	R/W	0
28	**InitOk**	Indicate the boot program is already finish system initialization. System is ready for download.	R/W	0
27	**ProbeEn**	Indicate Probe is enabled	R	0
26	**DM**	Debug Mode	R	0
25	**IBusEP**	Indicates if a Instruction Bus Error Exception pending	R	0
24	**DBusEP**	Indicates if a Data Bus Error Exception pending	R	0
23:14	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
13:9	**DExcCode**	Indicates the cause of latest exception in Debug Mode	R	Undef
8	**SSEn**	Single-step enable	R/W	0
7:6	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
6	**DDBLV**	Indicate a Debug Data Break Load exception match both address and data occurred	R	Undef
5	**DINT**	Indicate a Debug Interrupt exception occurred	R	Undef
4	**DIB**	Indicate a Debug Instruction Break exception occurred	R	Undef
3	**DDBS**	Indicate a Debug Data Break Store exception occurred	R	Undef
2	**DDBLA**	Indicate a Debug Data Break Load exception match address occurred	R	Undef
1	**DBP**	Indicate a Debug Breakpoint exception (caused by SDBBP) occurred	R	Undef
0	**DSS**	Indicate a Debug Single Step exception occurred	R	Undef

DEPC Register (CR30)

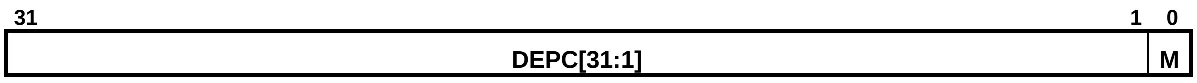


Fig 3-15 DEPC Register Format

Table 3-12 DEPC Register Field Description

Bits	Name	Description	R/W	Reset
31:1	**DEPC**	Debug Exception Program Counter. This register would save value the program counter when a debug exception occurs. When a DRTE instruction is executed, DEPC register value would be restored to program counter register.	R/W	Undef
0	**M**	Indicates the instruction mode (32-bit or 16-bit) of the instruction at which a debug exception occurs. For exceptions that occur after D stage: 1 : indicates that exception instruction is a pce or 16-bit instruction 0 : indicates that exception instruction is a 32-bit instruction Otherwise, this bit would be undefined.	R	0

DSAVE Register (CR31)

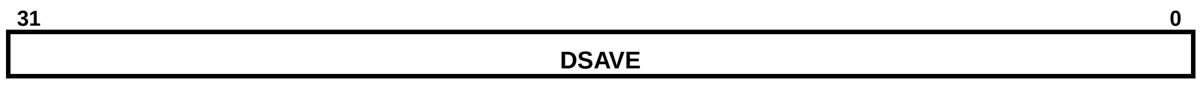


Fig 3-16 DSAVE Register Format

Table 3-13 DSAVE Register Field Description

Bits	Name	Description	R/W	Reset
31:0	**DSAVE**	Debug Exception Save contents	R/W	Undef

4 Exceptions

4.1 Exception Flow

The RTE instruction is used to return from exception handling. When the RTE instruction is executed, the EPC contents are restored to PC, and some of the PSR/Conditional register contents are right shifted (popped). The CPU returns from the exception handling routine by branching to the EPC address.

All exceptions are recognized in M (memory) stage, according the exception priority order.

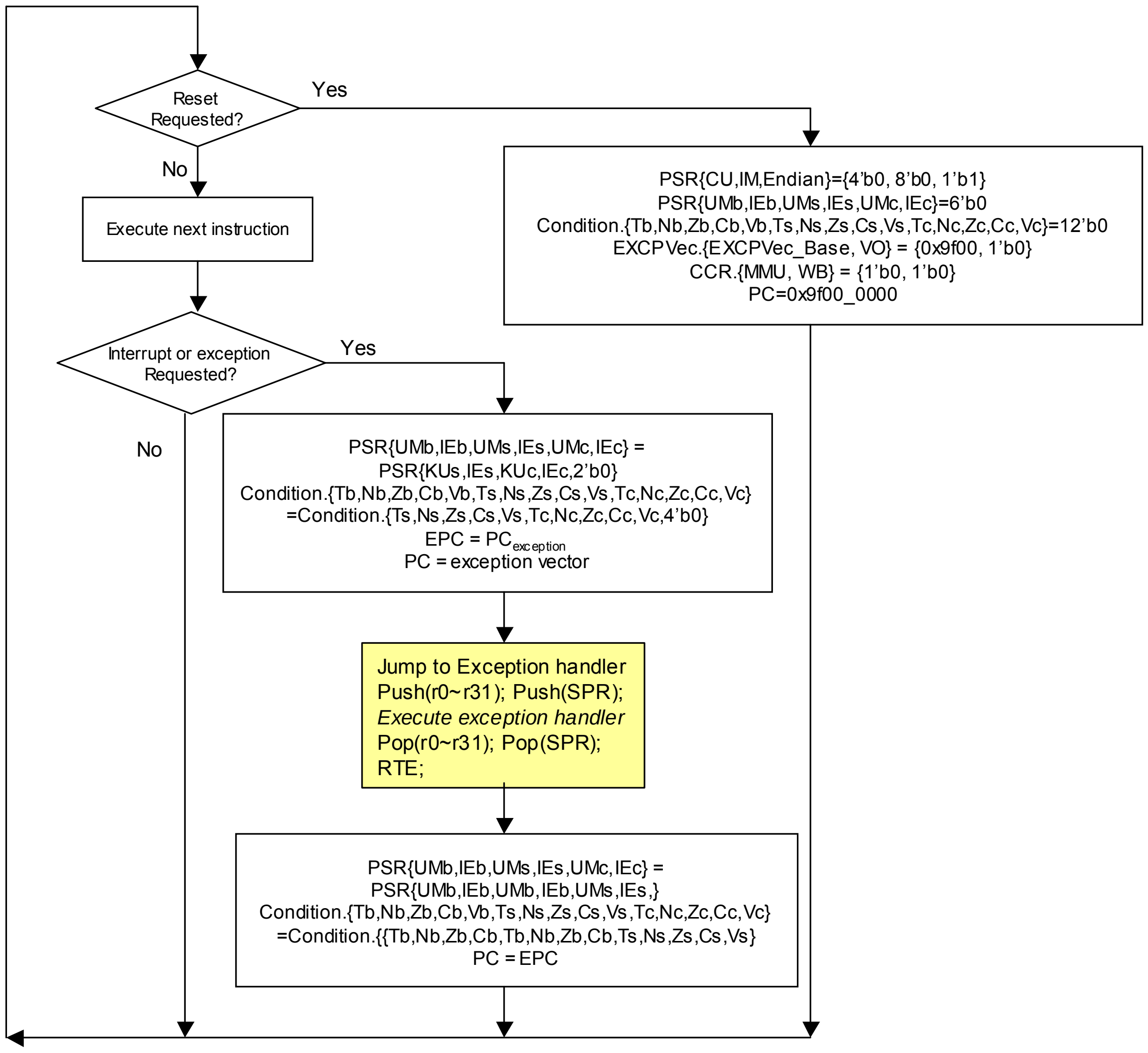

Reset Requested?
Yes
No
Execute next instruction
PSR{CU,IM,Endian}={4'b0, 8'b0, 1'b1}
PSR{UMb,IEb,UMs,IEs,UMc,IEc}=6'b0
Condition.{Tb,Nb,Zb,Cb,Vb,Ts,Ns,Zs,Cs,Vs,Tc,Nc,Zc,Cc,Vc}=12'b0
EXCPVec.{EXCPVec_Base, VO} = {0x9f00, 1'b0}
CCR.{MMU, WB} = {1'b0, 1'b0}
PC=0x9f00_0000
Interrupt or exception Requested?
Yes
No
PSR{UMb,IEb,UMs,IEs,UMc,IEc} =
PSR{KUs,IEs,KUc,IEc,2'b0}
Condition.{Tb,Nb,Zb,Cb,Vb,Ts,Ns,Zs,Cs,Vs,Tc,Nc,Zc,Cc,Vc}
=Condition.{Ts,Ns,Zs,Cs,Vs,Tc,Nc,Zc,Cc,Vc,4'b0}
EPC = PC exception
PC = exception vector
Jump to Exception handler
Push(r0~r31); Push(SPR);
Execute exception handler
Pop(r0~r31); Pop(SPR);
RTE;
PSR{UMb,IEb,UMs,IEs,UMc,IEc} =
PSR{UMb,IEb,UMb,IEb,UMs,IEs,}
Condition.{Tb,Nb,Zb,Cb,Vb,Ts,Ns,Zs,Cs,Vs,Tc,Nc,Zc,Cc,Vc}
=Condition.{{Tb,Nb,Zb,Cb,Tb,Nb,Zb,Cb,Ts,Ns,Zs,Cs,Vs}
PC = EPC

Fig 4-1 Exception Flow

4.2 Exception Priorities

4.2.1 Normal mode

When more than one exception can occur for a single instruction, only one exception is reported, with priority given in the order shown in Table 4-1.

Table 4-1 Exception Priority

Priority	Exception	Description
1 (Highest)	Reset	Reset
2	DSS	Debug single step exception
3	DINT	Debug processor bus break or JTAG break exception
4	DDBLV	Debug data address break load exception
5	NMI	Non-maskable interrupt NMI
6	Interrupt [63]	External hardware interrupt exception
.	.	.
.	.	.
68	Interrupt [1]	External hardware interrupt exception
69	DIB	Debug instruction address break exception
70	AdEL-instruction	Instruction fetch address error exception
71	BusEL-instruction	Instruction fetch bus error exception
72	P-EL	Instruction P-bit check error exception
73	DBP	Debug breakpoint exception (SDBBP)
73	SYSCALL	System call trap exception
73	CCU	Control or coprocessor unusable exception
73	RI	Reserved instruction exception
73	Trap	Conditional trap exception
74	DDBLA	Debug data address break load exception
75	DDBS	Debug data address break store exception
76	AdEL-data	Data load address error exception
77	AdES	Data store address error exception
78	CeE	Custom engine execute exception (Divided by zero)
78	CpE	Coprocessor z execute exception
79	BusEL-data	Data access bus error exception
80(Lowest)	SWI [1]	Internal software interrupt
80(Lowest)	SWI [2]	Internal software interrupt

4.2.2 Debug Mode

At debug mode, the debug exceptions and interrupt related instructions would be blocked. Table 4-2 lists the behavior of the exceptions during debug mode. Other exception will cause a re-entry to debug mode with a debug mode exception code (DExc) which shown in Table 4-2.

Table 4-2 Debug mode exception priority

Priority	Exception	Core Behavior
Highest	Reset	Reset
	Debug single step exception (DSS)	Blocked
	Debug Interrupt (DINT)	Blocked
	Debug data break load exception match both address and data (DDBLV)	Blocked
	Non-maskable interrupt NMI	Blocked
	External hardware interrupt exception	Blocked
	Debug instruction address break exception (DIB)	Blocked
	Instruction fetch address error exception	Re-enter Debug Mode
	Instruction fetch bus error exception	Re-enter Debug Mode
	Instruction parity check error exception	Re-enter Debug Mode
	Debug breakpoint exception (SDBBP)	Blocked
	Conditional trap exception	Re-enter Debug Mode
	System call trap exception	Re-enter Debug Mode
	Control or coprocessor unusable exception	Re-enter Debug Mode
	Reserved instruction exception	Re-enter Debug Mode
	Debug data break load exception match address only (DDBLA)	Blocked
	Debug data break store exception (DDBS)	Blocked
	Data load address error exception	Re-enter Debug Mode
	Data store address error exception	Re-enter Debug Mode
	Custom engine execute exception	Re-enter Debug Mode
	Coprocessor z execute exception	Re-enter Debug Mode
	Data access bus error exception (precise)	Re-enter Debug Mode
	Data access bus error exception (imprecise)	Blocked
Lowest	Internal software interrupt	Blocked

4.3 Exception Cause Table

Table 4-3 Normal exception code encoding

Exception	Exception Code
Reset	0
NMI	1
AdEL-instruction	2
-	3
-	4
BusEL-instruction	5
P-EL	6
SYSCALL	7
CCU	8
RI	9
Trap	10
AdEL-data	11
AdES	12
-	13
-	14
-	15
CeE	16
CpE	17
BusEL-data	18
SWI	19
Interrupt	20

4.4 Exception Vector

Badr = {EXCPVec_Base[31:16], 16’b0}

Table 4-4 Exception Vector Table

Exception Type	**Vector Address**		**Exceptions**
	VO=0	**VO=1**	
Reset	**0x9F00_0000**	**0x9F00_0000**	Reset
Debug	**Badr + 0x1FC**	**Badr + 0x1FC**	Debug
General	**Badr + 0x200**	**Badr + 0x200**	Non-maskable interrupt (NMI) AdEL-instruction Instruction Fetch Bus Error (IBusEL) P-bit Error (P-EL) System Call (SYSCALL) Control or Coprocessor Unusable (CCU) Reserve Instruction (RI) Trap Load address Error (AdEL-data) Store address Error (AdES) Custom engine exception (CeE) Coprocessor exception (CpE) Data access bus error(DBusEL) Software Interrupt (SWI) [1] Software Interrupt (SWI) [2]
Int 1	**Badr + 0x204**	**Badr + 0x210**	Interrupt [1]
.	**.**	**.**	.
.	**.**	**.**	.
Int 63	**Badr + 0x2FC**	**Badr + 0x5F0**	Interrupt [63]

4.4.1 Debug Exception Code

Table 4-5 Debug exception code encoding

Exception	Exception Code
Reset	0
NMI	1
AdEL-instruction	2
-	3
-	4
BusEL-instruction	5
P-EL	6
SYSCALL	7
CCU	8
RI	9
Trap	10
AdEL-data	11
AdES	12
-	13
-	14
-	15
CeE	16
CpE	17
BusEL-data	18
SWI	19
Interrupt	20

4.5 Exception Descriptions

4.5.1 Reset Exception

Cause. When CPU reset signal is asserted, the CPU reset exception occurs. This exception is unmaskable.

Handle. The CPU provides a special reset vector (0x9f00_0000) for this exception. This reset vector resides in un-cached region, and the CPU is under fixed mapping mode.

The contents of all registers in the CPU are undefined when this exception occurs except the following:

- The bit fields IEc, UMc are zeros in Processor Status register (PSR)
- The bit fields MMU, WB are zeros in CCR register
- The Exc_code in ECR register is 0
- The EXCVec_Base in EXCVEC register is 0x9F00_0000

4.5.2 Non-Maskable Interrupt (NMI) Exception

Cause. The non-maskable interrupt exception occurs in response to the falling edge of the NMI pin. This exception is regardless of the setting of the IEc bit in PSR. It is a non-recoverable exception.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR register is 1
- The EPC register points at the instruction that cause the NMI

4.5.3 Address Error Exception

Cause. (1) load, or store a word that is not aligned on a word boundary (2) load, store a halfword that is not aligned on a halfword boundary (3) The instruction address is not align to halfword boundary (the LSB of instruction address is equal to one) (4) reference a Kernel address space from User mode (5) reference a Debug address space from Kernel and User mode

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 2 (AdEL-instruction), 11(AdEL-data), 12(AdES-data)
- The EPC register points at the instruction that causes the exception
- The Exception Memory Address (EMA) register retains the virtual address that is not properly aligned or which referenced protected address space
- The EPC register points at the instruction that cause the exception
- The EMA and PEVN registers hold the virtual address that failed address translation

4.5.4 Bus Error Exception

Cause. The bus error exception occurs when signaled by board-level circuitry for events such as bus time-out, backplane bus P-bit errors, and invalid physical memory address or access types.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 5 (instruction), 18(data)
- The EPC register points at the instruction that cause the exception (If the processor has write-buffer, the EPC caused by data bus error will not correctly point to instruction. Then it will be a non-recoverable exception.)

4.5.5 Trap Exception

Cause. The trap exception occurs on an attempt to execute the Trap instruction.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 10
- The EPC register points at the Trap instruction

4.5.6 SYSCALL Exception

Cause. The SYSCALL exception occurs on an attempt to execute the SYSCALL instruction.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 7
- The EPC register points at the SYSCALL instruction

4.5.7 P-EL Exception

Cause. (1) The P-EL exception occurs on an attempt to execute an instruction that is P-bit check fail. (2) The instruction address (Ex. Branch target address) points to the lower halfword of an instruction word, but this instruction word is a 32-bit instruction. Instruction address word alignment error (instruction address bit 1=1).

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, UMc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 6
- The EMA register points at the instruction that cause the exception
- The EPC register points at the instruction that cause the exception

For example:

```
0          la     $5, 0x102
4          jr     $5
:
:
100        add    $2, $2,0x1 ← Instruction P-bit check error. EPC = 0x102
:
```

4.5.8 Reserved Instruction (RI) Exception

Cause. The RI exception occurs on an attempt to execute an instruction whose opcode is undefined.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, KUc in PSR register are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR register is 9
- The EPC register points at the reserved instruction

4.5.9 Control or Coprocessor Unusable (CCU) Exception

Cause. This exception occur when an attempt is made to execute one of the following instructions:

- Control register instruction, when the unit has not been marked usable and the processor is executing in User mode, or
- Coprocessor instructions that their corresponding coprocessor unit has not been marked usable

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, KUc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 8
- The CE in ECR indicates the unusable coprocessor number
- The EPC register points at the unusable instruction

4.5.10 Custom engine Execution exception (Divided by zero)

Cause. The CeE exception occurs when a custom engine is attempted to execute an extended instruction and the execution result reports an exception. For example, divide by zero, multiply and accumulate overflow. We define the divided by zero exception as the default custom engine execution exception.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, KUc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 16
- The EPC register points at the custom engine extension instruction

4.5.11 Coprocessor z Execute (CpE) Exception

Cause. The CpE exception occurs when a coprocessor z is attempted to execute a coprocessor instruction and the execution result reports an exception. For example, floating point divide by zero, floating point multiply and accumulate overflow.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, KUc in PSR register are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR register is 17
- The CE in ECR register indicates the exception coprocessor number
- The EPC register points at the coprocessor instruction

4.5.12 Interrupt Exception

Cause. The interrupt exception occurs when one of the sixty-five interrupt conditions is asserted.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields IEc, KUc in PSR are pushed
- The bit fields Tc, Nc, Zc, Cc, Vc in Condition register are pushed
- The Exc_code in ECR is 19 (software interrupt), 20 (hardware interrupt)
- The EPC register points at the instruction that interrupt has occurs. When interrupt service routine has finished, we must go back to the EPC instruction.

Note: Before priority encoded the external sixty-three interrupts, the interrupt controller has to synchronize and de-bounce the sixty-three interrupt inputs using processor clock. Otherwise external interrupt glitch may cause processor interrupt recognition error.

4.5.13 Debug Interrupt Exception

Cause. A debug interrupt execution happens in non-debug mode.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields DINT, and DM in control register 29 (DREG) are set
- The control register 30 (DEPC) holds the address where processing resume after exception routine has finished. Execution of the DRTE instruction causes a jump to the address in the DEPC.

4.5.14 Debug Single Step Exception

Cause. After single step execution of an instruction in non-debug mode and the SSEn bit in the Debug register is set.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields DSS, and DM in control register 29 (DREG) are set
- The control register 30 (DEPC) holds the address where processing resume after exception routine has finished. Execution of the DRTE instruction causes a jump to the address in the DEPC.

4.5.15 Debug Breakpoint Exception

Cause. After executing of the SDBBP (software debug breakpoint) instruction.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- If the exception is taken in non-debug mode, the bit field DBp in control register 29 (DREG) is set.
- If the exception is taken in debug mode, the bit fields of DSS, DBP, DDBL, DDBS, DIB, DDB, and DINT are cleared and DexcCode is set to SDBBP in control register 29 (DREG).
- The bit field DM in control register 29 (DREG) is set
- Control register 30 (DEPC) holds the address where processing resume after exception routine has finished. Execution of the DRTE instruction causes a jump to the address in the DEPC.

4.5.16 Debug Data Address Break Exception

Cause. Data address match during a load/store memory instruction in non-debug mode.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit field DDBLA, DDBLV or DDBS in control register 29 (DREG) is set
- The bit field DM in control register 29 (DREG) is set
- Control register 30 (DEPC) holds the address where processing resume after exception routine has finished. Execution of the DRTE instruction causes a jump to the address in the DEPC.

4.5.17 Debug Instruction Address Break Exception

Cause. When an instruction address match occurred in non-debug mode.

The contents of all registers in the CPU are unchanged when this exception occurs except the following:

- The bit fields DIB, and DM in control register 29 (DREG) are set
- Control register 30 (DEPC) holds the address where processing resume after exception routine has finished. Execution of the DRTE instruction causes a jump to the address in the DEPC.

5 Cache

S+core processor supports separate instruction and data caches. The use of separate caches allows instruction and data references to proceed simultaneously. Both caches are virtually indexed and physically tagged, allowing cache access to occur in parallel with virtual-to-physical address translation. **Table 5-1** lists the cache specification of S+core:

Table 5-1 S+core Cache Specification

	I-Cache	D-Cache
Cache Size	4 Kbytes	4 Kbytes
Set Association	2	2
Cache-line Size	4 Words	4 Words
Write Strategy	NA	Write Through
Allocate Strategy	Read Allocate	Read Allocate
Replacement Strategy	LRU	LRU
Pre-fetch A Cache-line	Virtual Address Mode	Virtual Address Mode
Pre-fetch and lock a Cache-line	Virtual Address Mode	Virtual Address Mode
Invalid A Cache-line	Virtual Address Mode	Virtual Address Mode
Invalid Entire Cache	Yes	Yes
Drain Write Buffer	NA	Yes
Write Buffer	NA	4 Words and Addresses

Cache control register (CCR): Control Register 4

Bit 0: Write Buffer Enable/Disable (Read)

The tag memory array records the data in cache-line unit while the data memory array records the data in word unit. I-Cache and D-Cache are two-way set-association structure and the cache size is 4K bytes. The memory array structure shows in Fig 5-1.

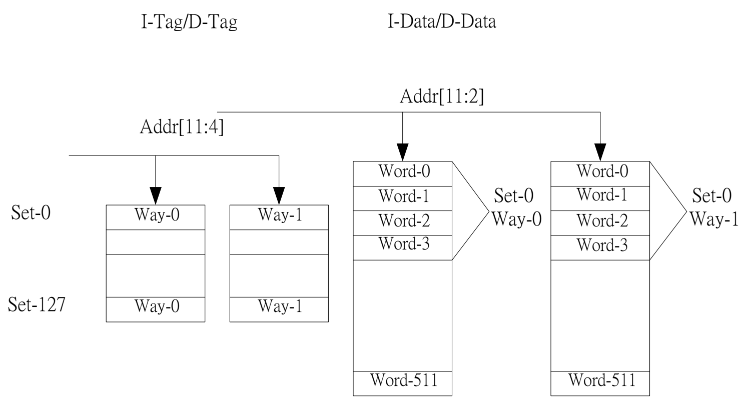


Fig 5-1 Tag and Data device of S+core cache controller

5.1 Instruction Cache

I-Cache controller is an independent device unit for processor core. There are two request buses between processor core and I-Cache controller. One is instruction bus (**I-Bus**) and the other one is I-Cache instruction bus. I-Cache controller needs to complete the two requests from the two buses. I-Cache controller will issue device busy to hold pipeline for solving some event when it can't complete the request in next cycle. We will describe the all pipeline events about I-Cache controller in the section.

I-Cache Features:

1. I-Cache is two-way set-association cache architecture.
2. I-Cache size is 4K~128K bytes.
3. Cache-line size is four/eight words.
4. Allocation strategy is read allocation.
5. Support invalid one cache-line command (Virtual Address Mode).
6. Support invalid entire cache-line command.
7. Support pre-fetch one cache-line command (Virtual Address Mode).
8. Support pre-fetch and lock a cache-line command (Virtual Address Mode).
9. There are "instruction fetch request bus (**I-Bus**)" and "I-Cache instruction request bus" between core and I-Cache controller.
10. Precise bus error exception for Instruction fetch.
11. Processor core can kill previous and current I-Cache device request.

5.2 Data Cache

D-Cache controller is an independent device unit for processor core. There are two request buses from processor core. One bus is for Load/Store instruction request (**D-Bus**) and the other one bus is for D-Cache instruction. D-Cache controller needs to complete the two requests from the two buses. D-Cache controller will hold processor pipeline when it can't complete request in next cycle.

1. D-Cache is two-way set association cache architecture.
2. D-Cache size is 4K~128K bytes.
3. Cache-line size is four/eight words.
4. Allocation strategy is read allocation.
5. Support cache-line invalid command.
6. Support cache-line pre-fetch command.
7. Support cache-line pre-fetch and lock command.
8. There are Load/Store Instruction request bus (D-Bus) and D-Cache Instruction operation request bus.
9. Precise bus error exception for Load Instruction and "Store Instruction (Write-buffer is disabled)".

10. Imprecise bus error exception for “Store Instruction (Write-buffer is enabled)”.
11. Processor core can kill previous and current D-Cache device request.

5.3 Virtual Alias

Alias occurs when multiple virtual addresses map to a same physical address. Since the caches are virtually indexed and physically tagged, a potential issue referred to as virtual alias might exist. Virtual alias occurs if the virtual bits used to index a cache array are not consistent with the overlapping physical bits, after the virtual address has been translated to a physical address. S^+core has two mode of MMU architecture. One is fixed memory management and the other is full function mode MMU. The possibility of virtual alias only occurs in full function mode MMU. In full function mode MMU, virtual alias may occur if the cache size per set is greater than the page size. The page size of S^+core in MMU mode is 4Kbytes.

For example, consider a 16 KB cache organized as 2-way set associative. The size per set is then 8 KB, so virtual address bits [12:0] are used to index the array. If the address is in a translated region with a page size of 4 KB, then address bits [11:0] are untranslated but address bits [31:12] will be mapped and for these bits the virtual and physical addresses may be different. In this example, bit [12] could pose a potential problem due to virtual alias. Imagine two virtual addresses, VA0 and VA1, whose only difference is the value of bit [12], which map to the same physical address. These two virtual addresses would be indexed to two different lines by the cache, even though they were intended to represent the same physical address. Then if a program does a load using VA0 and a store using VA1, or vice-versa, the cache may not return the expected data.

A related issue can occur in virtually indexed, physically tagged caches if the number of physical bits stored in the tag array do not fully overlap the physically translated bits for the smallest page size. For S^+core, there are at least 20 address bits stored in the cache tag, representing bits [31:12] of the physical address. Since the minimum page size is 4 KB for the S^+core, with bits [31:12] physically translated by the TLB, the cache tag size does overlap the translated bits and this issue will not occur.

5.4 Memory Coherency

Memory coherency issues must be considered in the system design. Since a cache holds a copy of memory data, it is possible for another memory master to modify the memory location, thus making other copies of that location stale if those copies are still in use. System design or software must handle the memory coherence issues. There are four typical memory coherency issues in the following:

Writing instructions: When the CPU itself is storing instructions into memory for subsequent execution, you must first ensure that the instructions are written back to memory and then make sure that the corresponding I-Cache locations are invalidated: S^+core CPU has no connection between the D-cache and the I-cache.

Write Back Cache: If a device is taking data out of memory (uncacheable), it's vital that it gets the right data. If the data cache is write back and a program has recently written some data, some of the correct data may still be held in the D-Cache but not yet be written back to main memory. The CPU can't see this problem. So before the DMA device starts reading data from memory, any data for that range of locations that is currently held in the D-Cache must be written back to memory if necessary.

DMA into memory: If a device is loading data into memory, it's important to invalidate any cache entries purporting to hold copies of the memory locations concerned; otherwise, the CPU reading these locations will obtain stale cached data. The cache entries should be invalidated before the CPU uses any data from the DMA input stream.

Write Buffer: Some data is still in write buffer when write buffer is enabled. Therefore, the memory content/ IO register isn't correct in the moment.

5.5 Cache Instruction

CACHE Cache-OP, [rA, offset] CACHE 5, [r2, 0x0]

	29–25	24–20	19–15	14–0
CACHE	OP	Cache_op	rA	Imm15
	1 1 0 0 0			

Table 5-2 Cache instruction sub-OP code

Cache-OP[4:0]	I-Cache/D-Cache	Function	Data
0x00	I-Cache	Pre-fetch a Cache-line	VA
0x01	I-Cache	Pre-fetch and lock a Cache-line	VA
0x02	I-Cache	Invalid and unlock a Cache-line	VA
0x03	I-Cache	Fill LIM (local instruction memory) device	VA (PFN & Size)
0x04	I-Cache	Re-Fill LIM (local instruction memory with the PFN and Size of previous value) device	VA
0x08	D-Cache	Pre-fetch a Cache-line	VA
0x09	D-Cache	Pre-fetch and lock a Cache-line	VA
0x0A	D-Cache	Invalid and unlock a Cache-line	VA
0x0B	D-Cache	Fill LDM (local data memory) device	VA (PFN & Size)
0x0C	D-Cache	Write-Back LDM (local data memory) device to main memory	NA
0x0D	D-Cache	Force write-back a Cache-line and set valid when the cache-line is valid and dirty	VA
0x0E	D-Cache	Force write-back a Cache-line and set invalid when the cache-line is valid and dirty	VA
0x10	I-Cache	Invalid entire cache	NA
0x11	I-Cache	Toggle Instruction Pre-fetch Buffer Function (Enable/Disable)	NA

Cache-OP[4:0]	I-Cache/D-Cache	Function	Data
0x18	D-Cache	Invalid entire Data cache	NA
0x1A	D-Cache	Drain Write Buffer	NA
0x1B	D-Cache	Toggle Write Buffer Function	NA
0x1C	D-Cache	Toggle Data Pre-fetch Buffer Function (Enable/Disable)	NA
0x1D	D-Cache	Toggle Write-back D-Cache Function (Enable/Disable)	NA
0x1E	D-Cache	Force write-back entire D-Cache and set valid of the cache-lines are valid and dirty. (Write-out)	NA
0x1F	D-Cache	Force write-back entire D-Cache and set invalid of the cache-lines are valid and dirty. (Flush)	NA

Virtual address Mode Operation:

S Stage:

Processor core issue the cache-op to cache controller.

E Stage:

Cache controller responses the cache-op of S-Stage issued.

Generate Undefined Exception (RI) for wrong cache op-code.

VA = GPR[base] + sign_extend(Offset)

A Stage:

MMU(VA → PA)

Issue an I-Cache/D-cache instruction to Cache controller in A Stage.

M Stage: (Perhaps Stall Pipeline several cycle)

Cache controller:

Get Cache_Ins(OP, VA[17:4])

Get PA[31:12] from D-MMU

Issue Freeze Request to processor core

Cycle 1: read the I-Tag/D-Tag:

Cycle 2: Compare {PA[31:12], VA[11] } with TAG(PA) of all way

Cache-line invalid operation:

{

Case 1: Find one way Tag (PA) equal to the cache-inst.'s PFN

Deassert the Tag valid bit and deassert the "Lock" bit when the way is way-0.

```
          Deassert the Tag valid bit when the way is way-1
Case 2:All way Tag (PA) is unequal to the cache-inst.'s PFN
          Ignore the cache instruction.
}



Cache-line pre-fetch operation:
{
Phase 1: Replace Strategy
Case 1: Find one way Tag (PA) equal to the cache-inst.'s PFN – Hit
          Replace the cache-line and assert valid and modify MRU bit.
Case 2:   All way Tag (PA) is unequal to the cache-inst.'s PFN   - Miss
          Cache-line replaces strategy is MRU.
         Write-back the cache line when it is dirty.

Phase 2: Replace the cache-line
}

Cache-line pre-fetch and lock operation:
{
  Phase 1: Replace Strategy
Case 1: Find Way-0 Tag (PA) equal to the cache-inst.'s PFN -- Hit
               Replace the cache-line and assert MRU and Lock bit.
Case 2: Find Way-1 Tag (PA) equal to the cache-inst.'s PFN -- Hit
               Deassert the valid bit of Way-1 Tag.
               Replace the cache-line of Way-0 and assert valid, MRU and Lock bit.
Case 3:All way Tag (PA) is unequal to the cache-inst.'s PFN -- Miss
               Replace the cache-line of Way-0 and assert valid, MRU and Lock bit.
             Write-back the cache line when it is dirty.

Phase 2: Refill the cache-line
}

Cache-line write-back and set valid (Write-out):
{
  Phase 1:
Case 1: When Way-0/Way-1 Tag (PA) is equal to cache inst. PFN -- Hit
```

```
                  Write back the cache line to main memory when the cache line is dirty.
                And then set the line to clean. The MRU bit is set to the line.
Case 2: When Way-0 and Way-1 Tag (PA) are unequal to cache inst. PFN -- Miss
                 Ignore the cache instruction.
}

Cache-line write-back and set invalid (Flush):
{
  Phase 1:
Case 1: When Way-0/Way-1 Tag (PA) is equal to cache inst. PFN -- Hit
                  Write back the cache line to main memory when the cache line is dirty.
                And then set the cache line to invalid and clean. The MRU bit is set to another way.
Case 2: When Way-0 & Way-1 Tag (PA) are unequal to cache inst. PFN -- Miss
                 Ignore the cache instruction.
}

Cache-line invalid operation:
{
Case 1: Deassert the MRU, Dirty, Valid and Lock bit for Way-0 operation.
Case 2: Deassert the MRU (Tag-0), Dirty and Valid (Tag-1) bit for Way-1 operation.
}

Invalid entire I-Cache/D-Cache operation:
{
     Deassert the MRU, Dirty, Valid and Lock bit of all Way-0 Tag.
     Deassert the Dirty and Valid bit of all Way-1 Tag.
}

Drain Write Buffer operation:
{
     Issue a drain write buffer to BIU.
     Wait BIU completes drain write buffer.
}

Force Write-back Entire D-cache and set valid (Write-out):
{
  While (Scan search every cache-line) {
          Write-Back the entry when it is dirty and valid
         Deassert dirty bit
  }
}
Force Write-back Entire D-cache and set invalid (Flush):
{
   While (Scan search every cache-line) {
          Write-Back the entry when it is dirty and valid
        Deassert dirty and valid bit
        Deassert lock bit when hit on way-0
        Set MRU bit to another way
  }
}
```

Restrictions:

Cache instruction will generate UNDEFINED (RI) exception when the cache-op is undefined.

Exception:

Undefined Instruction Exception (RI).

Bus Error Exception (Pre-fetch/fill Cache instruction: BusErr-Data) is a precise exception for the "Cache" instruction.

Address Error Exception: (1) load, or store a word that is not aligned on a word boundary (2) load, store a halfword that is not aligned on a halfword boundary (3) The instruction address is not align to halfword boundary (the LSB of instruction address is equal to one) (4) reference a Kernel address space from User mode (pre-fetch and write-back cache inst.) (5) reference a Debug address space from Kernel and User mode(pre-fetch and write-back cache inst.).

Note:

1. Cache instruction doesn't generate address error by address align issue (base address + offset field). Cache controller ignores some bits field from LSB. Cache instruction doesn't generate address error exception of item from 1 to 3.
2. Cache controller will ignore the command when pre-fetch and write-back cache instructions operate to I-Cache or D-Cache but the address (segment/page) is un-cacheable range. LIM and LDM fill-inst are the exceptions.
3. The size of LIM/LDM cache instructions is using line number of the VA. The minimal operation size is one line (line-0) for filling, re-filling or write-back.

1. Base address (PFN address) of LIM/LDM cache instructions was aligned to the LIM/LDM memory size. (For example: The base address is PFN [19:1] when LIM is 8K bytes. The maximal size is 512 lines when cache line size is 4 words and the maximal size is 256 when cache line size is 8 words.)

5.6 Local Instruction Memory and Local Data Memory

5.6.1 Local Instruction Memory

Local instruction memory (LIM) system includes the SRAM (synchronous RAM) in I-Cache controller. The synchronous memory interface gives designers the faster interface to memory blocks. You can enable the LIM by setting bit 2 of the CCR control register. User must be taken to ensure that the LIM device is appropriately initialized before it is enabled and used. S+core supports the “Cache” instruction initialize/fill the SRAM content.

5.6.2 Enhance Data Local Memory Solution

Local data memory (LDM) system includes the SRAM (synchronous RAM). The synchronous memory interface gives designers the faster interface to the memory blocks. You can enable the LDM by setting bit 3 of the CCR control register. User must be taken to ensure that the LDM device is appropriately initialized before it is enabled and used. S+core supports the “Cache” instruction initialize/fill the SRAM content.

The LDM address range is configurable by the cache instruction (fill LDM). The physical address and size of LDM device are recorded by I-Cache controller when the LDM fill instruction was executed.

6 Debug and SJTAG

6.1 Overview

The debug solution in processor includes following features:

- **Off-chip debug memory access**

  SJTAG allows processor in the debug mode access the instructions or data that are not physically on the chip. Memory access to the special segment will be transferred to JTAG protocol and then sent to the debug probe that controlled by debug host PC. Debug probe will handle the memory access by redirect the access to its local memory. Then, the accessed data will feedback to processor via JTAG again.

  This mechanism allows system debugging without requiring of the debug ROM on chip and provide a communication path between processor and debug host PC.

- **Hardware breakpoint**

  Two types of hardware breakpoint are included which can be configured to cause debug exception on

  1) Instruction fetch of a specified address (Breakpoint)
  2) Data fetch of a specified address and the access data value (Watchpoint).

- **Software breakpoint instruction**

  Two additional instructions are added to achieve system debug: Software Debug Breakpoint (SDBBP) and Debug Exception Return (DRET).

- **Single step execution**

  A dedicate single step exception in processor is provided for single step debugging.

- **Debug interrupt**

  A debug interrupt is provided to force processor enter debug mode at any time.

- **DMA Access**

  A DMA channel that controlled by SJTAG directly access system bus through BIU. This feature is very useful when user need to download code to system memory.

6.1.1 Processor Debug Behavior

The processor debug follows one rule of thumb: when system run to a special condition user expect the bug happens, the debugger take some mechanism to dump system internal states and information that we can not obtain from chip boundary. The “special condition” for debug can be instruction fetch from a user indicate address, load of a special data value or just 2000 cycles after system boot. Generally, we implement the hardware and software breakpoint as the trigger point of the special condition.

When the special condition happens, debugger program in the host PC must have some mechanism to access the chip internal information, such as PC and GPR value. An easy method is to get the control of processor execution program, and then you can let CPU to dump any information you want.

According the idea above, a mechanism is proposed as shown in Fig 6-1 and Fig 6-2.

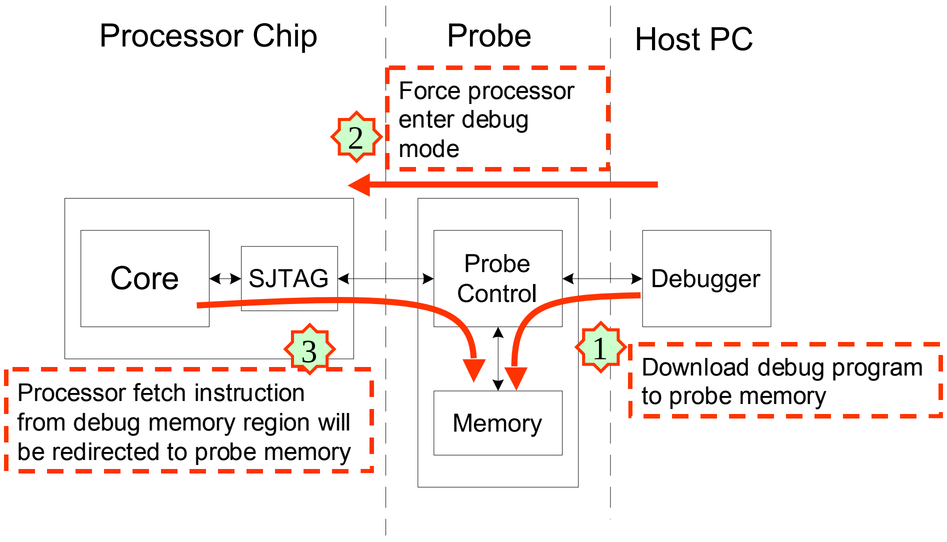

Processor Chip
Probe
Host PC
Force processor enter debug mode
2
Core
SJTAG
Probe Control
Debugger
3
1
Memory
Processor fetch instruction from debug memory region will be redirected to probe memory
Download debug program to probe memory

Fig 6-1 How core get control of processor

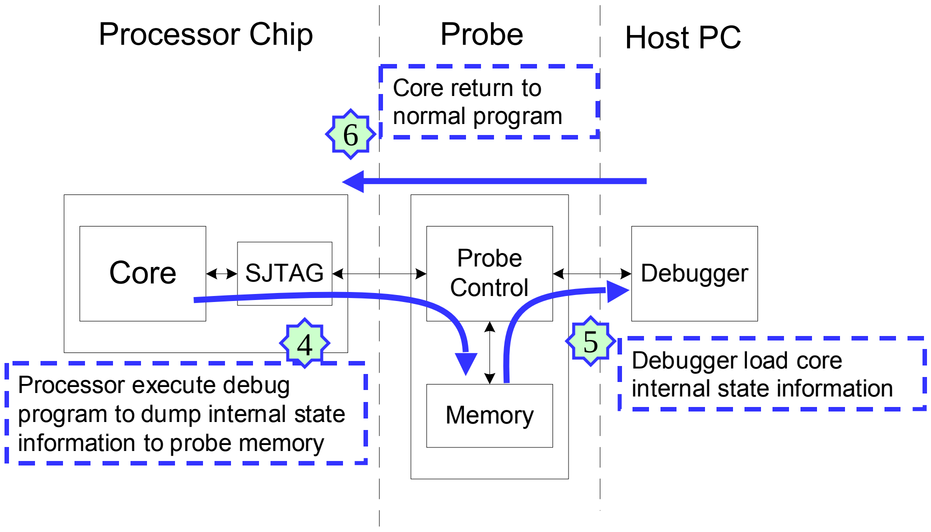

Processor Chip
Probe
Host PC
Core return to normal program
6
Core
SJTAG
Probe Control
Debugger
4
5
Memory
Processor execute debug program to dump internal state information to probe memory
Debugger load core internal state information

Fig 6-2 Debugger load processor internal state information in debug mode

6.1.2 Debug Architecture

The whole debug architecture includes two parts: the processor core extension and SJTAG debug module, as shown in Fig 6-3.

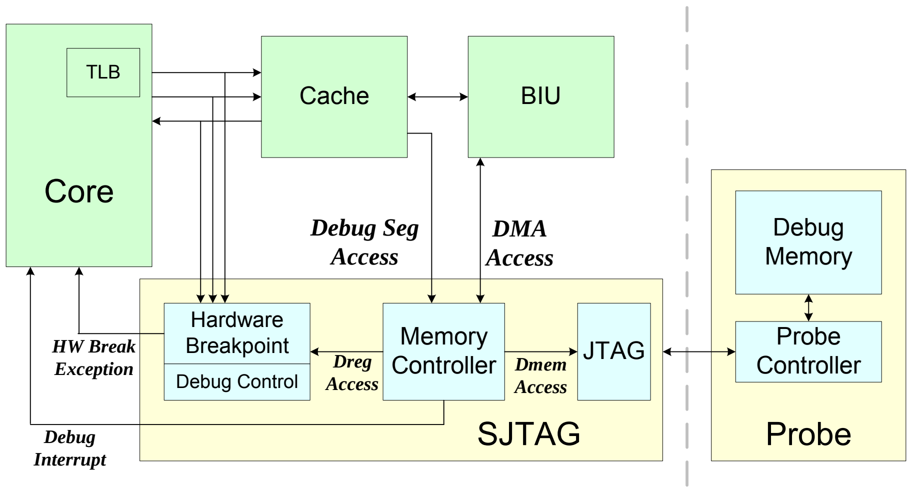


Fig 6-3 Debug system architecture

6.1.3 Processor Core Debug Extension

The processor core debug extension contains the debug memory segment, debug exception, and *the control registers* for debug mode. Processor enter debug mode when the debug exception was taken. After core enter debug mode, the program execute in the debug segment memory region and *the control registers* record the core debug status, and debug exception return address.

6.1.4 SJTAG Module

The SJTAG module includes some blocks as shown in Fig 6-3.

- The hardware point unit will assert the breakpoint exception request to core.
- A memory controller handles the memory request from core if the address is located in debug segment. If the access is in debug register segment, it will directly access the debug registers such as breakpoint unit registers or debug **control** register. If the access is in debug memory segment, it will pass the access to debug probe through JTAG interface.
- A JTAG controller handles the communication between probe and SJTAG module.

6.1.5 Processor Core Debug Extension

The processor core extension for debug includes three parts:

- A special debug operation mode with privilege to access debug segment address space, different behavior to handle exceptions and three debug registers in the control register set.

- Some debug exceptions is provided to change processor into debug mode.
- Two instructions are added to the instruction set for debug.

6.1.6 Debug Mode Address Space

Debug segment locates at memory region from 0xFF00_0000 to 0xFFFF_FFFF. Debug segment can be divided into debug memory segment and debug register segment as listed in Table 6-1.

Table 6-1 Debug segment address range

Name	Segment	Address Range
Debug Memory	*dmseg*	*0xFF00_0000 – 0xFFEF_FFFF*
Debug Register	*drseg*	*0xFFF0_0000 – 0xFFFF_FFFF*

Access to the debug segment in non-debug mode (user or kernel mode) will cause an address error exception.

If processor requests a memory access to the debug memory segment, the SJTAG will redirect the request to the debug probe through JTAG interface. These mean the processor can directly run program and access data at the off-chip probe memory. Moreover, debug host PC also can access the probe memory content. This creates a path for the debug host to take the control to processor core as shown in Fig 6-1.

The debug register segment contains the breakpoint module control registers, such as breakpoint match address, breakpoint control and breakpoint status. Processor can configure the hardware breakpoint by programming the register in this segment.

6.1.7 Debug Exceptions

This section describes the debug exception and the issues related to these exceptions. The processor will enter the debug mode when the debug exception is launched.

- **Debug Exception Priorities**

  The priority of the debug exception in normal mode is listed in Table 6-2. This priority meets our exception rule: higher priority exception happens in earlier stage.

Table 6-2 Debug exception priority

Type	Priority	Exception
Non-debug	Highest	Reset
Debug		Debug single step exception (DSS)
Debug		Debug Interrupt (DINT)
Debug		Debug data break load exception match both address and data (DDBLV)
Non-debug		Non-maskable interrupt NMI
Non-debug		External hardware interrupt exception
Debug		Debug instruction address break exception (DIB)
Non-debug		Instruction fetch address error exception
Non-debug		Instruction fetch bus error exception
Non-debug		Instruction parity check error exception
Debug		Debug breakpoint exception (SDBBP)
Non-debug		Conditional trap exception
Non-debug		System call trap exception
Non-debug		Control or coprocessor unusable exception
Non-debug		Reserved instruction exception
Debug		Debug data break load exception match address only (DDBLA)
Debug		Debug data break store exception (DDBS)
Non-debug		Data load address error exception
Non-debug		Data store address error exception
Non-debug		Custom engine execute exception
Non-debug		Coprocessor z execute exception
Non-debug		Data access bus error exception
Non-debug	**Lowest**	Internal software interrupt

- **Debug Exception Vector Location**

  The location of the debug exception handler depends on the probe enable (ProbEn) field of DREG register. If the probe is enabled, the debug exception vector address is located the start of debug memory segment (***dmseg***, 0xFF00_0000). This means the debug exception can directly start from the external probe memory. If the probe does not active, the exception vector will change to the normal exception table, as shown in Table 6-3.

Table 6-3 Debug exception vector address

$DREG_{ProbEn}$	$DREG_{IceDis}$	Vector Address
0	x	{$EXCPVec_{Base}$, 16'h0} + *0x1FC*
1	1	{$EXCPVec_{Base}$, 16'h0} + *0x1FC*
1	0	*0xFF00_0000*

- **General Debug Exception Processing**

Operation:

```
DEPC ← PC
$DREG_{DM}$ ← 1
$DREG_{DSS, DBP, DDBL, DDBS, DIB, DINT}$ ← DebugExceptionType
if ( $DREG_{ProbEn}$ == 1 && $DREG_{IceDis}$ == 0) then
    PC ← 0xFF00_0000
else
    PC ← {$EXCPVec_{Base}$, 16'h0} + 0x1FC
```

- **Debug Breakpoint Exception (SDBBP)**

The Debug Breakpoint exception is caused by the execution of the SDBBP instruction. The DEPC register will save the address of the SDBBP. Thus, the debug software may need to modify the DEPC to next instruction before debug exception return.

- **Debug Instruction Break Exception (DIB)**

The Debug Instruction Break exception is caused by the SJTAG asserting the *ibrk* signal. The SJTAG will monitor the instruction fetch address and compare it to the instruction breakpoint configuration registers. If the address matches the breakpoint, it will assert the *ibrk* signal and latch the breakpoint number.

- **Debug Data Break Load/Store Exception (DDBS/DDBL)**

The Debug Data Breakpoint exception is caused by the SJTAG asserting the *dbrk* signal. SJTAG can be configured to compare both address and data (DDBLV/DDBS) or just compare address only(DDBL/DDBS). Both the DDBS and DDBL are precise exception while DDBLV saves the next instruction's pc as DEPC.

- **Debug Single Step Exception (DSS)**

When the SSEn field of control register DREG is set, the single step mode is enabled and then a debug Single Step exception occurs each time the processor taken a single step executing in Non-Debug Mode. The DEPC register point to the instruction on which the Debug Single Step exception occurred.

- **Debug Interrupt Exception (DINT)**

  The properties of the debug interrupt are very similar to the normal interrupt. The external module can assert at any time, so the core will enter debug exception as soon as possible. The DEPC will store the address of the address that be interrupted when processor enter debug mode.

  If the processor is in debug mode, the debug interrupt will not be taken.

6.1.8 Debug Mode Exceptions

When the processor enters the debug mode, there is a little difference of exception handling. Some exception will be blocked in debug mode and processor exception behavior is different to normal mode exceptions.

- **Exception Allowed in Debug Mode**

  At debug mode, the debug exceptions and interrupt related instructions would be blocked. Table 4-2 lists the behavior of the exceptions during debug mode. Other exception will cause a re-entry to debug mode with a debug mode exception code (DExc) which shown in Table **6-4**.

Table 6-4 Debug mode exception priority

Priority	Exception	Core Behavior
Highest	Reset	Reset
	Debug single step exception (DSS)	Blocked
	Debug Interrupt (DINT)	Blocked
	Debug data break load exception match both address and data (DDBLV)	Blocked
	Non-maskable interrupt NMI	Blocked
	External hardware interrupt exception	Blocked
	Debug instruction address break exception (DIB)	Blocked
	Instruction fetch address error exception	Re-enter Debug Mode
	Instruction fetch bus error exception	Re-enter Debug Mode
	Instruction parity check error exception	Re-enter Debug Mode
	Debug breakpoint exception (SDBBP)	Blocked
	Conditional trap exception	Re-enter Debug Mode
	System call trap exception	Re-enter Debug Mode
	Control or coprocessor unusable exception	Re-enter Debug Mode
	Reserved instruction exception	Re-enter Debug Mode
	Debug data break load exception match address only (DDBLA)	Blocked
	Debug data break store exception (DDBS)	Blocked
	Data load address error exception	Re-enter Debug Mode

Priority	Exception	Core Behavior
	Data store address error exception	Re-enter Debug Mode
	Custom engine execute exception	Re-enter Debug Mode
	Coprocessor z execute exception	Re-enter Debug Mode
	Data access bus error exception (precise)	Re-enter Debug Mode
	Data access bus error exception (imprecise)	Blocked
Lowest	Internal software interrupt	Blocked

- **Debug Mode Exception Processing**

  At debug mode, some exception will cause processor re-enter debug exception and DEPC will update to the address of the instruction that cause this new exception. The DEPC must be saved before the exception occurs.

Operation:

```
DEPC ← PC
DREG_DM ← 1
DREG_DSS, DBP, DDBL, DDBS, DIB, DINT ← 0
DREG_DExcCode ← DebugExceptionType
if ( DREG_ProbEn == 1 && DREG_IceDis == 0) then
    PC ← 0xFF00_0000
else then
    PC ← {EXCPVec_Base, 16'h0} + 0x1FC
```

Table 6-5 Debug exception code encoding

Exception	Exception Code
Reset	Reserved
NMI	Reserved
AdEL-instruction	2
-	3
-	4
BusEL-instruction	5
P-EL	6
SYSCALL	7
CCU	8
RI	9
Trap	10
AdEL-data	11
AdES	12
-	13
-	14
-	15

Exception	Exception Code
CeE	16
CpE	17
BusEL-data	18
SWI	Reserved
Interrupt	Reserved

6.1.9 CR Debug Registers

There is three debug registers in control registers: DREG, DEPC and DSAVE. The register number is listed in Table 6-6. Table 6-7, Table 6-8 and Table 6-9 show the bit field of these registers.

Table 6-6 CR debug registers

Control Register Number	Name	Description
29	DREG	Debug register
30	DEPC	Debug exception program counter register
31	DSAVE	Debug save template register

31	29	28	27	26	25	24	23 14	13 9	8	7	6	5	4	3	2	1	0
-	IceDis	InitOk	ProbeEn	DM	IBusEP	DBusEP	-	DExeCode	SSEn	-	DDBLV	DINT	DIB	DDBS	DDBL	DB	DSS

Fig 6-4 Debug register format

Table 6-7 Debug register field description

Bits	Name	Description	R/W	Reset
31:30	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
29	**IceDis**	Disable SJTAG module	R/W	0
28	**InitOk**	Indicate the boot program is already finish system initialization. System is ready for download.	R/W	0
27	**ProbeEn**	Indicate Probe is enabled	R	0
26	**DM**	Debug Mode	R	0
25	**IBusEP**	Indicates if a Instruction Bus Error Exception pending	R	0
24	**DBusEP**	Indicates if a Data Bus Error Exception pending	R	0
23:14	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
13:9	**DExcCode**	Indicates the cause of latest exception in Debug Mode	R	Undef
8	**SSEn**	Single-step enable	R/W	0
7:6	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
6	**DDBLV**	Indicate a Debug Data Break Load exception match both address and data occurred	R	Undef
5	**DINT**	Indicate a Debug Interrupt exception occurred	R	Undef

Bits	Name	Description	R/W	Reset
4	**DIB**	Indicate a Debug Instruction Break exception occurred	R	Undef
3	**DDBS**	Indicate a Debug Data Break Store exception occurred	R	Undef
2	**DDBLA**	Indicate a Debug Data Break Load exception match address occurred	R	Undef
1	**DBP**	Indicate a Debug Breakpoint exception (caused by SDBBP) occurred	R	Undef
0	**DSS**	Indicate a Debug Single Step exception occurred	R	Undef

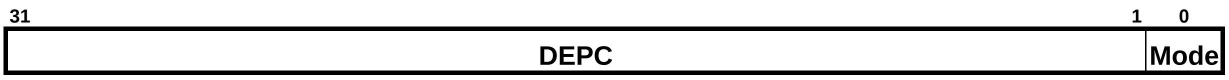


Fig 6-5 Debug exception program counter register format

Table 6-8 Debug exception program counter register field description

Bits	Name	Description	R/W	Reset
31:1	**DEPC**	Debug Exception Program Counter. This register would save value the program counter when a debug exception occurs. When a DRTE instruction is executed, DEPC register value would be restored to program counter register.	R/W	Undef
0	**Mode**	Indicates the instruction mode (32-bit or 16-bit) of the instruction at which a debug exception occurs. For exceptions that occur after D stage: 1 : indicates that exception instruction is a pce or 16-bit instruction 0 : indicates that exception instruction is a 32-bit instruction Otherwise, this bit would be undefined.	R	Undef

Fig 6-6 Debug exception save template register format

Table 6-9 Debug exception save register field description

Bits	Name	Description	R/W	Reset
31:0	**DSAVE**	Debug Exception Save contents. This register can be used as a temporal register when in a debug service routine	R/W	Undef

6.1.10 Debug Instructions

There are two additional instruction for the debug: SDBBP (Software Debug Breakpoint) and DRTE (Return from Debug Exception).

- **SDBBP (Software Debug Breakpoint)**

  Purpose:

  To cause a Debug Breakpoint exception

  Description:

  If CPU is not in debug mode, a debug exception occurs. If the CPU is in debug mode, it will not cause any exception.

  Operation:

```
if(DREG_DM ==0) then
DEPC ← PC
DREG_DM ← 1
DREG_DBP ← 1
if ( DREG_ProbEn == 1 && DREG_IceDis == 0) then
    PC ← 0xFF00_0000
    else
      PC ← {EXCPVec_Base, 16'h0} + 0x1FC
```

- **DRTE (Return from Debug Exception)**

  Description:

  This instruction returns processor from debug mode to normal mode and restore the PC from DEPC.

  Restrictions:

  Since this instruction is a control register instruction, executing this instruction in user mode with CRA bit false would cause a control register unusable exception. It is suggested that programmers to use this instruction in debug mode.

  Operation:

```
PC ← DEPC
DREG_DM ← 0
```

6.2 SJTAG Breakpoint Unit

The SJTAG Breakpoint unit will assert the instruction breakpoint and data breakpoint to the processor according to the configuration registers. The breakpoint configuration, control and status registers locate in debug register segment (drseg) that can be directly accessed by processor memory space. The number of breakpoints in the breakpoint unit is configurable. The default is two instruction breakpoints and two data breakpoints. The maximal number is 15 instruction breakpoints and 15 data breakpoints.

6.2.1 SJTAG Registers

The SJTAG contain three parts:

- Debug control register (DCR) provides the probe status.
- Instruction breakpoint related registers contain the status of each instruction breakpoint and the address to match in each breakpoint.
- Data breakpoint related registers contain the status of each data breakpoint and the address/data to match in each breakpoint.

Table 6-10 Registers in debug segment

Name	Offset in *drseg*	Description
DCR	*0x0000*	Debug Control Register
IBS	*0x1000*	Instruction Breakpoint Status
IBAn	*0x2000 + 0x100 * (n-1)*	Instruction Breakpoint Address n
IBMn	*0x2004 + 0x100 * (n-1)*	Instruction Breakpoint Address Mask n
IBASIDn	*0x2008 + 0x100 * (n-1)*	Instruction Breakpoint ASID n
IBCn	*0x200C + 0x100 * (n-1)*	Instruction Breakpoint control n
DBS	*0x3000*	Data Breakpoint Status
DBAn	*0x4000 + 0x100 * (n-1)*	Data Breakpoint Address n
DBMn	*0x4004 + 0x100 * (n-1)*	Data Breakpoint Address Mask n
DBASIDn	*0x4008 + 0x100 * (n-1)*	Data Breakpoint ASID n
DBCn	*0x400C + 0x100 * (n-1)*	Data Breakpoint Control n
DBVn	*0x4010 + 0x100 * (n-1)*	Data Breakpoint Value n

6.2.2 Debug Control Register (DCR)

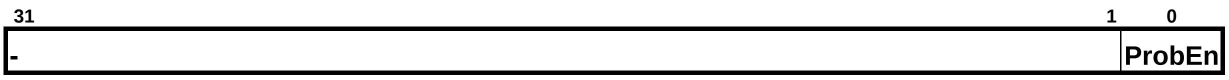


Fig 6-7 Debug control register format

Table 6-11 Debug control register field description

Bits	Name	Description	R/W	Reset
31:1	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
0	**ProbEn**	Indicates value of the ProbeEn value in the SJCR register	R	SJCR

6.2.3 Instruction Breakpoint Registers

The instruction breakpoint related registers contain an instruction breakpoint status register (IBS) for all instruction breakpoints. For each breakpoint there are an instruction breakpoint control (IBC), an instruction breakpoint address (IBA), an instruction breakpoint mask (IBM) and an instruction ASID (IBASID) register.

As shown in Table **6-12**, the IBS includes the information of the total number of breakpoint implemented in the breakpoint unit

The IBA, IBM, and IBC contain the value to match the instruction breakpoint.

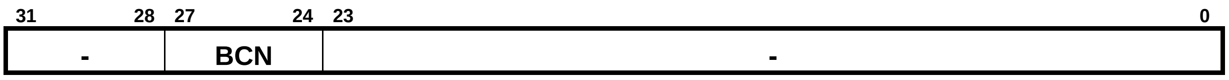


Fig 6-8 Instruction breakpoint status register format

Table 6-12 Instruction breakpoint status register field description

Bits	Name	Description	R/W	Reset
31:28	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
27:24	**BCN**	Number of instruction breakpoint implemented	R	Preset
23:0	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0

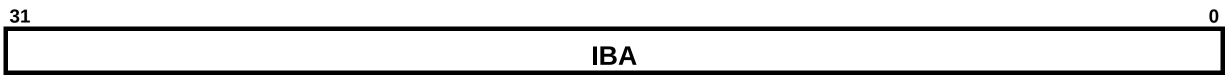


Fig 6-9 Instruction breakpoint address register format

Table 6-13 Instruction breakpoint address register field description

Bits	Name	Description	R/W	Reset
31:0	**IBA**	The address value to be compared during instruction fetch	R/W	Undef

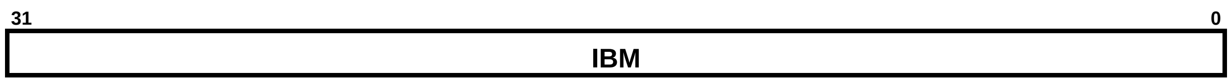


Fig 6-10 Instruction breakpoint mask register format

Table 6-14 Instruction breakpoint mask register field description

Bits	Name	Description	R/W	Reset
31:0	**IBM**	The address mask used for instruction fetch address compare	R/W	Undef

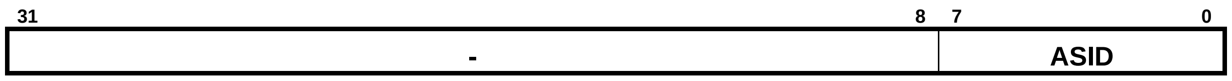


Fig 6-11 Instruction breakpoint ASID register format

Table 6-15 Instruction breakpoint ASID register field description

Bits	Name	Description	R/W	Reset
31:8	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
7:0	**ASID**	The ASID value to be compare during instruction fetch	R/W	Undef

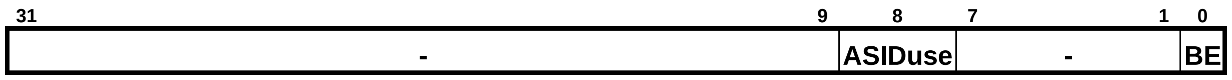


Fig 6-12 Instruction breakpoint control register format

Table 6-16 Instruction breakpoint control register field description

Bits	Name	Description	R/W	Reset
31:9	Reserve	Reserved; read will return zero and write must be zero.	R0/W0	0
8	ASIDuse	Use ASID value in compare for instruction breakpoint	R/W	Undef
7:1	Reserve	Reserved; read will return zero and write must be zero.	R0/W0	0
0	BE	Use instruction breakpoint n as breakpoint	R/W	0

6.2.4 Instruction Breakpoint Match

The instruction breakpoint match condition is shown below:

IB_match =
(!**IBC**$_{\text{ASIDuse}}$ || (ASID == **IBASID**)) &&
((**IBM** | (~(PC[31:0] ^ **IBA**))) == 32'hFFFF_FFFF)

When the instruction breakpoint matches, breakpoint unit will assert instruction breakpoint exception request to processor if this breakpoint is enabled (**IBC**$_{\text{BE}}$ bit is set).

6.2.5 Data Breakpoint Registers

The data breakpoint related registers contain a data breakpoint status register (DBS) for all instruction breakpoints. For each breakpoint there is a data breakpoint control (DBC), a data breakpoint address (DBA), a data breakpoint mask (DBM), a data ASID (DBASID), and a data breakpoint value register.

As shown in Table 6-17, the DBS includes the information of

- The total number of data breakpoint implemented in the breakpoint unit

The DBA, DBM, DBASID, DBV and DBC contain the value to match the data breakpoint.

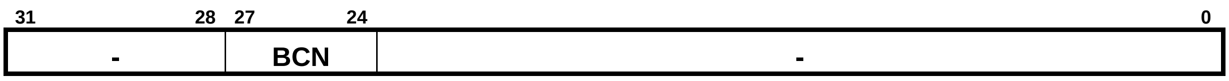


Fig 6-13 Data breakpoint status register format

Table 6-17 Data breakpoint status register field description

Bits	Name	Description	R/W	Reset
31:28	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
27:24	**BCN**	Number of data breakpoint implemented	R	Preset
23:0	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0

Fig 6-14 Data breakpoint address register format

Table 6-18 Data breakpoint address register field description

Bits	Name	Description	R/W	Reset
31:0	**DBA**	The address value to be compared during data fetch address	R/W	Undef

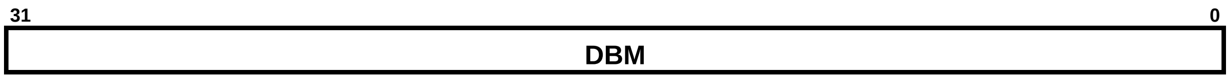


Fig 6-15 Data breakpoint mask register format

Table 6-19 Data breakpoint mask register field description

Bits	Name	Description	R/W	Reset
31:0	**DBM**	The address mask used for data fetch address compare	R/W	Undef

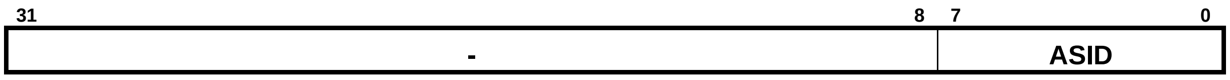


Fig 6-16 Data breakpoint ASID register format

Table 6-20 Data breakpoint ASID register field description

Bits	Name	Description	R/W	Reset
31:8	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
7:0	**ASID**	The ASID value to be compare during data fetch	R/W	Undef

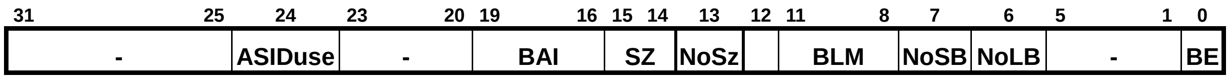


Fig 6-17 Data breakpoint control register format

Table 6-21 Data breakpoint control register field description

Bits	Name	Description	R/W	Reset
31:25	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
24	**ASIDuse**	Use ASID value in compare for instruction breakpoint	R/W	Undef
23:20	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
19:16	**BAI**	Byte access ignore	R/W	Undef
15:14	**SZ**	Access size to match 2’b00: byte 2’b01: halfword 2’b10: tribyte 2’b11 word	R/W	2’b11
13	**NoSZ**	Do not match the access size	R/W	1
12	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
11:8	**BLM**	Byte lane mask for value compare on data breakpoint	R/W	Undef
7	**NoSB**	Controls whether match data breakpoint on a store access	R/W	Undef
6	**NoLB**	Controls whether match data breakpoint on a load access	R/W	Undef
5:1	**Reserve**	Reserved; read will return zero and write must be zero.	R0/W0	0
0	**BE**	Use instruction breakpoint n as breakpoint	R/W	0

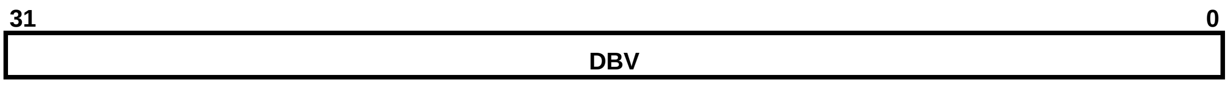


Fig 6-18 Data breakpoint value register format

Table 6-22 Data breakpoint value register field description

Bits	Name	Description	R/W	Reset
31:0	**DBV**	Data value to be compared for data breakpoint	R/W	Undef

6.2.6 Data Breakpoint Match

The data breakpoint match conditions are shown below:

DB_match =
((*load_access* && !**DBC**$_{\text{NoLB}}$)||(*store_accee* && !**DBC**$_{\text{NoSB}}$))&&
DB_addr_match && (DB_no_value || DB_value_match)

DB_addr_match =
(!**DBC**$_{\text{ASIDuse}}$ || (ASID == **DBASID**))&&
(**DBC**$_{\text{NoSZ}}$ || (Size == **DBC**$_{\text{SZ}}$))&&
((**DBM** | (~ *data_addr* ^ **DBA**)) == 32hFFFF_FFFF) &&
((**DBC**$_{\text{BAI}}$ | *data_bytelane*) == 4'b1111)

DB_no_value =
(**DBC**$_{\text{BLM}}$ | **DBC**$_{\text{BAI}}$ | ~*data_bytelane*) == 4'b1111

DB_value_match =
((data[31:24] == **DBV**[31:24])||!bytelane[3]||**DBC**$_{\text{BLM[3]}}$||**DBC**$_{\text{BAI[3]}}$)&&
((data[23:16] == **DBV**[23:16])||!bytelane[2]||**DBC**$_{\text{BLM[2]}}$||**DBC**$_{\text{BAI[2]}}$)&&
((data[15:8] == **DBV**[15:8]) ||!bytelane[1]||**DBC**$_{\text{BLM[1]}}$||**DBC**$_{\text{BAI[1]}}$)&&
((data[7:0] == **DBV**[7:0]) ||!bytelane[0]||**DBC**$_{\text{BLM[0]}}$||**DBC**$_{\text{BAI[0]}}$)

7 Bus Interface Unit (BIU)

The S^{+}core BIU implements a fully-compliant AHB 2.0 bus master interface and incorporates a write buffer to increase system performance. The BIU is the link between I-Cache controller, D-Cache controller and SJTAG controller with the external AHB interface. Fig 8-1 shows the block diagram of BIU interface.

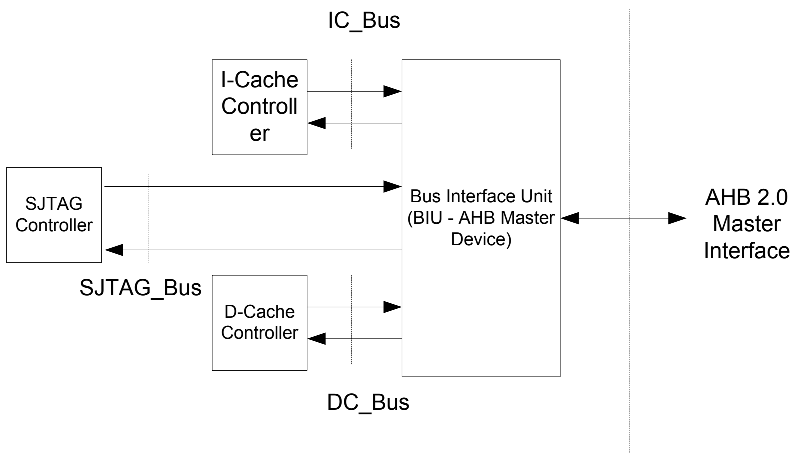


Fig 7-1 BIU interface block diagram

The S^{+}core macrocell supports an *Advanced* Microprocessor *Bus Architecture* (AMBA) *Advanced High-performance Bus* (AHB) interface. The AHB of AMBA interface addresses the requirements of high-performance synthesizable designs, including:

- single clock edge operation (rising edge)
- unidirectional (nontristate) buses
- mapped burst transfers
- single-cycle bus master handover.

See the *AMBA Specification* (Rev 2.0) for full details of this bus architecture.

7.1 AHB Overview

The AHB architecture is based on separate cycles for address and data. The address and control values for an access are broadcast from the rising edge of **HCLK** in the cycle before the data is expected to be read or written. During this data cycle, the address and control values for the next cycle are driven out. This leads to a fully pipelined address architecture.

When an access is in its data cycle, a slave can wait the access by driving the **HREADY** response LOW. This has the effect of stretching the current data cycle and therefore the pipelined address and control for the next access is also stretched. This creates a system where all AHB masters and slaves sample

HREADY on the rising edge of the **HCLK** to determine whether an access has completed and a new address can be sampled or driven out.

7.2 Endian Issue

Computer systems store data and transfer data differently. If the data are viewed by the application as 8-bit, but are transferred over the bus as 32-bit word, the four bytes must be in the opposite order for a big endian bus and a little endian bus. If the data had a halfword width of 16 bits, the transfer would require the upper 16 bits of the word size bus to swap with the lower 16 bits. If the word width of the data were 32 bits, no swapping would be required.

Byte Location Definition:

Byte(0) refers to the bytes stored at address that are multiples of four. Byte(1) is defined as the byte after Byte(0), and so forth up to Byte(3). For example: 0x04: Byte(0), 0x05: Byte(1), 0x06: Byte(2), 0x07: Byte(3)

Byte Access: Data:

Address [1:0]	Little Endian Mode	Big Endian Mode
00 (Byte 0 Data at AHB Bus)	HRDATA [7:0] / HWDATA [7:0]	HRDATA [31:24] / HWDATA [31:24]
01 (Byte 1 Data at AHB Bus)	HRDATA [8:15] / HWDATA [8:15]	HRDATA [23:16] / HWDATA [23:16]
10 (Byte 2 Data at AHB Bus)	HRDATA [16:23] / HWDATA [16:23]	HRDATA [15:8] / HWDATA [15:8]
11 (Byte 3 Data at AHB Bus)	HRDATA [31:24] / HWDATA [31:24]	HRDATA [7:0] / HWDATA [7:0]

Halfword Access:

Address [1]	Little Endian Mode	Big Endian Mode
0 (Byte 0-1 Data at AHB Bus)	HRDATA [15:0] / HWDATA [15:0]	HRDATA [31:16] / HWDATA [31:16]
1 (Byte 2-3 Data at AHB Bus)	HRDATA [31:16] / HWDATA [31:16]	HRDATA [15:0] / HWDATA [15:0]

Word Access:

Address [1:0]	Little Endian Mode	Big Endian Mode
00 (Byte 3-0 Data at AHB Bus)	HRDATA [31:0] / HWDATA [31:0]	HRDATA [31:0] / HWDATA [31:0]

For Example:

Word data for writing operation (Register Content): 0x12345678 [Bit 31:0]

Halfword data for writing operation: 0xCCDD [Bit 15:0]

Byte data for writing operation: 0xAA [Bit 7:0]

If Address [1:0] = 00

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HWDATA[31:0])			
	Byte-3 /Bit[31:24]	Byte2 /Bit[23:16]	Byte-1 /Bit[15:8]	Byte-0 /Bit[7:0]	Byte-0 /Bit[31:24]	Byte-1 /Bit[23:16]	Byte-2 /Bit[15:8]	Byte-3 /Bit[7:0]
Word Write	0x12	0x34	0x56	0x78	0x12	0x34	0x56	0x78
Halfword Write			0xCC	0xDD	0xCC	0xDD		
Byte Write				0xAA	0xAA			

If Address [1:0] = 01

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HWDATA[31:0])			
	Byte-3 /Bit[31:24]	Byte2 /Bit[23:16]	Byte-1 /Bit[15:8]	Byte-0 /Bit[7:0]	Byte-0 /Bit[31:24]	Byte-1 /Bit[23:16]	Byte-2 /Bit[15:8]	Byte-3 /Bit[7:0]
Byte Write			0xAA			0xAA		

If Address [1:0] = 10

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HwDATA[31:0])			
	Byte-3 /Bit[31:24]	Byte2 /Bit[23:16]	Byte-1 /Bit[15:8]	Byte-0 /Bit[7:0]	Byte-0 /Bit[31:24]	Byte-1 /Bit[23:16]	Byte-2 /Bit[15:8]	Byte-3 /Bit[7:0]
Halfword Write	0xCC	0xDD					0xCC	0xDD
Byte Write		0xAA					0xAA	

If Address [1:0] = 11

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HWDATA[31:0])			
	Byte-3 /Bit[31:24]	Byte2 /Bit[23:16]	Byte-1 /Bit[15:8]	Byte-0 /Bit[7:0]	Byte-0 /Bit[31:24]	Byte-1 /Bit[23:16]	Byte-2 /Bit[15:8]	Byte-3 /Bit[7:0]
Byte Write	0xAA							0xAA

If Address [0] = 0

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HWDATA[31:0])			
	Byte-3 /Bit[31:24]	**Byte2 /Bit[23:16]**	**Byte-1 /Bit[15:8]**	**Byte-0 /Bit[7:0]**	**Byte-0 /Bit[31:24]**	**Byte-1 /Bit[23:16]**	**Byte-2 /Bit[15:8]**	**Byte-3 /Bit[7:0]**
Tri-Byte Write	▨	0x34	0x56	0x78	0x12	0x34	0x56	▨

If Address [0] = 1

	Little Endian Mode (HWDATA[31:0])				Big Endian Mode (HWDATA[31:0])			
	Byte-3 /Bit[31:24]	**Byte2 /Bit[23:16]**	**Byte-1 /Bit[15:8]**	**Byte-0 /Bit[7:0]**	**Byte-0 /Bit[31:24]**	**Byte-1 /Bit[23:16]**	**Byte-2 /Bit[15:8]**	**Byte-3 /Bit[7:0]**
Tri-Byte Write	0x12	0x34	0x56	▨	▨	0x34	0x56	X078

8 Instruction Set Overview

32/16-bit hybrid instruction is an important feature of S+core instruction set. Two P-bits are utilized to distinguish four instruction formats: one 32-bit instruction, two 16-bit instructions, parallel condition execution instructions and undefined instructions. In current implementation of S+core ISA, the positions of P-bits are 31st and 15th bit in 32-bit instruction formats, as shown in Fig 8-1.

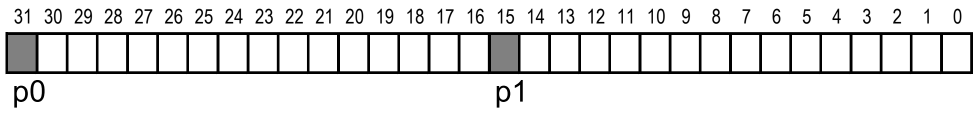


Fig 8-1 Positions of P-bits

Table 8-1 Instruction Formats Denoted by P-bits

p0	p1	Meaning	Format Notation
1	**1**	**32-bit Instruction Format**	**32BF**
0	**0**	**16bit + 16bit Instruction Format**	**16BF**
		bit 31~bit16 : higher 16-bit instruction 1	
		bit 15~bit0 : lower 16-bit instruction 2	
0	**1**	**Parallel Conditional Execution (PCE) instructions**	**PCEF**
		bit 31~bit16 : higher 16-bit instruction 1, executed only when Flag T =True	
		bit 15~bit0 : lower 16-bit instruction 1, executed only when Flag T = False	
1	**0**	**Undefined. Issue Parity-Check Error Exception**	**UDEF**

8.1 32/16-Bit Hybrid Instruction

When executing programs, the two P-bits in a 32-bit instruction word denote the format of instruction set. For example, if a seven 32-bit word program has {p0, p1} pattern as shown in Table 8-2, the processor will recognize the instruction format of this program as Fig 8-2. The execution sequence is shown in Fig 8-3, if no branch, jump exception and interrupt.

Table 8-2 Example of P-bits Sequence

p0	p1	Format
1	1	32BF
1	1	32BF
0	0	16BF
0	0	16BF
1	1	32BF
0	0	16BF
1	1	32BF

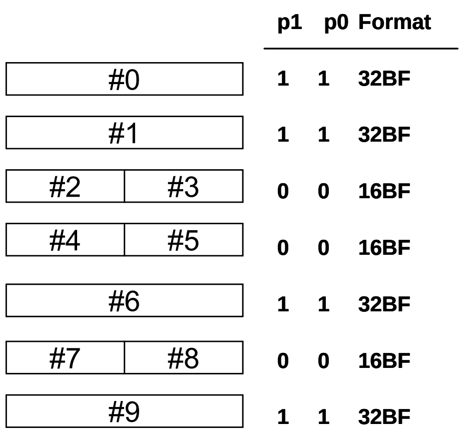


Fig 8-2 Example of Format Reorganization

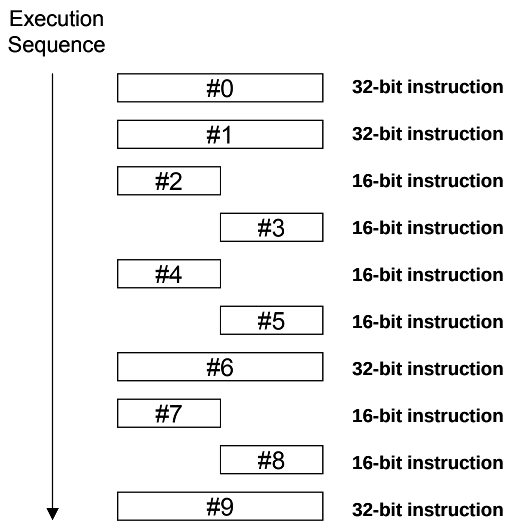


Fig 8-3 Example of 32/16-bit Hybrid Instruction Execution

8.2 Parallel Conditional Execution

Parallel conditional execution (PCE) is a Sunplus-patent-pended feature to avoid the branch penalty in traditional computer architecture. In traditional computer architecture, the conditional branch always plays an important role in program sequence control, but the processor suffers from pipeline flush caused by un-continuous execution address. Nevertheless, the PCE offers a method to smoothly execute program without branch penalty. Compared to conditional branch, the PCE utilized “select route” method instead of “jump if true” method. Since the 32/16-bit hybrid instruction offer an instruction format to contain two 16-bit instructions in a 32-bit format, we can treat one 16-bit instruction as first route for logical true, while the another one is the route for logical false. Therefore, the program execution can be smoothly without branch penalty.

For example, if a seven 32-bit word program has {p0, p1} pattern as shown in Table 8-3. The processor will recognize the instruction format of this program as Fig 8-4. In S^+core, the flag T is utilized for the route selection for PCE. The Fig 8-5(a) shows the execution sequence when flag T=1 (True Route), and the Fig 8-5(b) when T=0 (false route).

Table 8-3 Example of P-bits Sequence with PCE

p0	p1	Format
1	1	32BF
1	1	32BF
0	1	PCEF
0	1	PCEF
1	1	32BF
0	1	PCEF
1	1	32BF

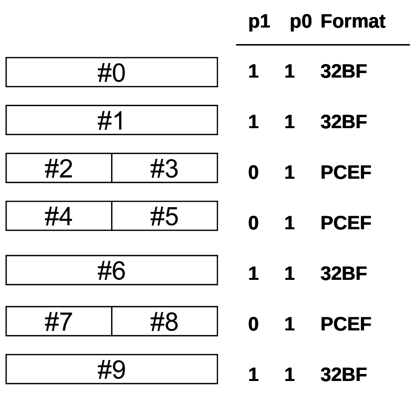


Fig 8-4 Example of Format Reorganization with PCE

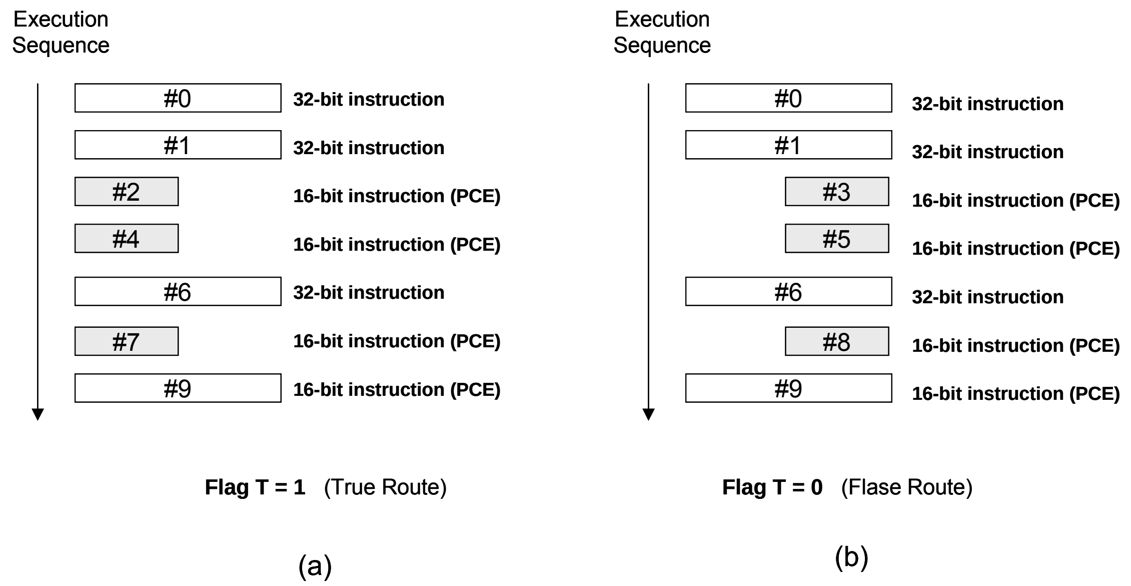


Fig 8-5 Example of PCE Execution (a) Flag T=1 (b) Flag T=0

Assembly programmers use double pipe (||) to specify a pair of PCE execution instructions. For example:

```
ADD! r2, r7 || SUB! r2, r7
```

The add instruction would be executed if T flag is true while the sub instruction would be executed if T flag is false.

8.3 Instruction Format and Encoding

S$^+$core defines two instruction sets, one is 32-bit and the other is 16-bit. In the following sections, the S$^+$core instruction sets are introduced by instruction forms. The instruction formats are shown below:

32-bit instruction set

- Jump instruction format (J-form)
- Conditional Branch instruction format (BC-form)
- Special instruction format (Special-form)
- Immediate instruction format (I-form)
- Register-Immediate instruction format (RI-form)
- Register-Immediate updating instruction format (RIX-form)
- Control register instruction format (CR-form)
- Coprocessor instruction format (Coprocessor)
- Custom Engine Instruction format (CEINST)

16-bit instruction set

- Conditional Branch instruction format (BC-form)
- Jump instruction format (J-form)
- Register instruction format (R-form)
- Short Immediate instruction format (I-form-1)
- Long Immediate instruction format (I-form-2)

Table 8-4 shows descriptions of each instruction format field and Fig 8-6 shows the S^{+}core instructions formats.

Table 8-4 The S^{+}core instruction format fields

Field	Description
OP	The main opcode of the instruction set
Imm	Immediate
Disp	Signed Branch Displacement.
BC	Branch condition. The encoding of BC field specifies the corresponding Condition flag check shown in Fig 8-25 (p.94).
EC	Execution condition. The encoding of EC field specifies the corresponding Condition flag check shown in Fig 8-26(p.95).
rD	Destination register
rA	Source register A
rB	Source register B
func	Function code
SPar	Software parameters
SA	Shift amount. A 5-bit immediate that specifies the shift/rotate amount of a shift/rotate instruction.
BN	Bit number. A 5-bit immediate that specifies the bit in rA that is to be modified by bit operation instructions.
TC	Test condition. The 2-bit encoding of TC field specifies the corresponding condition flag check: 2'b00 specifies Z flag check, 2'b01 specifies N flag check, 2'b10 specifies unconditional set T flag. T flag is set according to the condition flag check result.
code	Software debug break point code
Srn	The n'th special register. Sr0 is CNT, Sr1 is LCR, Sr2 is SCR.
H	This bit specifies that CEH register is accessed in move to/from custom engine instruction
L	This bit specifies that CEL register is accessed in move to/from custom engine instruction
USD	User defined parameter. Field specified by this notation can be either immediate or destination register number (depends on user definition).
CR_op	Control register instruction operation code.
Cache_op	Cache operation code. The cache instruction operates according to this field.
CrD	Coprocessor destination register
CrA	Coprocessor source register A

Field	Description
CrB	Coprocessor source register B
COP-code	Coprocessor code. The coprocessor operation instruction operates according to this field.
CP#	Coprocessor number
Sub-OP	Coprocessor sub opcode. This opcode distinguishes different coprocessor instruction.
CU	Conditional update bit. If CU=1, the instruction would update Condition flag. If the suffix ".c" is added to a 32-bit instruction, the CU bit of that instruction would be set to 1. Note that for 16-bit instructions, every condition flag update instruction would update condition flag by nature.
LK	Link bit. If true, the PC of next instruction (PC+4/PC+2) would be stored into link register after a jump or a branch
S	Sign bit. Indicates an "add exponent" instruction (addei!) whether to perform addition or subtraction.
Exp	Exponent of a radix 2 number. This field indicates the exponent of the radix 2 that is to be added in an "add exponent" instruction.
g0	The lower registers of the general purpose register file. Register set g0 is composed of r0~r15.
g1	The higher registers of the general purpose register file. Register set g1 is composed of r16~r30.
rD_{g0}	Destination register that belongs to register g0 set
rD_{g1}	Destination register that belongs to register g1 set
rA_{g0}	Source register A that belongs to register g0
rA_{g1}	Source register B that belongs to register g1
X[n:m]	Selection of bits n through m of bit string X

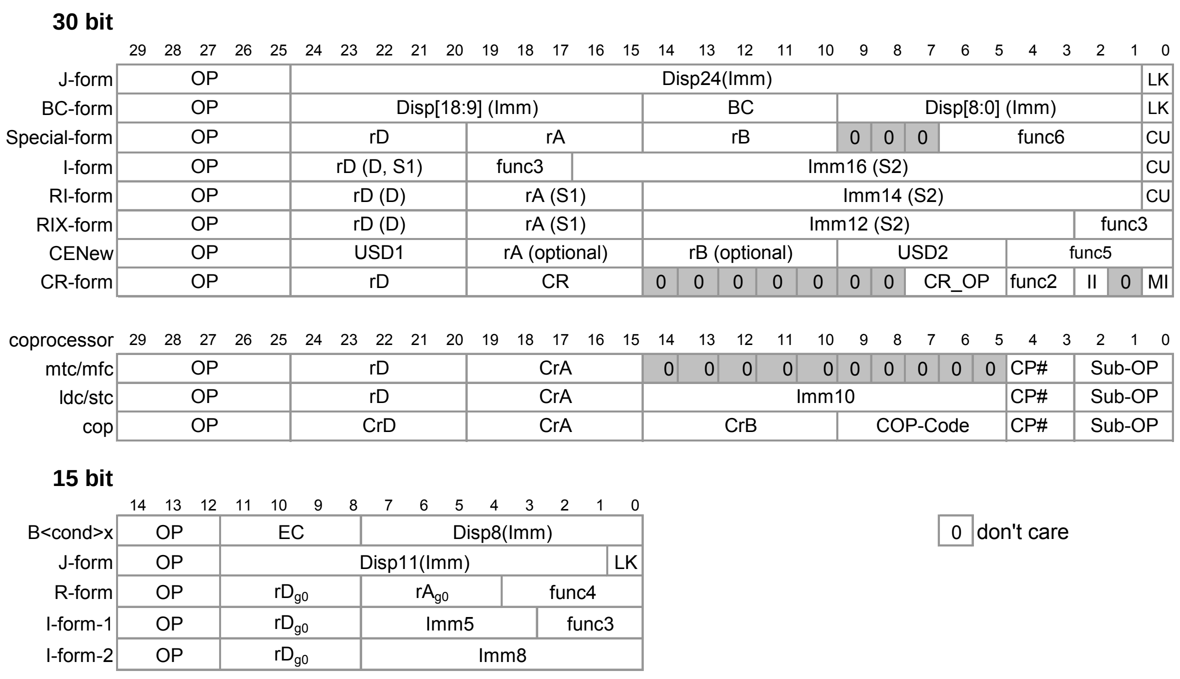

30 bit
J-form: OP | Disp24(Imm) | LK
BC-form: OP | Disp[18:9] (Imm) | BC | Disp[8:0] (Imm) | LK
Special-form: OP | rD | rA | rB | 0 0 0 | func6 | CU
I-form: OP | rD (D, S1) | func3 | Imm16 (S2) | CU
RI-form: OP | rD (D) | rA (S1) | Imm14 (S2) | CU
RIX-form: OP | rD (D) | rA (S1) | Imm12 (S2) | func3
CENew: OP | USD1 | rA (optional) | rB (optional) | USD2 | func5
CR-form: OP | rD | CR | 0 0 0 0 0 0 0 | CR_OP | func2 | II | 0 | MI
coprocessor
mtc/mfc: OP | rD | CrA | 0 0 0 0 0 0 0 0 0 0 | CP# | Sub-OP
ldc/stc: OP | rD | CrA | Imm10 | CP# | Sub-OP
cop: OP | CrD | CrA | CrB | COP-Code | CP# | Sub-OP
15 bit
B<cond>x: OP | EC | Disp8(Imm)
J-form: OP | Disp11(Imm) | LK
R-form: OP | rDg0 | rAg0 | func4
I-form-1: OP | rDg0 | Imm5 | func3
I-form-2: OP | rDg0 | Imm8
0 don't care

Fig 8-6 The S^+core instruction format

Fig 8-7 ~Fig 8-11 shows the encoding of 32-bit instruction set.

opcode 27..25

		0	1	2	3	4	5	6	7
29..28		000	001	010	011	100	101	110	111
0	0 0	Special-func	I-form-1	Jx inst.	RIX-form-1	B<cond>x	I-form-2	CR/CP	RIX-form-2
1	0 1	ADDRI				ANDRI	ORRI		
2	1 0	LW	LH	LHU	LB	SW	SH	LBU	SB
3	1 1	CACHE				CENew			

Fig 8-7 S^+core Encoding of the 32-bit Opcode field

3..1

		0	1	2	3	4	5	6	7
6..4		000	001	010	011	100	101	110	111
0	0 0 0	NOP	SYSCALL	TRAP<cond>	SDBBP	BR<cond>x		ALW	ASW
1	0 0 1	ADD	ADDC	SUB	SUBC	CMP<TC>.c	CMPZ<TC>.c		NEG
2	0 1 0	AND	OR	NOT	XOR	BITCLR.c	BITSET.c	BITTST.c	BITTGL.c
3	0 1 1	SLL		SRL	SRA	ROR	RORC.c	ROL	ROLC.c
4	1 0 0	MUL	MULU	DIV	DIVU	MFCE inst.	MTCE inst.		
5	1 0 1	MFSR	MTSR	T<cond>	MV<cond>	EXTSB	EXTSH	EXTZB	EXTZH
6	1 1 0	LCB	LCW		LCE	SCB	SCW		SCE
7	1 1 1	SLLI		SRLI	SRAI	RORI	RORIC.c	ROLI	ROLIC.c

Fig 8-8 S^+core Encoding of the func6 field of 32-bit Special-form

I-form-1 19..17

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
ADDI		CMPI.c		ANDI	ORI	LDI	

I-form-2 19..17

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
ADDIS				ANDIS	ORIS	LDIS	

Fig 8-9 S^+core Encoding of the func3 field of 32-bit I-form

RIX-form-1 2..0

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
LW.B	LH.B	LHU.B	LB.B	SW.B	SH.B	LBU.B	SB.B

RIX-form-2 2..0

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
LW.A	LH.A	LHU.A	LB.A	SW.A	SH.A	LBU.A	SB.A

Fig 8-10 S^+core Encoding of the 32-bit RIX-form func3 field

CR-form 7..5

		0	1	2	3	4	5	6	7
9..8		000	001	010	011	100	101	110	111
0	0 0	STLB	MFTLB	MTPTLB	MTRTLB	RTE	DRTE	SLEEP	
1	0 1								
2	1 0								
3	1 1								

Fig 8-11 S^+core Encoding of the 32-bit Cop0-Code field

Fig 8-12~Fig 8-14 shows the encoding of 16-bit instruction set. The notation “!” is added to indicate a 16-bit instruction.

opcode 14..12

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
R-form-1		R-form-2	J-form	B<cond>!	LDIU!	I-form-1a	I-form-1b

Fig 8-12 S$^+$core Encoding of the 16-bit Op field

R-form-1 2..0

	0	1	2	3	4	5	6	7
3	000	001	010	011	100	101	110	111
0	NOP!	MLFH!	MHFL!	MV!	BR<cond>!	T<cond>!		
1	SLL!	ADDC!	SRL!	SRA!	BR<cond>L!			

R-form-2 2..0

	0	1	2	3	4	5	6	7
3	000	001	010	011	100	101	110	111
0	ADD!	SUB!	NEG!	CMP!	AND!	OR!	NOT!	XOR!
1	LW!	LH!	POP!	LBU!	SW!	SH!	PUSH!	SB!

Fig 8-13 S$^+$core Encoding of the 16-bit R-form func4 field

I-form-1a 2..0

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
addei inst.	SLLI!	SDBBP!	SRLI!	BITCLR!	BITSET!	BITTST!	BITTGL!

I-form-1b 2..0

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
LWP!	LHP!		LBUP!	SWP!	SHP!		SBP!

Fig 8-14 S$^+$core Encoding of the 16-bit I-form-1a and I-form-1b func3 field

Table 8-5 shows the naming conventions of the S$^+$core instruction set.

Table 8-5 Instruction naming conventions

R	reg	.c	condition update	.B	update address before memory access
S	shift	!	16 bit instruction	P	ld/st with base pointer
U	unsign	C	carry	.A	update address after memory access
W	write	I	immediate	L	link

8.3.1 32-Bit instruction

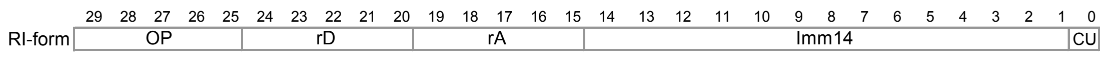
The 5-bit main opcode field, OP, distinguishes the 32-bit instruction into several instruction forms and instructions. Fig 8-15 shows the categorization of instructions that distinguished by OP field.

Opcode

instruction form	data processing instructions	load/store instructions	Control register instructions	Coprocessor instructions	Cache instruction	jump and branch instruction
special-form	ADDRI	LW	CR	CP	CACHE	Jx
I-form-1	ANDRI	LHU				B<cond>x
I-form-2	ORRI	LBU				
RIX-form-1		LB				
RIX-form-2		SW				
CENew		SH				
		SB				
		LH				

Fig 8-15 Categorization of instructions that distinguished by OP field

The data processing instructions, load/store instructions and cache instruction distinguished by OP field are of RI-form.



The operation of RI-form data processing instructions is:

```
rD = rA OP Imm14*
if(CU==1)
    update conditional code
```

* For ADDRI Imm14 is signed, for ANDRI and ORRI Imm14 is unsigned

The CU bit is meaningless to load/store instructions and cache instruction, thus the address calculation of RI-form load/store instructions and cache instruction is:

```
address = rA + SImm15*
```

* SImm15 represents a singed 15-bit immediate

Special-form

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9	8	7	6 5 4 3 2 1	0
Special-form	OP	rD	rA	rB	0	0	0	func6	CU
	0 0 0 0 0								

All Special-form instructions share the same OP encoding. The func6 field distinguishes different Special-form instructions. Fig 8-16 shows the Special-form instructions that are categorized by function.

Special-func

special instructions	A op B instructions	bit opration instruction	multiplication and division	register transfer instructions	conditional instructions	shift and rotate instructions	Extension instructions	load/store combine word
NOP	ADD	BITCLR.c	MUL	MTCE inst.	TRAP<cond>	SLL	EXTSB	LCB
SYSCALL	ADDC	BITSET.c	MULU	MFCE inst.	BR<cond>x	SRL	EXTSH	LCW
SDBBP	SUB	BITTGL.c	DIV	MTSR	MV<cond>	SRA	EXTZB	LCE
ALW	SUBC	BITTST.c	DIVU	MFSR	T<cond>	ROL	EXTZH	SCB
ASW	NEG					ROR		SCW
	AND					ROLC.c		SCE
	OR					RORC.c		
	NOT					SLLI		
	XOR					SRLI		
	CMP<cond>.c					SRAI		
	CMPZ<cond>.c					ROLI		
						RORI		
						ROLIC.c		
						RORIC.c		

Fig 8-16 Special-form instructions categorized by function

Special-form special instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14	13	12	11	10	9	8	7	6 5 4 3 2 1	0
SYSCALL	OP	SWP15							0	0	0	func6	CU
SDBBP	OP	0 0 0 0 0	code5	0	0	0	0	0	0	0	0	func6	CU
ALW	OP	rD	rA	0	0	0	0	0	0	0	0	func6	CU
ASW	OP	rD	rA	0	0	0	0	0	0	0	0	func6	CU
NOP	OP	0 0 0 0 0	0 0 0 0 0	0	0	0	0	0	0	0	0	func6	CU
	0 0 0 0 0												

For SYSCALL and SDBBP instructions, no operands are specified. Instead, programming information may be preserved in the instructions by specifying software parameters or software debug break point code. These two instructions would trigger exception for their corresponding purpose.

There are two instructions, ALW (Atomic Load Word) and ASW (Atomic Store Word), for atomic memory access. These two instructions utilize the content of rA as memory access address.

Special-form A op B instructions

	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
general	OP					rD					rA					rB					0	0	0	func6						CU
neg	OP					rD					0	0	0	0	0	rB					0	0	0	func6						CU
CMP<TC>.c	OP					0	0	0	TC		rA (S1)					rB (S2)					0	0	0	func6						CU
CMPZ<TC>.c	OP					0	0	0	TC		rA (S1)					0	0	0	0	0	0	0	0	func6						CU

0 0 0 0 0

The rA field of NEG instruction is ignored. NEG instruction operations is:

```
rD = 0 - rB
```

For compare instruction, CMP<TC>.c and CMPZ<TC>.c, TC field specifies the test condition that would set T flag. There are three compare instructions corresponding to the 2-bit TC encoding for compare instructions:

TC	<TC>	Instruction	Set T flag Condition
00	<TEQ>	CMPTEQ.c	If Z flag is true, set T flag; else clear T flag
		CMPZTEQ.c	
01	<TMI>	CMPTMI.c	If N flag is true, set T flag; else clear T flag
		CMPZTMI.c	
11	none	CMP.c	Does not chang T flag
		CMPZ.c	

The general operation of compare instructions is:

```
1. Update condition flags, NZCV, according to the result of rA – rB
2. Set/Clear T flag according to the updated condition flags
```

For more information, please refer to section 8.4. Note that for CMPZ<TC>.c instructions, rA would be subtracted by zero to get the corresponding result of compare to zero.

For other A op B instructions of Special-form, their operations are:

```
rD = rA func6 rB
```

Special-form bit operation instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7	6 5 4 3 2 1	0
bittst.c	OP	0 0 0 0 0	rA	BN5	0 0 0	func6	CU
bit operation	OP	rD	rA	BN5	0 0 0	func6	CU
	0 0 0 0 0						

For Special-form bit operation instructions, the rB field is replaced by a 5-bit immediate, BN5. BN5 specifies a certain bit in rA for bit manipulation. The operation for Special-form bit operation instructions is:

```
rD = rA (BN5 specified bit modified)
```

Since BITTST.c instruction only tests the bit specified by BN5 not updating a destination, the rD field is ignored.

Special-form Multiplication and division instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7	6 5 4 3 2 1	0
mul/div	OP	0 0 0 0 0	rA	rB	0 0 0	func6	CU
	0 0 0 0 0						

For multiplication and division instructions of Special-form, since the computation result would be in CEH/CEL registers, only source registers need to be specified. The general operation of Special-form multiplication and division instruction is:

```
CEH | CEL = rA func6 rB
```

Special-form register transfer instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7	6 5 4 3 2 1	0
MFCE/MTCE inst.	OP	rD	rA(optional)	0 0 0 H L	0 0 0	func6	CU
MFSR	OP	rD	0 0 0 0 0	Srn	0 0 0	func6	CU
MTSR	OP	0 0 0 0 0	rA (S1)	Srn	0 0 0	func6	CU
	0 0 0 0 0						

There are two kinds of Special-form register transfer instructions, one is for register transfer between general purpose registers and CEH/CEL registers of custom engine, the other is for register transfer between general purpose registers and special registers.

MFCE and MTCE instructions transfer CEH/CEL registers with general purpose registers according to the H and L bit in the instruction form. H bit specifies the transfer of CEH register while L bit specifies the transfer of CEL register. The following table lists the instruction name, encoding and the corresponding operation of MFCE and MTCE instructions.

H	L	Instruction	Operations
0	0	Reserved	Reserved
0	1	MFCEL	rD = CEL
		MTCEL	CEL = rD
1	0	MFCEH	rD = CEH
		MTCEH	CEH = rD
1	1	MFCEHL	rD = CEH, rA = CEL
		MTCEHL	CEH = rD, CEL = rA

Since MFCE and MTCE instructions may transfer one or two registers at one time, rA is optional for MFCE and MTCE instructions.

For MFSR and MTSR instructions, Srn specifies the special register to be transferred. MFSR needs only to specify destination register rD and special register Srn while MTSR needs only to specify source register rA and special register Srn.

Special-form Conditional execution instructions

	29 28 27 26 25	24	23	22	21	20	19	18	17	16	15	14 13 12 11 10	9	8	7	6 5 4 3 2 1	0
BR<cond>	OP	0	0	0	0	0	rA					BC	0	0	0	func6	LK
TRAP<cond>	OP	0	0	0	0	0	SWP5					EC	0	0	0	func6	CU
MV<cond>	OP	rD					rA					EC	0	0	0	func6	CU
TC<cond>	OP	0	0	0	0	0	0	0	0	0	0	EC	0	0	0	func6	LK
	0 0 0 0 0																

Conditional execution instructions of Special-form feature EC field, for conditional execution, and BC field, for conditional branch register. Conditional execution instructions would be executed if execution condition checks are true while conditional branch register instruction would jump to the target if branch condition checks are true. For more information about EC and BC field, please refer to section 8.4.

Note: Bit14 is don't care for the EC/BC field.

Special-form shift and rotate instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9	8	7	6 5 4 3 2 1	0
shift/rotate	OP	rD	rA	rB	0	0	0	func6	CU
shift/rotate imm	OP	rD	rA	SA5	0	0	0	func6	CU
	0 0 0 0 0								

There are two types of shift and rotate instructions, shift/rotate with register rB and shift/rotate with 5-bit immediate SA5. The general operation of shift/rotate instructions is:

```
rD = rA func6 rB or
rD = rA func6 SA5
```

Note that for rotate with carry instructions, their default CU bit is one for the C flag in Condition register must be updated.

Special-form extension instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14	13	12	11	10	9	8	7	6 5 4 3 2 1	0
ext	OP	rD	rA	0	0	0	0	0	0	0	0	func6	CU
	0 0 0 0 0												

For sign/zero extension instructions, their operations only require one source register. The general operation of Special-form extension instructions is:

```
rD = func6 (rA)
```

Special-form load/store combined word instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14	13	12	11	10	9	8	7	6 5 4 3 2 1	0
LCx, SCx	OP	rD (D)	rA (addr)	0	0	0	0	0	0	0	0	func6	CU
	0 0 0 0 0												

The featured load/store combined word instructions that solve unaligned memory access well are of special form. These load/store instructions utilize rA for memory access address and do combine and rotate operations with LCR/SCR according to the least significant two bits of their memory access address, rA.

The general addressing mode of these instructions is:

```
Load/Store address = rA
rA = rA + 4
```

I-form

	29 28 27 26 25	24 23 22 21 20	19 18 17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	0
I-form-1	OP	rD	func3	Imm16	CU
	0 0 0 0 1				

	29 28 27 26 25	24 23 22 21 20	19 18 17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	0
I-form-2	OP	rD	func3	Imm16	CU
	0 0 1 0 1				

I-form format provide 16-bit immediate operations. For arithmetic instructions, Imm16 is represented in signed number. For logical instructions, Imm16 is represented in unsigned number. There are two groups of I-form instructions, I-form-1 instructions operate with 16-bit immediate while I-form-2 instructions operate with shifted 16-bit immediate. The general operation of I-form-1 instructions is:

```
rD = rD func3 Imm16
```

The general operation of I-form-2 instruction is:

```
rD = rA func3 (Imm16 shift left 16 bit)
```

RIX-form

	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RIX-form-1	OP					rD					rA					Imm12												func3		
	0	0	0	1	1																									

	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RIX-form-2	OP					rD					rA					Imm12												func3		
	0	0	1	1	1																									

Load/store instructions with indexing mode are of RIX-form. RIX-form-1 specifies pre-index addressing mode load/store instructions while RIX-form-2 specifies post-index addressing mode load/store instructions. The address calculation and rA update of RIX-form-1 is:

```
Load/store address = rA + SImm12
rA = rA + SImm12
```

* SImm12 represents a singed 12-bit immediate

The address calculation and rA update of RIX-form-2 is:

```
Load/store address = rA
rA = rA + SImm12
```

* SImm12 represents a singed 12-bit immediate

Control register instructions

	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CR-form	OP					rD					CR					0	0	0	0	0	0	0	CR_OP			func2		II	0	MI
mfcr/mtcr	OP					rD					CR					0	0	0	0	0	0	0	0	0	0	func2		II	0	MI
xtlb, rte/drte	OP					0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	CR_OP			func2		II	0	MI
	0	0	1	1	0																					0	0			

All control register instructions share one main opcode encoding and one certain func2 code. There are two types of control register instructions: control register move instructions and control register control instructions. MI field would be true for control register move instructions while II field true for control register control instructions. The sub-field CR_OP specifies the corresponding control register control instruction operations: RTE, DRTE and SLEEP. All the control register instructions can only be accessed either in kernel mode or in user mode with CRA bit enabled.

Cache instructions

	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
CACHE	OP	Cache_op	rA	Imm15
	1 1 0 0 0			

For cache instruction, a reference address is calculated as:

```
address = rA + SImm15
```

* SImm15 represents a singed 15-bit immediate

Table 8-6 shows the Cache_op and the corresponding cache operation.

Table 8-6 Cache_op and the corresponding cache operation

Cache-OP[4:0]	I-Cache/ D-Cache	Function	Data
0x00	I-Cache	Pre-fetch a Cache-line	VA
0x01	I-Cache	Pre-fetch and lock a Cache-line	VA
0x02	I-Cache	Invalid and unlock a Cache-line	VA
0x03	I-Cache	Fill LIM (local instruction memory) device	VA (PFN & Size)
0x04	I-Cache	Re-Fill LIM (local instruction memory with the PFN and Size of previous value) device	VA
0x08	D-Cache	Pre-fetch a Cache-line	VA
0x09	D-Cache	Pre-fetch and lock a Cache-line	VA
0x0A	D-Cache	Invalid and unlock a Cache-line	VA
0x0B	D-Cache	Fill LDM (local data memory) device	VA (PFN & Size)
0x0C	D-Cache	Write-Back LDM (local data memory) device to main memory	NA
0x0D	D-Cache	Force write-back a Cache-line and set valid when the cache-line is valid and dirty	VA
0x0E	D-Cache	Force write-back a Cache-line and set invalid when the cache-line is valid and dirty	VA
0x10	I-Cache	Invalid entire cache	NA
0x11	I-Cache	Toggle Instruction Pre-fetch Buffer Function (Enable/Disable)	NA
0x18	D-Cache	Invalid entire cache	NA
0x1A	D-Cache	Drain Write Buffer	NA
0x1B	D-Cache	Toggle Write Buffer Function	NA
0x1C	D-Cache	Toggle Data Pre-fetch Buffer Function (Enable/Disable)	NA
0x1D	D-Cache	Toggle Write-back D-Cache Function (Enable/Disable)	NA

Cache-OP[4:0]	I-Cache/ D-Cache	Function	Data
0x1E	D-Cache	Force write-back entire D-Cache and set valid of the cache-lines are valid and dirty. (Write-out)	NA
0x1F	D-Cache	Force write-back entire D-Cache and set invalid of the cache-lines are valid and dirty. (Flush)	NA

Coprocessor instructions

coprocessor	29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3	2 1 0
mtc/mfc	OP	rD	CrA	0 0 0 0 0	0 0 0 0 0	CP#	Sub-OP
ldc/stc	OP	rD	CrA	Imm10		CP#	Sub-OP
cop	OP	CrD	CrA	CrB	COP-Code	CP#	Sub-OP
	0 0 1 1 0						

All coprocessor instructions share one main opcode encoding. There are three types of coprocessor instructions: coprocessor register transfer instructions, coprocessor data transfer instructions and coprocessor operation instructions. The Sub-OP field distinguishes different coprocessor instructions while CP# specifies the coprocessor number. Coprocessor register transfer instructions are MTC# (move to coprocessor) and MFC# (move from coprocessor). Coprocessor data processing instructions are LDC# (coprocessor load) and STC# (coprocessor store). The address calculation of coprocessor data transfer is:

Load/store address = rD + (SImm10 shift left 2 bits)

* SImm10 represents a singed 10-bit immediate

Coprocessor operation instructions are opened to users to define application special functional instructions. COP-Code indicates the functionality of different coprocessor operation instructions. Note that CP# = 2'b00 is reserved, thus three coprocessors in the S+core architecture.

Jump instructions

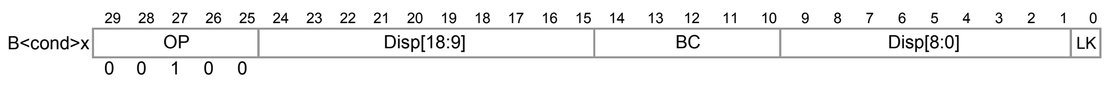
	29 28 27 26 25	24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	0
J-form	OP	Disp24	LK
	0 0 0 1 0		

There are two jump instructions: J(Jump) and JL(Jump and link). The target address calculation is:

Target = {PC[31:25], Disp24 << 1'b0}

* Disp24 represents a singed 24-bit immediate

Conditional branch instructions



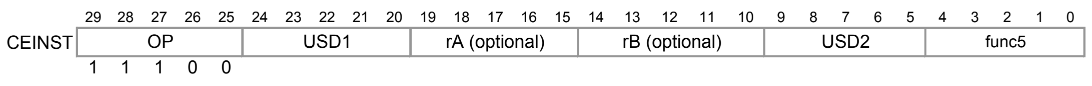
There are two types of conditional branch instructions: conditional branch instructions and conditional branch and link instructions. The target address calculation is:

Target = PC + (Disp19 shift left one bit)

* Disp19 represents a singed 19-bit immediate

The conditional branch instructions would jump to the target if BC (branch condition) check is true. For more information about BC field, please refer to section 8.4.

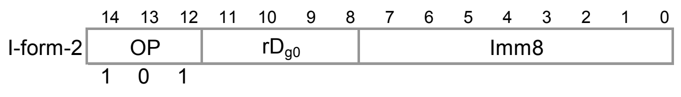
CEISNT form



For custom engine instruction extension, CEINST format is defined. In this format, func5 defines the custom engine operation. This format allows maximum two source registers from general purpose registers. Though rA and rB fields are optional, if one need to specify one or two general purpose source registers, the register indexes should be placed in these two fields. USD1 and USD2 are user defined fields. These two fields could be either immediate for computation, parameters for operation control or destination general purpose register index.

8.3.2 16-Bit instruction

The 3-bit Op field of 16 bit instructions distinguishes different types of 16-bit instructions. The only instruction that specified by OP field is LDIU!. LDIU! is represented as I-form-2.



LDIU! instruction loads a 8-bit unsigned immediate. The operation of this instruction is:

RD_{g0} = Imm8

R-form

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-form-1	OP			rD_{g0}				rA_{g0}				func4			
	0	0	0												

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-form-2	OP			rD_{g0}				rA_{g0}				func4			
	0	1	0												

There are generally two operands in R-form instructions. Due to the 4-bit register index, only the lower 16 registers, R0~R15, can be accessed by 16 bit instructions. Fig 8-17 shows the R-form instructions categorized in functions.

special instructions	A op B instructions	register transfer instructions	conditional instructions	shift and rotate instructions	load/store instructions
NOP!	ADDC!	MLFH!	BR<cond>!	SLL!	LW!
T<cond>!	ADD!	MHFL!	BR<cond>L!	SRL!	LH!
	SUB!	MV!		SRA!	LBU!
	NEG!				SW!
	CMP!				SH!
	AND!				SB!
	OR!				PUSH!
	NOT!				POP!
	XOR!				

Fig 8-17 R-form instructions categorized in function

R-form special instructions

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T<cond>!	OP			EC				0	0	0	0	func4			
	0	0	0												

The T<cond> instructions are used to set/clear T flag according to the result of condition flag test. EC field specifies the corresponding condition flag test. For more information about EC field, please refer to section 8.4.

R-form A op B instructions

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-form-1	OP			rD_{g0}				rA_{g0}				func4			
	0	0	0												

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R-form-2	OP			rD_{g0}				rA_{g0}				func4			
	0	1	0												

The general operation of R-form A op B instruction is:

rD_{g0} = rD_{g0} func4 rA_{g0}

R-form register transfer instructions

	14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
MHFL!	OP	rD_{g1}	rA_{g0}	func4
MLFH!	OP	rD_{g0}	rA_{g1}	func4
	0 0 0			

Due to the limitation on number of registers can be accessed in 16-bit instructions, move from higher register (MLFH!) and move to higher register (MHFL!) are included in 16-bit instruction set. Programmers can access higher registers (R16~R32) via these two instructions.

R-form conditional instructions

	14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
BR<cond>!	OP	BC	rA_{g0}	func4
BR<cond>L!	OP	BC	rA_{g0}	func4
	0 0 0			

There is only one type of R-form conditional instructions, conditional branch register instructions. Conditional branch register instructions feature BC field. Conditional branch register instructions would be executed if execution condition checks are true. For more information about BC field, please refer to section 8.4.

R-form load/store instructions

	14 13 12	11 10 9 8	7	6 5 4	3 2 1 0
push!, pop!	OP	rD	H	rA	func4
	0 1 0				

The address calculation of R-form load/store instructions is:

Load/store address = rA

Note that for push! and pop! instructions, there could be eight possible base registers, r0~r7 while these instructions could access source / destination registers r0~r31. H bit is used to indicate whether rD indicates r16~r31(H bit = 1) or r0~r15(H bit = 0).

The address calculation and update of PUSH! instruction is:

Store address = rA – 4
RA = rA – 4

The address calculation and update of POP! instruction is:

```
Load address = rA
rA = rA + 4
```

I-form-1

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I-form-1a	OP			rD_{g0}				Imm5					func3		
	1	1	0												

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
I-form-1b	OP			rD_{g0}				Imm5					func3		
	1	1	1												

There is generally one operand in I-form instructions. Due to the 4-bit register index, only the lower 16 registers, R0~R15, can be accessed by 16 bit instructions. Fig 8-18 shows the I-form instructions categorized in functions.

R-form

special instructions	A op B instructions	register transfer instructions	conditional instructions	shift and rotate instructions	load/store instructions
NOP!	ADDC!	MLFH!	BR<cond>!	SLL!	LW!
T<cond>!	ADD!	MHFL!	BR<cond>L!	SRLI!	LH!
	SUB!	MV!		SRL!	LBU!
	NEG!			SRA!	SW!
	CMP!				SH!
	AND!				SB!
	OR!				PUSH!
	NOT!				POP!
	XOR!				

Fig 8-18 I-form instructions categorized in function

I-form-1 special instruction

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SDBBP!	OP			0	0	0	0	Code					func3		
	1	1	0												

There is only one special instruction of I-form, SDBBP. The software debug break point code is preserved in the Code field.

I-form-1 A op B instruction

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
bit operation	OP			rD_{g0}				BN5					func3		
addei!	OP			rD_{g0}				S	Exp4				func3		
	1	1	0												

There are two types of I-form A op B instructions, bit operation instructions and add exponent instructions. For bit operation instructions, BN5 specifies a certain bit in rD_{g0} for bit manipulation. The operation for I-form bit operation instructions is:

$$rD_{g0} = rD_{g0}\ (\text{BN5 specified bit modified})$$

For add exponent instructions, Exp4 represents a positive exponent of the radix-2 addend. S bit indicates that whether the instruction operation is an addition or a subtraction. The general operation of an add exponent instruction is:

```
If(S=0)
    $rD_{g0} = rD_{g0} + 2^{Exp4}$
else
    $rD_{g0} = rD_{g0} - 2^{Exp4}$
```

I-form-1 shift instruction

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SLLI! / SRLI!	OP			rD_{g0}				SA5					func3		
	1	1	0												

There is only two shift instruction in I-form, SLLI!, shift left with 5-bit immediate SA5; and SRLI!, shift right with 5-bit immediate SA5. The general operation of shift/rotate instructions is:

$$rD_{g0} = rD_{g0}\ \text{func3 SA5}$$

I-form-1 load/store instruction

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ld/st with BP	OP			rD_{g0}				Imm5					func3		
	1	1	1												

I-form load/store instructions calculate their address with base pointer register. The address calculation of I-form load/store instructions is:

Load/store address = base pointer + (Imm5 shift left *L* bits)
For word access: *L* = 2
For halfword access: *L* = 1
For byte access: *L* = 0

* Imm5 represents an unsigned 5-bit immediate

J-form instructions

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
J-form	OP			Disp11(Imm)											LK
	0	1	1												

There are two jump instructions: J!(Jump) and JL!(Jump and link). The target address calculation is:

Target = {PC[31:12], Disp11 << 1'b0}

* Disp11 represents a singed 11-bit immediate

Conditional branch instructions

	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B<cond>x	OP			BC				Disp8(Imm)							
	1	0	0												

There are two types of conditional branch instructions: conditional branch instructions and conditional branch and link instructions. The target address calculation is:

Target = PC + (Disp8 shift left one bit)

* Disp8 represents a singed 8-bit immediate

The conditional branch instructions would jump to the target if BC (branch condition) check is true. For more information about BC field, please refer to section 8.4.

8.3.3 Instruction Set Summary

Load/Store Instructions

load word	load halfword	load byte	store word	store halfword	store byte
LW	LH	LB	SW	SH	SB
LW.B	LH.B	LB.B	SW.B	SH.B	SB.B
LW.A	LH.A	LB.A	SW.A	SH.A	SB.A
	LHU	LBU			
	LHU.B	LBU.B			
	LHU.A	LBU.A			
LCB			SCB		
LCW			SCW		
LCE			SCE		
ALW			ASW		
LW!	LH!	LBU!	SW!	SH!	SB!
LWP!	LHP!	LBUP!	SWP!	SHP!	SBP!
POP!			PUSH!		

Fig 8-19 S$^+$core load/store instructions

Data Processing Instructions - Arithmetic

add	cmp	neg	sub
ADD	CMP<cond>.c	NEG	SUB
ADDC	CMPI.c		SUBC
ADDI	CMPZ<cond>.c		
ADDIS			
ADDRI			
ADD!	CMP!	NEG!	SUB!
ADDC!			
ADDEI!			SUBEI!

Fig 8-20 S$^+$core data processing instructions - arithmetic

Data Processing Instructions - Logical

and	bit operation	li	not	nop	or	xor
AND	BITCLR.c	LDI	NOT	NOP	OR	XOR
ANDI	BITSET.c	LDIS			ORI	
ANDIS	BITTGL.c				ORIS	
ANDRI	BITTST.c				ORRI	
AND!	BITCLR!	LDIU!	NOT!	NOP!	OR!	XOR!
	BITSET!					
	BITTGL!					
	BITTST!					

Fig 8-21 S^+core data processing instruction – logical

Data Processing Instructions - miscs

shift	rotate	rotate with carry	extension	register transfer
SLL	ROL	ROLC.c	EXTSB	MFSR
SLLI	ROLI	ROLIC.c	EXTSH	MTSR
SRA	ROR	RORC.c	EXTZB	MV<cond>
SRAI	RORI	RORIC.c	EXTZH	MFCR
SRL				MTCR
SRLI				
SLL!				MV!
SLLI!				MLFH!
SRLI!				MHFL!
SRA!				
SRL!				

Fig 8-22 S^+core data processing instructions - miscs

Control Instructions

jump and branch	system control	cache	debug	tlb
B<cond>	NOP	CACHE	SDBBP	STLB
B<cond>L	RTE			MFTLB
BR<cond>	SYSCALL			MTPTLB
BR<cond>L	T<cond>			MTRTLB
J				
JL				
B<cond>!	T<cond>!		SDBBP!	
BR<cond>!				
BR<cond>L!				
J!				
JL!				

Fig 8-23 S^+core control instructions

Extended Instructions

coprocessor	custom engine
mfc#	MUL
mtc#	MULU
ldc#	DIV
stc#	DIVU
cop#	MFCE
	MTCE
	CENew

Fig 8-24 S^+core extended instructions

8.4 Condition Flags

As introduced before, conditional instructions would be executed according to the result of execution condition (EC) check while conditional branch instructions branches according to the result of branch condition (BC) check. Fig 8-25 shows the branch condition (BC) and the corresponding condition flag check. Note that CNZ decrement only valid for branch instructions. Since counter register is used for loop control, only conditional branch instructions utilize BC of CNZ (counter register non-zero) and decrease CNT by one for each conditional branch on counter register non-zero instruction.

	BC				operation	cf test	Suffix
0	0	0	0	0	branch on carry set (>=unsigned)	C	CS(GEU)
1	0	0	0	1	branch on carry clear (<unsigned)	~C	CC(LTU)
2	0	0	1	0	branch on (>unsigned)	C & ~Z	GTU
3	0	0	1	1	branch on (<=unsigned)	~C | Z	LEU
4	0	1	0	0	branch on (=)	Z	EQ
5	0	1	0	1	branch on (!=)	~Z	NE
6	0	1	1	0	branch on (>signed)	(Z = 0) & (N = V)	GT
7	0	1	1	1	branch on (<=signed)	(Z = 1) | (N != V)	LE
8	1	0	0	0	branch on (>=signed)	N = V	GE
9	1	0	0	1	branch on (<signed)	N != V	LT
10	1	0	1	0	branch on -	N	MI
11	1	0	1	1	branch on +/0	~N	PL
12	1	1	0	0	branch overflow	V	VS
13	1	1	0	1	branch no overflow	~V	VC
14	1	1	1	0	branch on (CNT>0), CNT--	CNT>0	CNZ
15	1	1	1	1	branch always	-	AL

* CNZ condition only for branch instructions

Fig 8-25 branch condition and condition flag check

CNT	= counter register
BC	= branchcondition field in conditional instructions
cf test	= condtion flag test according to the EC
suffix	= the corresponding suffix for conditional instructions

Fig 8-26 shows the execution condition and condition flag check for conditional execution instructions. For conditional execution instructions, conditional move and conditional trap, EC field of encoding 1110 (CNZ) is reserved. In other words, there are no MVCNZ and TRAPCNZ instructions. Note that for CNZ (counter none zero) condition, instead of checking condition flags N, Z, C, V, both BC and EC checks the value of counter register.

	EC				operation	cf test	Suffix
0	0	0	0	0	execute on carry set (>=unsigned)	C	CS(GEU)
1	0	0	0	1	execute on carry clear (<unsigned)	~C	CC(LTU)
2	0	0	1	0	execute on (>unsigned)	C & ~Z	GTU
3	0	0	1	1	execute on (<=unsigned)	~C | Z	LEU
4	0	1	0	0	execute on (=)	Z	EQ
5	0	1	0	1	execute on (!=)	~Z	NE
6	0	1	1	0	execute on (>signed)	(Z = 0) & (N = V)	GT
7	0	1	1	1	execute on (<=signed)	(Z = 1) | (N != V)	LE
8	1	0	0	0	execute on (>=signed)	N = V	GE
9	1	0	0	1	execute on (<signed)	N != V	LT
10	1	0	1	0	execute on -	N	MI
11	1	0	1	1	execute on +/0	~N	PL
12	1	1	0	0	execute on overflow	V	VS
13	1	1	0	1	execute on no overflow	~V	VC
14	1	1	1	0	execute on (CNT>0)	CNT>0	CNZ
15	1	1	1	1	execute always	-	AL

*cf test for CNZ does not check condition flag

Fig 8-26 execution condition and condition flag check

CNT	=	counter register
EC	=	execution condition field in conditional instructions
cf test	=	condtion flag test according to the EC
suffix	=	the corresponding suffix for conditional instructions

Fig 8-27 shows the instructions that update condition flags.

bit operation

	N	Z	C	V
BITCLR.c	v	v		
BITSET.c	v	v		
BITTGL.c	v	v		
BITTST.c	v	v		
BITCLR!	v	v		
BITSET!	v	v		
BITTGL!	v	v		
BITTST!	v	v		

logical

	N	Z	C	V
AND.c	v	v		
ANDI.c	v	v		
ANDIS.c	v	v		
ANDRI.c	v	v		
OR.c	v	v		
ORI.c	v	v		
ORIS.c	v	v		
ORRI.c	v	v		
XOR.c	v	v		
NOT.c	v	v		
AND!	v	v		
OR!	v	v		
XOR!	v	v		
NOT!	v	v		

shift

	N	Z	C	V
SRA.c	v	v	v	
SRAI.c	v	v	v	
SRL.c	v	v	v	
SRLI.c	v	v	v	
SLL.c	v	v	v	
SLLI.c	v	v	v	
SRA!	v	v	v	
SRL!	v	v	v	
SLL!	v	v	v	
SLLI!	v	v	v	
SRLI!	v	v	v	

rotate

	N	Z	C	V
ROL.c	v		v	
ROLI.c	v		v	
ROLC.c	v		v	
ROLIC.c	v		v	
ROR.c	v		v	
RORI.c	v		v	
RORC.c	v		v	
RORIC.c	v		v	

extension

	N	Z	C	V
EXTSB.c	v			
EXTSH.c	v			
EXTZB.c	v			
EXTZH.c	v			

arithmetic

	N	Z	C	V
ADD.c	v	v	v	v
ADDC.c	v	v	v	v
ADDI.c	v	v	v	v
ADDIS.c	v	v	v	v
ADDRI.c	v	v	v	v
SUB.c	v	v	v	v
SUBC.c	v	v	v	v
CMP<TC>.c	v	v	v	v
CMPZ<TC>.c	v	v	v	v
CMPI.c	v	v	v	v
NEG.c	v	v	v	v
ADD!	v	v	v	v
ADDC!	v	v	v	v
ADDEI!	v	v	v	v
SUB!	v	v	v	v
SUBEI!	v	v	v	v
CMP!	v	v	v	v
NEG!	v	v	v	v

Fig 8-27 Instructions that update condition flags

For bit operation instructions, the corresponding condition flags updating is:

```
// N flag
N = rD[31];
// Z flag would true if the calculated result is zero, else Z flag would be false
Z = (rD==0)? 1:0;
// C flag stores the BN specified bit before it is modified
If 32-bit
   C = rA[BN];
If 16-bit
   C = rDorg[BN];
```

For logical instructions, the corresponding condition flags updating is:

```
// N flag
N = rD[31];
// Z flag would true if the calculated result is zero, else Z flag would be false
Z = (rD==0)? 1:0;
```

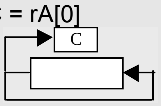
For shift instructions, the corresponding condition flags updating is:

```
// N flag
N = rD[31];
// Z flag would true if the calculated result is zero, else Z flag would be false
Z = (rD==0)? 1:0;
// C flag
C = the last shift out bit;
```

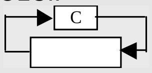
For rotate instructions, the corresponding condition flags updating is:

```
// N flag
N = rD[31];
```

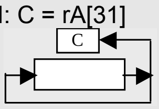
ROL/ROLI: C = rA[0]



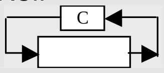
ROLC/ROLCI:



ROR/RORI: C = rA[31]



RORC/RORCI:



The C flag would be the bit that rotate in C flag register in the rotate operation.

For extension instructions, the corresponding condition flags updating is:

```
// N flag would be updated with the MSB of rD
N = rD[31]
```

For arithmetic instructions, the corresponding condition flags updating is:

```
// N flag would be updated with the MSB of rD
N = rD[31]
// Z flag would true if the calculated result is zero, else Z flag would be false
Z = (rD==0)?1:0
// C flag would be updated with the carry bit generated by A func B
C = carry of (A func B)
// V flag would be true if A func B operation overflows.
V = overflow of (A func B)
```

8.5 Addressing Modes

Table 8-7 shows the addressing modes of load/store instructions. Effective address represents the actual address that used to load/store a data. rA update row shows the value of rA after the load/store instruction.

Table 8-7 Load/store instructions of different addressing modes

load byte	LB	LB.B	LB.A									
load byte unsigned	LBU	LBU.B	LBU.A	LBU!	LBUP!							
load halfword	LH	LH.B	LH.A									
load halfword unsigned	LHU	LHU.B	LHU.A	LH!	LHP!							
load word	LW	LW.B	LW.A	LW!	LWP!	POP!	LCB	LCW	LCE		ALW	LDC
store byte	SB	SB.B	SB.A	SB!	SBP!							
store halfword	SH	SH.B	SH.A	SH!	SHP!							
store word	SW	SW.B	SW.A	SW!	SWP!		SCB	SCW	SCE	PUSH!	ASW	STC
Effective address	rA + SImm15	rA + SImm12	rA	rA	BP + Imm5*	rA	rA	rA	rA	rA-4	rA	rA + SImm10**
rA update	rA	rA + SImm12	rA + SImm12	rA	rA	rA + 4	rA + 4	rA + 4	rA + 4	rA-4	rA	rA

* Imm5 shift left 2 bits for word access, 1 bit for halfword access

** SImm10 shift left 2 bit for word access

Fig 8-28 shows the target address calculation of each jump and branch instruction.

all Disp are represented in signed immediate

Effective address	{PC[31:25], Disp24, 1'b0}	PC + (Disp19<<1)	rA	{PC[31:12], Disp11, 1'b0}	PC + (Disp8<<1)	rA
	J	B<cond>	BR<cond>	J!	B<cond>!	BR<cond>!
	JL	B<cond>L	BR<cond>L	JL!		BR<cond>L!

Fig 8-28 Target address calculation of different branch and jump instructions

9 32-Bit Instructions

This chapter provides an overview of the S$^+$core 32-bit instructions. The mnemonic, operation and action for each instruction will be listed in the tables of the following subsections. The mnemonic and action performed by each instruction using a high-level language notation. Special symbols used in the notation are described in Table **9-1**.

Table 9-1 CPU Instruction Action Notations

Symbol	Meaning	Example
Imm_n	n-bit unsigned immediate	
$SImm_n$	n-bit signed immediate	
Mem_n[address]	n-bit content of the memory that specified by address	
k'bX	k-bit binary representation of X value	2'b11 = 3
k'hX	k-bit hexadecimal representation of X value	8'h2a = 42
X[n]	selection of bit n of bit string X	X=2'b10, X[1]=2'b1
X[n:m]	selection of bits n through m of bit string X	X = 6'b111001, X[3:0] = 4'b1001
k{A}	replication of bit value A into a k-bit string	K = 0, 2{0} = 2'b00
k{X[n]}	replication of bit value X[n] into a k-bit string	X = 2'b01, 2{X[1]} = 2'b00
k{X[n:m]}	replication of bit value X[n:m] into a (k*(n-m+1))-bit string	X = 2'b01, 2{X[1:0]} = 4'b0101
{X, Y}	concatenation of X,Y	X = 2'b11, Y=2'b00, {X, Y} = 4'b1100
SE(X)	sign extend X to a 32-bit string	X=8'hfe, SE(X) = 32'hfffe
ZE(X)	zero extend X to a 32-bit string	X=8'hfe, ZE(X) = 32'h00fe
+	add	
-	subtract	
*	multiplication	
/	division	
Q(X/Y)	Quotient of X divided by Y	Q(5/2) = 2
R(X/Y)	Remainder of X divided by Y	R(5/2) = 1
&	bitwise logic and	X=2'b11, Y=2'b00, X&Y = 2'b00
|	bitwise logic or	X=2'b11, Y=2'b00, X|Y =2'b11
~	bitwise logic not	X=2'b11, Y = ~X, Y=2'b00
^	bitwise logic xor	X=2'b11, Y=2'b01, X^Y = 2'b10
<<	shift left	
ROR	rotate right	
ROL	rotate left	
X^Y	this represents X to the power of Y	10^3 = 1000
cond	cond is true if the icc test according to EC/BC field in the instruction is true	BEQ target. BC = 4'b0000, the corresponding icc test is Z flag test. In this case, if Z flag is true, cond is true; otherwise, cond is false.

Symbol	Meaning	Example
<cond>	suffix of conditional execution/branch instructions which specifies the execution/branch condition(EC/BC)	
LR	link register, generally is R3	
BP	base pointer register	
GPR[Rn]	this indicates that Rn is a general purpose register	
COPz(Rn)	this indicates that Rn which belongs to COPz	
Bn	Bit Number	
T	T flag	
C	carry flag	
Z	zero flag	
Srn	Special registers	Sr0 is CNT register Sr1 is LCR register Sr2 is SCR register
byte	byte number	
CEop<n>	Custom Engine instruction operation	

S^{+}core 32-bit instructions can be divided into the following functional categories:

- Load and store instructions
- Data processing instructions
- Custom engine instructions
- Jump and branch instructions
- Special instructions
- Coprocessor instructions

9.1 Load/Store Instructions

The load/store instructions transfer data between register and memory.

Table **9-2** lists the supported data types of load/store instructions.

Table 9-2 Data Types for Load and Store instructions

Data Type		**Supported Operation**	**Note**
Byte	Signed	Load	
	Unsigned	Load, Store	
Halfword	Signed	Load	Must be aligned to halfword boundary
	Unsigned	Load, Store	
Word		Load, Store	Must be aligned to word boundary

The addressing mode of S$^+$core is formed from base register and offset, which are used in three different ways to form the effective address:

- ***Offset:*** The base register and offset are added or subtracted to form the effective address.
- ***Pre-indexed:*** The base register and offset are added or subtracted to form the effective address. The base register is then updated with this new address, to allow automatic indexing through an array or memory block.
- ***Post-indexed:*** The effective address is formed by the base register. If the memory access instruction contains offset, the base register and offset are added or subtracted to update the base register, to allow automatic indexing through an array or memory block. If the memory access instruction does not contain offset, the base register would be updated with base register increment by 4.
- ***Register:*** The effective address is the content of the general purpose register.

Table 9-3 shows the register allocation and their corresponding load/store address in different Endian system. In this table, memory words are represented in groups of 4 blocks. Each block represents a memory byte. Numbers marked on the blocks indicate the addresses of the bytes. The highlighted blocks are accessed memory bytes of different granularity. The following descriptions give examples for different access orders.

1. For load word instructions in Big Endian systems, byte of address 3 is placed in the least significant 8 bits of the destination register while byte of address 0 is placed in the most significant 8 bits of the destination register.
2. For store word instructions in Little Endian systems, the least significant 8 bits of the source register is placed in byte address 3 while the most significant 8 bits of the source register is placed in byte address 0.
3. For load word instructions in Little Endian systems, byte of address 3 is placed in the most significant 8 bits of the destination register while byte of address 0 is placed in the least significant 8 bits of the destination register.

4. For store word instructions in Little Endian systems, the least significant 8 bits of the source register is placed in byte address 0 while the most significant 8 bits of the source register is placed in byte address 3.
5. For load half word instructions with address LSB 00 in Big Endian systems, byte of address 1 is placed in the least significant 8 bits of the destination register while byte of address 0 is placed in the 8 bits next to the least significant 8 bits of the destination register.
6. For load half word instructions with address LSB 10 in Big Endian systems, byte of address 3 is placed in the least significant 8 bits of the destination register while byte of address 2 is placed in the 8 bits next to the least significant 8 bits of the destination register.
7. …

Table 9-3 Register allocation vs. memory address

Access Type	**Address LSB 2 bits**	**Register allocation vs. memory address**	
		Big Endian **31 __________ 0**	**Little Endian** **31 __________ 0**
Word	00	0 1 2 3	3 2 1 0
Halfword	00	0 1 ▨ ▨	▨ ▨ 1 0
	10	▨ ▨ 2 3	3 2 ▨ ▨
byte	00	0 ▨ ▨ ▨	▨ ▨ ▨ 0
	01	▨ 1 ▨ ▨	▨ ▨ 1 ▨
	10	▨ ▨ 2 ▨	▨ 2 ▨ ▨
	11	▨ ▨ ▨ 3	3 ▨ ▨ ▨

9.1.1 Load Instructions

The load instructions of S$^+$core 32-bit mode are listed in Table 9-4.

Table 9-4 Load Instructions

Mnemonic	**Operation**	**Action**
LB rD, [rA, $SImm_{15}$]	Load Byte Signed	rD = SE(Mem_8[rA+SE($SImm_{15}$)])
LBU rD, [rA, $SImm_{15}$]	Load Byte Unsigned	rD = ZE(Mem_8[rA+SE($SImm_{15}$)])
LB.B rD, [rA, $SImm_{12}$]+	Load Byte Singed (Pre-index)	rD = SE(Mem_8[rA + SE($SImm_{12}$)]), rA = rA + SE($SImm_{12}$)
LBU.B rD, [rA, $SImm_{12}$]+	Load Byte Unsigned (Pre-index)	rD = ZE(Mem_8[rA + SE($SImm_{12}$)]), rA = rA + SE($SImm_{12}$)
LB.A rD, [rA]+, $SImm_{12}$	Load Byte Singed (Post-index)	rD = SE(Mem_8[rA]), rA = rA + SE($Simm_{12}$)
LBU.A rD, [rA]+, $Simm_{12}$	Load Byte Unsigned (Post-index)	rD = ZE(Mem_8[rA]), rA = rA + SE($Simm_{12}$)

Mnemonic	Operation	Action
LH rD, [rA, $SImm_{15}$]	Load Halfword Signed	rD = SE(Mem_{16}[rA+SE($SImm_{15}$)])
LHU rD, [rA, $SImm_{15}$]	Load Halfword Unsigned	rD = ZE(Mem_{16}[rA+SE($SImm_{15}$)])
LH.B rD, [rA, $SImm_{12}$]+	Load Halfword Singed (Pre-index)	rD = SE(Mem_{16}[rA + SE($SImm_{12}$)]), rA = rA + SE($SImm_{12}$)
LHU.B rD, [rA, $SImm_{12}$]+	Load Halfword Unsigned (Pre-index)	rD = ZE(Mem_{16}[rA + SE($SImm_{12}$)]), rA = rA + SE($SImm_{12}$)
LH.A rD, [rA]+, $SImm_{12}$	Load Halfword Singed (Post-index)	rD = SE(Mem_{16}[rA]), rA = rA + SE($SImm_{12}$)
LHU.A rD, [rA]+, $SImm_{12}$	Load Halfword Unsigned (Post-index)	rD = ZE(Mem_{16}[rA]), rA = rA + SE($SImm_{12}$)
LW rD, [rA, $SImm_{15}$]	Load Word	rD = (Mem_{32}[rA+SE($SImm_{15}$)])
LW.B rD, [rA, $SImm_{12}$]+	Load Word (Pre-index)	rD = (Mem_{32}[rA+SE($Simm_{12}$)]), rA = rA + SE($SImm_{12}$)
LW.A rD, [rA]+, $SImm_{12}$	Load Word(Post-index)	rD = (Mem_{32}[rA]), rA = rA + SE($SImm_{12}$)
LCB [rA]+	Load Combine Word Begin	LCR = Mem_{32}[{rA[31:2], 2'b0}], rA = rA + 4
LCW rD, [rA]+	Load Combine Word	byte = rA[1:0] xor LittleEndian, rD = {LCR[31-8*byte:0], Mem_{32}[{rA[31:2], 2'b0}][31:31-8*byte]}, LCR = Mem_{32}[{rA[31:2], 2'b0}], rA = rA+4
LCE rD, [rA]+	Load Combine Word End	byte = rA[1:0] xor LittleEndian, if(rA[1:0]==0) rD=LCR, rA=rA+4; else rD = {LCR[31-8*byte:0], Mem_{32}[{rA[31:2], 2'b0}][31:31-8*byte]}, LCR = Mem_{32}[{rA[31:2], 2'b0}], rA = rA+4
ALW rD, [rA]	Atomic Load Word	rD = Mem_{32}[rA] Atbit = 1

9.1.2 Store Instructions

The store instructions of S^{+}core 32-bit mode are listed in Table 9-5.

Table 9-5 Store Instructions

Mnemonic	Operation	Action
SB rD, [rA, $SImm_{15}$]	Store Byte	Mem_8[rA+SE($SImm_{15}$)] = rD[7:0]
SB.B rD, [rA, $SImm_{12}$]+	Store Byte (Pre-index)	Mem_8[rA +SE($SImm_{12}$)] = rD[7:0], rA = rA + SE($SImm_{12}$)
SB.A rD, [rA]+, $SImm_{12}$	Store Byte (Post-index)	Mem_8[rA] = rD[7:0], rA = rA + SE($SImm_{12}$)
SH rD, [rA, $SImm_{15}$]	Store Halfword	Mem_{16}[rA+SE($SImm_{15}$)] = rD[15:0]
SH.B rD, [rA, $SImm_{12}$]+	Store Halfword (Pre-index)	Mem_{16}[rA +SE($SImm_{12}$)] = rD[15:0], rA = rA + SE($SImm_{12}$)
SH.A rD, [rA]+, $SImm_{12}$	Store Halfword (Post-index)	Mem_{16}[rA] = rD[15:0], rA = rA + SE($SImm_{12}$)
SW rD, [rA, $SImm_{15}$]	Store Word	Mem_{32}[rA+SE($SImm_{15}$)] = rD
SW.B rD, [rA, $SImm_{12}$]+	Store Word (Pre-index)	Mem_{32}[rA +SE($SImm_{12}$)] = rD, rA = rA + SE($SImm_{12}$)
SW.A rD, [rA]+, $SImm_{12}$	Store Word (Post-index)	Mem_{32}[rA] = rD, rA = rA + SE($SImm_{12}$)

SCB rD, [rA]+	Store Combine Word Begin	byte = rA[1:0] xor LittleEndian,
		SCR = {rD[8*byte:0], rD[31:31-8*byte]},
		$Mem_{32-8*byte}$[rA[31:0]] =rD[31:8*byte],
		rA = rA + 4

SCW rD, [rA]+	Store Combine Word	byte = rA[1:0] xor LittleEndian,
		SCR = {rD[8*byte : 0],
		rD[31: 1-8*byte]},
		Mem_{32}[{rA[31:2], 2'b0}] = { SCR[31 :
		31-8*byte], rD[31 : 8*byte] },
		rA = rA + 4

SCE [rA]+	Store Combine Word End	byte = rA[1:0] xor LittleEndian,
		if(rA[1:0]==0)
		  rA=rA+4;
		else
		  Mem_{8*byte}[{rA[31:2], 2'b0}] =
		  SCR[31 : 31-8*byte],
		  rA = rA + 4

ASW rD, [rA]	Atomic Store Word	if(Atbit)
		  Mem_{32}[rA] = rD
		rD = {31'b0, AtBit}

9.2 Data Processing Instructions

There are five groups of data processing instructions:

- Arithmetic instructions
- Logical instructions
- Shift/rotate instructions
- Extension instructions
- Move instructions

9.2.1 Arithmetic Instructions

The arithmetic instructions are summarized in

Table **9-6**. This group contains the instructions which perform add or subtract operation. The sign immediate value of this group will be sign-extended to a 32-bit value. The flags will be updated by adding ".c" to Mnemonic. For example, "ADD.c rD, rA, rB" instruction will update flags while "ADD rD, rA, rB" won't update flags. All instructions of this group can add ".c" to update the flags except compare instructions are forced to update flags.

Table 9-6 Arithmetic Instructions

Mnemonic	Operation	Action
ADD rD, rA, rB	Add	rD = rA + rB
ADDC rD, rA, rB	Add With Carry	rD = rA + rB + C
ADDI rD, $SImm_{16}$	Add Immediate	rD = rD + SE($SImm_{16}$)
ADDIS rD, $SImm_{16}$	Add Immediate Shifted	rD = rD + {$SImm_{16}$, 16{0}}
ADDRI rD, rA, $SImm_{14}$	Add Immediate	rD = rA + SE($SImm_{14}$)
SUB rD, rA, rB	Subtract	rD = rA - rB
SUBC rD, rA, rB	Subtract With Carry	rD = rA - rB - (~C)
CMP<cond>.c rA, rB	Compare	rA - rB, if(cond) set T, else clear T
CMPTEQ.c	Compare, if(Z flag) set T flag, else clear T flag	
CMPTMI.c	Compare, if(N flag) set T flag, else clear T flag	
CMP.c	Compare	
CMZ<cond>.c rA	Compare to Zero	rA - 0, if(cond) set T, else clear T
CMPZTEQ.c	Compare to zero, if(Z flag) set T flag, else clear T flag	
CMPZTMI.c	Compare to zero, if(N flag) set T flag, else clear T flag	
CMPZ.c	Compare to zero	
CMPI.c rD, $SImm_{16}$	Compare Immediate	rD - SE($SImm_{16}$)
NEG rD, rB	Negative	rD = 0-rB

9.2.2 Logical Instructions

The logical instructions are summarized in Table 9-7. This group contains the logical operation instructions. The sign immediate value will be sign-extended to a 32-bit value and the unsigned immediate value will be zero-extended to a 32-bit value. The flags will be updated by adding ".c" to Mnemonic. For example, "AND.c rD, rA, rB" instruction will update flags but "AND rD, rA, rB" instruction won't update flags.

Table 9-7 Logical Instructions

Mnemonic	Operation	Action
AND rD, rA, rB	Logical And	rD = rA & rB
ANDI rD, Imm_{16}	Logical And Immediate	rD = rD & ZE(Imm_{16})
ANDIS rD, Imm_{16}	Logical And Immediate Shifted	rD = rD & {Imm_{16}, 16{0}}
ANDRI rD, rA, Imm_{14}	Logical And Immediate	rD = rA & ZE(Imm_{14})

BITCLR.c rD, rA, BN	Clear Bit In Register	rD = rA,
		rD[BN] = 0

BITSET.c rD, rA, BN	Set Bit In Register	rD = rA,
		rD[BN] = 1

BITTGL.c rD, rA, BN	Toggle Bit In Register	rD = rA,
		rD[BN] = ~rA[BN]

BITTST.c rA, BN	Test Bit In Register	Z flag = ~rA[BN]
LDI rD, $SImm_{16}$	Load Immediate	rD = SE($SImm_{16}$)
LDIS rD, Imm_{16}	Load Upper Immediate	rD = {Imm_{16}, 16{0}}
NOP	No Operation	
NOT rD, rA	Logical Not	rD = ~rA
OR rD, rA, rB	Logical Or	rD = rA | rB
ORI rD, Imm_{16}	Logical Or Immediate	rD = rD | ZE(Imm_{16})
ORIS rD, Imm_{16}	Logical Or Immediate Shifted	rD = rD | {Imm_{16}, 16{0}}
ORRI rD, rA, Imm_{14}	Logical Or Immediate	rD = rA | ZE(Imm_{14})
XOR rD, rA, rB	Logical Xor	rD = rA ^ rB

9.2.3 Shift/Rotate Instructions

Table 9-8 lists the shift and rotate instructions. The shift amount (SA) field is a 5-bit immediate which are embedded in the instruction word. The shift/rotate operations by register rB only use the low order 5 bits of register rB to specify the number of bits to shift/rotate. The flags will be updated by adding ".c" to Mnemonic. For example, "SRA.c rD, rA, rB" instruction will update flags but "SRA rD, rA, rB" won't update flags.

Table 9-8 Shift/Rotate Instructions

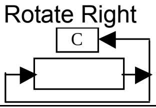
Mnemonic	Operation	Action

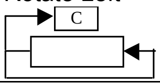
SRA rD, rA, rB	Shift Right Arithmetic	s=rB[4:0],
		rD = {s{rA[31]}, rA[31:s]}

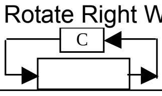
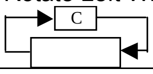
SRL rD, rA, rB	Shift Right Logical	s=rB[4:0],
		rD = {s{0}, rA[31:s]}

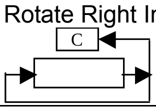
SLL rD, rA, rB	Shift Left Logical	s=rB[4:0],
		rD = {rA[(31-s):0], s{0}}

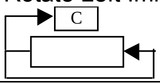
SRAI rD, rA, SA	Shift Right Arithmetic Immediate	rD = {SA{rA[31]}, rA[31:SA]}
SRLI rD, rA, SA	Shift Right Logical Immediate	rD = {SA{0}, rA[31:SA]}
SLLI rD, rA, SA	Shift Left Logical Immediate	rD = {rA[(31-SA):0], SA{0}}

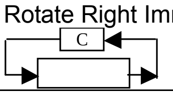
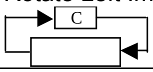
ROR rD, rA, rB	Rotate Right 	rA = rA ROR rB[4:0],
		C = rA[31]

ROL rD, rA, rB	Rotate Left 	rA = rA ROL rB[4:0],
		C = rA[0],

RORC.c rD, rA, rB	Rotate Right With Carry 	{C, rA} = {C, rA} ROR rB[4:0]
ROLC.c rD, rA, rB	Rotate Left With Carry 	{C, rA} = {C, rA} ROL rB[4:0]

RORI rD, rA, SA	Rotate Right Immediate 	C = rA[0],
		rA = rA ROR SA

ROLI rD, rA, SA	Rotate Left Immediate 	C = rA[31],
		rA = rA ROL SA

RORIC.c rD, rA, SA	Rotate Right Immediate With Carry 	{C, rA} = {C, rA} ROR SA
ROLIC.c rD, rA, SA	Rotate Left Immediate With Carry 	{C, rA} = {C, rA} ROL SA

9.2.4 Extension Instructions

The extension instructions of S^{+}core 32-bit mode are listed in

Table **9-9**. The extension operation can be performed as signed (sign-extend) or unsigned (zero-extend). The supported data types are byte and halfword.

Table 9-9 Extension Instructions

Mnemonic	Operation	Action
EXTSB rD, rA	Sign-extend Byte	rD = {24{rA[7]}, rA[7:0]}
EXTSH rD, rA	Sign-extend Halfword:	rD = {16{rA[15]}, rA[15:0]}
EXTZB rD, rA	Zero-extend Byte	rD = {24{0}, rA[7:0]}
EXTZH rD, rA	Zero-extend Halfword	rD = {16{0}, rA[15:0]}

9.2.5 Conditional Move Instructions

Table **9-10** lists the conditional move instructions, which support conditional data transfer between two general purpose registers or one special purpose register and one general purpose register.

Table 9-10 Conditional Move Instructions

Mnemonic	Operation	Action
MV<cond> rD, rA	Move On Condition	If(cond) rD=rA
MVCS/MVGEU	move on <Carry set/unsigned greater than or equal>	
MVCC/MVLSU	move on <carry clear/unsigned less>	
MVGTU	move on <unsigned greater than>	
MVLEU	move on <unsigned less than or equal>	
MVEQ	move on <equal>	
MVNE	move on <not equal>	
MVGT	move on <signed greater than>	
MVLE	move on <signed less than or equal>	
MVGE	move on <signed greater than or equal>	
MVLT	move on <signed less than>	
MVMI	move on <minus/negative>	
MVPL	move on <plus/positive or zero>	
MVVS	move on <overflow>	
MVVC	move on <no overflow>	
MV	move	
MFSR rD, Srn	Move From Special Purpose Register	rD = Srn
MFSR rD, Sr0 MFSR rD, CNT	Move From Counter Register	rD = CNT
MFSR rD, Sr1 MFSR rD, LCR	Move From Load Combine Register	rD = LCR

Mnemonic	Operation	Action
MFSR rD, Sr2 MFSR rD, SCR	Move From Store Combine Register	rD = SCR
MTSR rA, Srn	Move To Special Purpose Register	Srn = rA
MTSR rA, Sr0 MTSR rA, CNT	Move To Counter Register	CNT = rA
MTSR rA, Sr1 MTSR rA, LCR	Move To Load Combine Register	LCR = rA
MTSR rA, Sr2 MTSR rA, SCR	Move To Store Combine Register	SCR = rA

9.3 Custom Engine Instructions

Table 9-11 lists custom engine instructions. For multiplication, the CEH and CEL registers store the results of rA multiply rB. For division, the CEH and CEL registers store the quotient and remainder of rA divided by rB, respectively. The number of cycles required for multiply/divide operations is implementation-dependent. The MFCEH and MFCEL instructions are interlocked so that any attempt to read them before prior operations have completed will cause execution of these instructions to be delayed until the operation finishes.

Table 9-11 Multiplication and Division Instructions

Mnemonic	Operation	Action
MUL rA, rB	Multiply Signed	{CEH, CEL} = rA * rB (rA, rB are treated as signed)
MULU rA, rB	Multiply Unsigned	{CEH, CEL} = rA * rB (rA, rB are treated as unsigned
DIV rA, rB	Divide	CEL = Q(rA/rB), CEH = R(rA/rB) (rA, rB are treated as signed)
DIVU rA, rB	Divide Unsigned	CEL = Q(rA/rB), CEH = R(rA/rB) (rA, rB are treated as unsigned)
MFCEH rD	Move From CEH Register	rD = CEH
MFCEL rD	Move From CEL Register	rD = CEL

MFCEHL rD, rA	Move From CEH and CEL Registers	rD = CEH
		rA = CEL

MTCEH rD	Move To CEH Register	CEH = rD
MTCEL rD	Move To CEL Register	CEL = rD

MTCEHL rD, rA	Move To CEH and CEL Registers	CEH = rD
		CEL = rA

CEINST CEop1, rA(CEop2), rB(CEop3), CEop4, CEop5	Custom Engine Operation	The Custom Engine operation is based on the Five 5-bit CEop field

9.4 Jump and Branch Instructions

This subsection describes the jump and branch instructions of S^+core 32-bit mode.

9.4.1 Jump Instructions

S^+core has two jump instructions, "J" and "JL". The 24-bit displacement is shifted left one bit, then cascade with the most significant 7 bits of J/JL instruction's PC to form the target address. The address of the instruction after the "JL" instruction is placed in the link register.

Table 9-12 Jump Instructions

Mnemonic	Operation	Action
J Disp24	Jump	PC = {PC_{JL}[31:25], Disp24, 1'b0}

JL Disp24	Jump And Link	LR = PC_{JL} + 4
		PC = {PC_{JL}[31:25], Disp24, 1'b0)}

9.4.2 Branches Instructions

The S^+core 32-bit branch instructions can be divided into four types:

- ***Type I:*** Conditional branch register
- ***Type II:*** Conditional branch register and link
- ***Type III:*** Conditional branch displacement
- ***Type IV:*** Conditional branch displacement and link

The target address of type I and II is from the register rA. The 19-bit displacement of type III and IV is shifted left one bit and sign-extended to 32-bit, then added to the PC of the branch instruction to form the target address. Since S^+core ISA has 32/16-bit hybrid instruction execution feature, the instruction must be word-aligned or halfword-aligned. If the instruction of the target address is a 32-bit instruction, the two low order bits of rA should be zero or an address exception will occur when the branch target instruction is subsequently fetched. If the branch target instruction is a 16-bit instruction, the least significant bit of rA should be zero or an address exception will occur. The address of the instruction after the type II and IV branch instruction is placed in the link register.

Table 9-13 Branches Instructions

Mnemonic	Operation	Action

BR<cond> rA	Conditional Branch Register	if (cond)
		PC = rA;
		else
		NOP

Mnemonic	Operation	Action
BRCS/BRGEU	branch register on <Carry set/unsigned greater than or equal>	
BRCC/BRLSU	branch register on <carry clear/unsigned less>	
BRGTU	branch register on <unsigned greater than>	
BRLEU	branch register on <unsigned less than or equal>	
BREQ	branch register on <equal>	
BRNE	branch register on <not equal>	
BRGT	branch register on <signed greater than>	
BRLE	branch register on <signed less than or equal>	
BRGE	branch register on <signed greater than or equal>	
BRLT	branch register on <signed less than>	
BRMI	branch register on <minus/negative>	
BRPL	branch register on <plus/positive or zero>	
BRVS	branch register on <overflow>	
BRVC	branch register on <no overflow>	
BRCNZ	branch register on <counter register not zero> and decrement CNT by one	
BR	branch register	

BR<cond>L rA	Conditional Branch Register And Link	if (cond)
		PC = rA,
		LR = ($PC_{BR<cond>L}$+ 4);
		else
		NOP

BRCSL/BRGEUL	branch register and link on <Carry set/unsigned greater than or equal>	
BRCCL/BRLSUL	branch register and link on <carry clear/unsigned less>	
BRGTUL	branch register and link on <unsigned greater than>	
BRLEUL	branch register and link on <unsigned less than or equal>	
BREQL	branch register and link on <equal>	
BRNEL	branch register and link on <not equal>	
BRGTL	branch register and link on <signed greater than>	
BRLEL	branch register and link on <signed less than or equal>	
BRGEL	branch register and link on <signed greater than or equal>	
BRLTL	branch register and link on <signed less than>	
BRMIL	branch register and link on <minus/negative>	
BRPLL	branch register and link on <plus/positive or zero>	
BRVSL	branch register and link on <overflow>	
BRVCL	branch register and link on <no overflow>	
BRCNZL	branch register and link on <counter register not zero> and decrement CNT by one	
BRL	branch register and link	

B<cond> Disp19	Conditional Branch	if (cond)
		PC = $PC_{B<cond>}$ + SE(Disp19<<1)
		else
		NOP

BCS/BGEU	branch on <Carry set/unsigned greater than or equal>	
BCC/BLSU	branch on <carry clear/unsigned less>	

Mnemonic	Operation	Action
BGTU	branch on <unsigned greater than>	
BLEU	branch on <unsigned less than or equal>	
BEQ	branch on <equal>	
BNE	branch on <not equal>	
BGT	branch on <signed greater than>	
BLE	branch on <signed less than or equal>	
BGE	branch on <signed greater than or equal>	
BLT	branch on <signed less than>	
BMI	branch on <minus/negative>	
BPL	branch on <plus/positive or zero>	
BVS	branch on <overflow>	
BVC	branch on <no overflow>	
BCNZ	branch on <counter register not zero> and decrement CNT by one	
B	branch	

B<cond>L Disp19	Conditional Branch And Link	if (cond)
		PC = $PC_{B<cond>L}$ + SE(Disp19<<1),
		LR = ($PC_{B<cond>L}$ + 4);
		else
		NOP

BCSL/BGEUL	branch and link on <Carry set/unsigned greater than or equal>	
BCCL/BLSUL	branch and link on <carry clear/unsigned less>	
BGTUL	branch and link on <unsigned greater than>	
BLEUL	branch and link on <unsigned less than or equal>	
BEQL	branch and link on <equal>	
BNEL	branch and link on <not equal>	
BGTL	branch and link on <signed greater than>	
BLEL	branch and link on <signed less than or equal>	
BGEL	branch and link on <signed greater than or equal>	
BLTL	branch and link on <signed less than>	
BMIL	branch and link on <minus/negative>	
BPLL	branch and link on <plus/positive or zero>	
BVSL	branch and link on <overflow>	
BVCL	branch and link on <no overflow>	
BCNZL	branch and link on <counter register not zero> and decrement CNT by one	
BL	branch and link	

9.5 Special Instructions

This subsection describes some special instructions, which can be divided into four groups:

- System control instructions
- Cache instructions
- Debug instructions
- Control register instructions

9.5.1 System Control Instructions

The system control instructions, which are listed in Table 9-14, contain system call instruction and conditional trap instructions. The system call and trap instructions will force CPU into Kernel Mode.

Table 9-14 System Control Instructions

Mnemonic	Operation	Action
SYSCALL Imm15	System Call	System call Exception

TRAP<cond> Imm5	Conditional Trap	if(cond) trap
		else NOP

TRAPCS/TRAPGEU	trap on <Carry set/unsigned greater than or equal>	
TRAPCC/TRAPLSU	trap on <carry clear/unsigned less>	
TRAPGTU	trap on <unsigned greater than>	
TRAPLEU	trap on <unsigned less than or equal>	
TRAPEQ	trap on <equal>	
TRAPNE	trap on <not equal>	
TRAPGT	trap on <signed greater than>	
TRAPLE	trap on <signed less than or equal>	
TRAPGE	trap on <signed greater than or equal>	
TRAPLT	trap on <signed less than>	
TRAPMI	trap on <minus/negative>	
TRAPPL	trap on <plus/positive or zero>	
TRAPVS	trap on <overflow>	
TRAPVC	trap on <no overflow>	
TRAP	trap	

T<cond>	Set T Flag On Condition	if {cond} T flag=1
		else T flag =0

TCS/TGEU	set T flag on <Carry set/unsigned greater than or equal>	
TCC/TLSU	set T flag on <carry clear/unsigned less>	
TGTU	set T flag on <unsigned greater than>	
TLEU	set T flag on <unsigned less than or equal>	
TEQ	set T flag on <equal>	
TNE	set T flag on <not equal>	
TGT	set T flag on <signed greater than>	
TLE	set T flag on <signed less than or equal>	
TGE	set T flag on <signed greater than or equal>	
TLT	set T flag on <signed less than>	
TMI	set T flag on <minus/negative>	
TPL	set T flag on <plus/positive or zero>	
TVS	set T flag on <overflow>	
TVC	set T flag on <no overflow>	
TCNZ	set T flag on <CNT not zero>	
TSET	set T flag	

9.5.2 Cache Instructions

The second group of the special instructions is cache instruction that is listed in Table 9-15. The cache operation is based on the Cache_op field that is embedded in the cache instruction.

Table 9-15 Cache Instructions

Cache-OP[4:0]	I-Cache/ D-Cache	Function	Data
0x00	I-Cache	Pre-fetch a Cache-line	VA
0x01	I-Cache	Pre-fetch and lock a Cache-line	VA
0x02	I-Cache	Invalid and unlock a Cache-line	VA
0x03	I-Cache	Fill LIM (local instruction memory) device	VA (PFN & Size)
0x04	I-Cache	Re-Fill LIM (local instruction memory with the PFN and Size of previous value) device	VA
0x08	D-Cache	Pre-fetch a Cache-line	VA
0x09	D-Cache	Pre-fetch and lock a Cache-line	VA
0x0A	D-Cache	Invalid and unlock a Cache-line	VA
0x0B	D-Cache	Fill LDM (local data memory) device	VA (PFN & Size)
0x0C	D-Cache	Write-Back LDM (local data memory) device to main memory	NA
0x0D	D-Cache	Force write-back a Cache-line and set valid when the cache-line is valid and dirty	VA
0x0E	D-Cache	Force write-back a Cache-line and set invalid when the cache-line is valid and dirty	VA
0x10	I-Cache	Invalid entire cache	NA
0x11	I-Cache	Toggle Instruction Pre-fetch Buffer Function (Enable/Disable)	NA
0x18	D-Cache	Invalid entire cache	NA
0x1A	D-Cache	Drain Write Buffer	NA
0x1B	D-Cache	Toggle Write Buffer Function	NA
0x1C	D-Cache	Toggle Data Pre-fetch Buffer Function (Enable/Disable)	NA
0x1D	D-Cache	Toggle Write-back D-Cache Function (Enable/Disable)	NA
0x1E	D-Cache	Force write-back entire D-Cache and set valid of the cache-lines are valid and dirty. (Write-out)	NA
0x1F	D-Cache	Force write-back entire D-Cache and set invalid of the cache-lines are valid and dirty. (Flush)	NA

9.5.3 Debug Instructions

Table 9-16 lists the debug instructions of S^{+}core 32-bit mode.

Table 9-16 Debug Instructions

Mnemonic	Operation	Action

SDBBP code	Software Debug Break Point	if(**DREG**$_{DM}$ ==0) then
		DEPC ← **PC**
		DREG$_{DM}$ ← 1
		DREG$_{DBP}$ ← 1
		if (**DREG**$_{ProbEn}$ == 1) then
		    PC ← *0xFF00_0000*
		else then
		    PC ← {EXCPVec$_{Base}$, 16'h0} + *0x1FC*

DRTE	Return from debug exception	PC ← DEPC DREG$_{DM}$ ← 0

9.5.4 Control Register Instructions

Table 9-17 lists the control register instructions, which perform operations on the system control registers to manipulate the memory management and exception handling facilities of the processor. When the processor enters kernel mode, it remains in kernel mode until a return from exception (RTE) instruction is executed. S+core also provide low power instruction SLEEP. SLEEP instruction would hold CPU until next interrupt come in to wake up CPU.

Table 9-17 CR Instructions

Mnemonic	Operation	Action
DRTE	Return From Debug Exception	PC ← DEPC FREG$_{DM}$ ← 0
RTE	Return From Exception	PC=EPC, Restore PSR Bits, Restore Condition Bits,
SLEEP	Sleep	CPU would hold until interrupt to wake up
MFCR rD, CR	Move From Control register	rD = CR
MTCR rD, CR	Move To Control register	CR = rD

9.6 Coprocessor Instructions

There are three groups of coprocessor instructions:

- Coprocessor z register transfer instructions (z is from 1 to 3)
- Coprocessor z memory access instructions (z is from 1 to 3)
- Coprocessor z operation instructions (z is from 1 to 3)

S+core has up to three additional external coprocessors.

9.6.1 Coprocessor z Register Transfer Instructions

Table 9-18 summarizes the coprocessor register transfer instructions which can transfer data between S^+core general purpose registers and coprocessor registers.

Table 9-18 Coprocessor Register Transfer Instructions

Mnemonic	**Operation**	**Action**
MTCz rD, CrA	Move To Coprocessor z	COPz(CrA) = rD
MFCz rD, CrA	Move From Coprocessor z	RD = COPz(CrA)

9.6.2 Coprocessor z Memory Access Instructions

Table 9-19 lists the coprocessor memory access instructions which can transfer data between memory and coprocessor registers. The 10-bit immediate is shifted left two bits and sign-extended, then added to the contents of general purpose register rD to form the effective address. If either of the two least significant bits of the effective address is non-zero, an address error exception occurs.

Table 9-19 Coprocessor Memory Access Instructions

Mnemonic	**Operation**	**Action**
LDCz CrA, [rD, $SImm_{10}$]	Load To Coprocessor	COPz(CrA) = Mem_{32}[rD+SE($SImm_{10}$<<2)]
STCz CrA, [rD, $SImm_{10}$]	Store From Coprocessor	Mem_{32}[rD+SE($SImm_{10}$<<2)] = COPz(CrA)

9.6.3 Coprocessor z Operation Instructions

Table 9-20 lists the coprocessor instruction which performs a coprocessor operation. The coprocessor action can be decided by the function which is supported by the coprocessor and the COP-Code.

Table 9-20 Coprocessor Operation Instructions

Mnemonic	Operation	Action
COPz CrD, CrA, CrB, COP-Code	Coprocessor Operation Instruction	The Action is based on the function of coprocessor

10 16-Bit Instructions

The S^+core 16-bit instructions allow embedded designs to reduce system cost by reducing overall memory requirements. The mnemonic, operation and action for each instruction will be listed in the tables of the following subsections. The mnemonic and action performed by each instruction are using a high-level language notation. Parallel conditional execution (PCE) is a Sunplus-patent-pended feature to avoid the branch penalty. Assembly programmers use double pipe (||) to specify a pair of PCE execution instructions. For example:

```
ADD! r2, r7 || SUB! r2, r7
```

The add instruction would be executed if T flag is true while the sub instruction would be executed if T flag is false. Special symbols used in the notation are described in Table **10-1**.

Table 10-1 CPU Instruction Action Notations

Symbol	Meaning	Example
Imm_n	n-bit unsigned immediate	
$SImm_n$	n-bit signed immediate	
Mem_n[address]	n-bit content of the memory that specified by address	
k'bX	k-bit binary representation of X value	2'b11 = 3
k'hX	k-bit hexadecimal representation of X value	8'h2a = 42
X[n]	selection of bit n of bit string X	X=2'b10, X[1]=2'b1
X[n:m]	selection of bits n through m of bit string X	X = 6'b111001, X[3:0] = 4'b1001
k{A}	replication of bit value A into a k-bit string	K = 0, 2{0} = 2'b00
k{X[n]}	replication of bit value X[n] into a k-bit string	X = 2'b01, 2{X[1]} = 2'b00
k{X[n:m]}	replication of bit value X[n:m] into a (k*(n-m+1))-bit string	X = 2'b01, 2{X[1:0]} = 4'b0101
{X, Y}	concatenation of X,Y	X = 2'b11, Y=2'b00, {X, Y} = 4'b1100
SE(X)	sign extend X to a 32-bit string	X=8'hfe, SE(X) = 32'hfffe
ZE(X)	zero extend X to a 32-bit string	X=8'hfe, ZE(X) = 32'h00fe
+	add	
-	subtract	
*	multiplication	
/	division	
Q(X/Y)	Quotient of X divided by Y	Q(5/2) = 2
R(X/Y)	Remainder of X divided by Y	R(5/2) = 1
&	bitwise logic and	X=2'b11, Y=2'b00, X&Y = 2'b00
|	bitwise logic or	X=2'b11, Y=2'b00, X|Y =2'b11
~	bitwise logic not	X=2'b11, Y = ~X, Y=2'b00
^	bitwise logic xor	X=2'b11, Y=2'b01, X^Y = 2'b10

Symbol	Meaning	Example
<<	shift left	
ROR	rotate right	
ROL	rotate left	
X^Y	this represents X to the power of Y	$10^3 = 1000$
cond	cond is true if the icc test according to EC/BC field in the instruction is true	BEQ target. BC = 4'b0000, the corresponding icc test is Z flag test. In this case, if Z flag is true, cond is true; otherwise, cond is false.
<cond>	suffix of conditional execution/branch instructions which specifies the execution/branch condition(EC/BC)	
LR	link register, generally is R3	
BP	base pointer register	
GPR[Rn]	this indicates that Rn is a general purpose register	
COPz(Rn)	this indicates that Rn which belongs to COPz	
T	T flag	
C	carry flag	
Z	zero flag	

S^+core 16-bit instructions can be divided into the following functional categories:

- Load and store instructions
- Data processing instructions
- Jump and branch instructions
- Special instructions

Each instruction is 16 bits long. The mnemonic, operation and action for each instruction will be listed in the tables of the following subsections.

10.1 Load/Store Instructions

The load/store instructions transfer data between register and memory.

Table **9-2** lists the supported data types of load/store instructions.

Table 10-2 Data Types for Load and Store instructions

Data Type		Supported Operation	Note
Byte	Unsigned	Load, Store	
Halfword	Signed	Load	Must be aligned to halfword boundary. The 5-bit unsigned immediate is shifted left one bit and zero-extended.
	Unsigned	Store	
Word		Load, Store	Must be aligned to word boundary. The 5-bit unsigned immediate is shifted left two bits and zero-extended.

The addressing mode of S^{+}core is formed from general purpose register or base pointer register and offset, which are used in four different ways to form the effective address:

- ***Register:*** The effective address is the content of the general purpose register.
- ***Offset:*** The base pointer register and offset are added or subtracted to form the effective address. The byte access immediate value will be signed-extended. The 5-bit immediate value of halfword access is shifted left one bit and zero-extended. The 5-bit word access immediate value is shifted left two bits and zero-extended.
- ***Pre-indexed:*** The general purpose register subtracts 4 to form the effective address. The general purpose register is then updated with this new effective address, to allow automatic indexing through an array or memory block.
- ***Post-indexed:*** The effective address is formed by the content of the general purpose register. The general purpose register is updated by adding 4 to it, to allow automatic indexing through an array or memory block.

10.1.1 Load Instructions

Table 10-3 summarizes the S^{+}core 16-bit load instructions.

Table 10-3 Load Instructions

Mnemonic	Operation	Action
LBU! rD_{g0}, [rA_{g0}]	Load Byte Unsigned	rD_{g0} = ZE(Mem_8[rA_{g0}])
LBUP! rD_{g0}, Imm_5	Load Byte Unsigned With Base Pointer	rD_{g0} = ZE(Mem_8[BP+ZE(Imm_5)])
LH! rD_{g0}, [rA_{g0}]	Load Halfword Signed	rD_{g0} = SE(Mem_{16}[rA_{g0}])
LHP! rD_{g0}, Imm_6	Load Halfword Signed With Base Pointer	rD_{g0} = SE(Mem_{16}[BP+ZE({Imm_6[5:1], 1'b0})])
LW! rD_{g0}, [rA_{g0}]	Load Word	rD_{g0} = Mem_{32}[rA_{g0}]
LWP! rD_{g0}, Imm_7	Load Word With Base Pointer	rD_{g0} = Mem_{32}[BP + ZE({Imm_7[6:2], 2'b0})]

Mnemonic	Operation	Action

POP! rD, [rA]	Load Word (Post-index)	rD = Mem_{32}[rA],
		rA = rA + 4
		∗ rA could only indicates r0 ~ r7
		∗ rD could indicates r0 ~ r31

10.1.2 Store Instructions

The store instructions for S^+core 16-bit mode is listed in Table 10-4.

Table 10-4 Store Instructions

Mnemonic	Operation	Action
SB! rD_{g0}, [rA_{g0}]	Store Byte	Mem_8[rA_{g0}] = rD_{g0}[7:0]
SBP! rD_{g0}, Imm_5	Store Byte With Base Pointer	Mem_8[BP+ZE(Imm_5)] = rD_{g0}[7:0]
SH! rD_{g0}, [rA_{g0}]	Store Halfword	Mem_{16}[rA_{g0}] = rD_{g0}[15:0]
SHP! rD_{g0}, Imm_6	Store Halfword With Base Pointer	Mem_{16}[BP+ZE({Imm_6[5:1], 1'b0})] = rD_{g0}[15:0]
SW! rD_{g0}, [rA_{g0}]	Store Word	Mem_{32}[rA_{g0}] = rD_{g0}
SWP! rD_{g0}, Imm_7	Store Word With Base Pointer	Mem_{32}[BP+ZE({Imm_7[6:2], 2'b0})] = rD_{g0}
PUSH! rD, [rA]	Store Word (Pre-index)	Mem_{32}[rA - 4] = rD_g, rA = rA – 4 ∗ rA could only indicates r0 ~ r7 ∗ rD could indicates r0 ~ r31

10.2 Data processing Instructions

There are five groups of S^+core 16-bit data processing instructions:

- Arithmetic instructions
- Logical instructions
- Shift instructions
- Move instructions

10.2.1 Arithmetic Instructions

The arithmetic instructions are summarized in Table **10-5**. This group contains the instructions which perform add or subtract operation. The 4-bit sign immediate value of ADDEI and SUBEI instructions represents the exponent of the radix-2 addend.

Table 10-5 Arithmetic Instructions

Mnemonic	Operation	Action
ADD! rD_{g0}, rA_{g0}	Add	rD_{g0} = rD_{g0} + rA_{g0}

Mnemonic	Operation	Action
ADDC! rD_{g0}, rA_{g0}	Add With Carry	$rD_{g0} = rD_{g0} + rA_{g0} + C$
ADDEI! rD_{g0}, Imm_4	Add Exponent Immediate	$rD_{g0} = rD_{g0} + 2^{Imm4}$
SUB! rD_{g0}, rA_{g0}	Subtract	$rD_{g0} = rD_{g0} - rA_{g0}$
SUBEI! rD_{g0}, Imm_4	Subtract Exponent Immediate	$rD_{g0} = rD_{g0} - 2^{Imm4}$
CMP! rD_{g0}, rA_{g0}	Compare	$rD_{g0} - rA_{g0}$
NEG! rD_{g0}, rA_{g0}	Negative	$rD_{g0} = 0 - rA_{g0}$

10.2.2 Logical Instructions

The logical instructions are summarized in Table 10-6. This group contains the logical operation instructions.

Table 10-6 Logical Instructions

Mnemonic	Operation	Action
AND! rD_{g0}, rA_{g0}	Logical And	$rD_{g0} = rD_{g0}$ & rA_{g0}
BITCLR! rD_{g0}, BN	Clear Bit In Register	$rD_{g0}[BN] = 0$
BITSET! rD_{g0}, BN	Set Bit In Register	$rD_{g0}[BN] = 1$
BITTGL! rD_{g0}, BN	Toggle Bit In Register	$rD_{g0}[BN] = \sim rD_{g0}[BN]$
BITTST! rD_{g0}, BN	Test Bit In Register	Z flag = $\sim rD_{g0}[BN]$
LIU! rD_{g0}, Imm_8	Load Immediate	$rD_{g0} = \{24\{0\}, Imm_8\}$
NOP!	No Operation	
NOT! rD_{g0}, rA_{g0}	Logical Not	$rD_{g0} = \sim rA_{g0}$
OR! rD_{g0}, rA_{g0}	Logical Or	$rD_{g0} = rD_{g0} \mid rA_{g0}$
XOR! rD_{g0}, rA_{g0}	Logical Xor	$rD_{g0} = rD_{g0}$ ^ rA_{g0}

10.2.3 Shift Instructions

Table 10-7 lists the shift instructions of S^+core 16-bit mode. The shift amount (SA) field is a 5-bit immediate which are embedded in the instruction word. The shift operations by register rA_{g0} only use the low order 5 bits of register rA_{g0} to specify the number of bits to shift.

Table 10-7 Shift Instructions

Mnemonic	Operation	Action
SRA! rD_{g0}, rA_{g0}	Shift Right Arithmetic	s= $rA_{g0}[4:0]$, $rD_{g0} = \{ s\{rD_{g0}[31]\}, rD_{g0}[31:s]\}$
SRL! rD_{g0}, rA_{g0}	Shift Right Logical	s= $rA_{g0}[4:0]$, $rD_{g0} = \{s\{0\}, rD_{g0} [31:s]\}$
SLL! rD_{g0}, rA_{g0}	Shift Left Logical	s= $rA_{g0}[4:0]$, $rD_{g0} = \{ rD_{g0} [(31-s):0], s\{0\}\}$
SLLI! rD_{g0}, SA	Shift Left Logical Immediate	$rD_{g0} = \{ rD_{g0} [(31-SA):0], SA\{0\}\}$

Mnemonic	Operation	Action
SRLI! rD_{g0}, SA	Shift Right Logical Immediate	rD_{g0} = {SA{0}, rD_{g0} [31:(31-SA)}

10.2.4 Move Instructions

Table 10-8 lists the move instructions, which support data transfer between two general purpose registers in group 0 (r0~r15) or between one general purpose register in group 0 and another is in group 1 (r16~r31).

Table 10-8 Move Instructions

Mnemonic	Operation	Action
MV! rD_{g0}, rA_{g0}	Move On Condition	rD_{g0} = rA_{g0}
MLFH! rD_{g0}, rA_{g1}	Move From r16~r31	rD_{g0} = rA_{g1}
MHFL! rD_{g1}, rA_{g0}	Move From r0~r15	rD_{g1} = rA_{g0}

10.3 Jump and Branch Instruction

This subsection describes the jump and branch instructions of S^+core 16-bit mode.

10.3.1 Jump Instructions

S^+core has two 16-bit jump instructions, "J!" and "JL!". The 11-bit displacement is shifted left one bit, then cascade with the most significant 20 bits of JUMP instruction's PC to form the target address. The address of the instruction after the "JL!" instruction is placed in the link register.

Table 10-9 Jump Instructions

Mnemonic	Operation	Action
J! Disp11	Jump	PC = {$PC_{J!}$ [31:12], Disp11, 1'b0}

JL! Disp11	Jump And Link	LR = $PC_{JL!}$ + 2
		PC = {$PC_{J!}$ [31:12], Disp11, 1'b0}

10.3.2 Branch Instructions

The S^+core 16-bit branch instructions can be divided into three types:

- ***Type I:*** Conditional branch register
- ***Type II:*** Conditional branch register and link
- ***Type III:*** Conditional branch displacement

The target address of type I and I is from the register rA_{g0}. The 8-bit displacement of type III is shifted left one bit and sign-extended to 32-bit, then added to the PC value of conditional branch instruction to form the target address. Since S^{+}core ISA has 32/16-bit hybrid instruction execution feature, the instruction must be word-aligned or halfword-aligned. If the instruction of the target address is a 32-bit instruction, the two low order bits of rA_{g0} should be zero or an address exception will occur when the branch target instruction is subsequently fetched. If the branch target instruction is a 16-bit instruction, the least significant bit of rA_{g0} should be zero or an address exception will occur. The address of the instruction after the type II branch instruction is placed in the link register.

Table 10-10 Branch Instructions

Mnemonic	Operation	Action

BR<cond>! rA_{g0}	Conditional Branch Register	if (cond)
		PC = rA_{g0}
		else
		NOP

BRCS!/BRGEU!	branch register on <Carry set/unsigned greater than or equal>	
BRCC!/BRLSU!	branch register on <carry clear/unsigned less>	
BRGTU!	branch register on <unsigned greater than>	
BRLEU!	branch register on <unsigned less than or equal>	
BREQ!	branch register on <equal>	
BRNE!	branch register on <not equal>	
BRGT!	branch register on <signed greater than>	
BRLE!	branch register on <signed less than or equal>	
BRGE!	branch register on <signed greater than or equal>	
BRLT!	branch register on <signed less than>	
BRMI!	branch register on <minus/negative>	
BRPL!	branch register on <plus/positive or zero>	
BRVS!	branch register on <overflow>	
BRVC!	branch register on <no overflow>	
BRCNZ!	branch register on <counter register not zero> and decrement CNT by one	
BR!	branch register	

BR<cond>L! rA_{g0}	Conditional Branch Register And Link	if (cond)
		PC = rA_{g0},
		LR = ($PC_{BR<cond>L!}$ + 2)
		else
		NOP

BRCSL!/BRGEUL!	branch register and link on <Carry set/unsigned greater than or equal>	
BRCCL!/BRLSUL!	branch register and link on <carry clear/unsigned less>	
BRGTUL!	branch register and link on <unsigned greater than>	
BRLEUL!	branch register and link on <unsigned less than or equal>	
BREQL!	branch register and link on <equal>	
BRNEL!	branch register and link on <not equal>	
BRGTL!	branch register and link on <signed greater than>	

Mnemonic	Operation	Action
BRLEL!	branch register and link on <signed less than or equal>	
BRGEL!	branch register and link on <signed greater than or equal>	
BRLTL!	branch register and link on <signed less than>	
BRMIL!	branch register and link on <minus/negative>	
BRPLL!	branch register and link on <plus/positive or zero>	
BRVSL!	branch register and link on <overflow>	
BRVCL!	branch register and link on <no overflow>	
BRCNZL!	branch register and link on <counter register not zero> and decrement CNT by one	
BRL!	branch register and link	

B<cond>! Disp8	Conditional Branch	if (cond)
		  PC = $PC_{B<cond>!}$ + SE(Disp8<<1)
		else
		  NOP

BCS!/BGEU!	branch on <Carry set/unsigned greater than or equal>	
BCC!/BLSU!	branch on <carry clear/unsigned less>	
BGTU!	branch on <unsigned greater than>	
BLEU!	branch on <unsigned less than or equal>	
BEQ!	branch on <equal>	
BNE!	branch on <not equal>	
BGT!	branch on <signed greater than>	
BLE!	branch on <signed less than or equal>	
BGE!	branch on <signed greater than or equal>	
BLT!	branch on <signed less than>	
BMI!	branch on <minus/negative>	
BPL!	branch on <plus/positive or zero>	
BVS!	branch on <overflow>	
BVC!	branch on <no overflow>	
BCNZ!	branch on <counter register not zero> and decrement CNT by one	
B!	branch	

10.4 Special Instructions

This section describes some special instructions, which can be divided into 2 groups:

- System Control Instructions
- Debug Instructions

10.4.1 System Control Instructions

The S$^+$core 16-bit system control instructions are listed in

Table **10-11**.

Table 10-11 System Control Instructions

Mnemonic	Operation	Action

T<cond>!	Set T Flag On Condition	if {cond}
		  T flag=1
		else
		  T flag =0

TCS!/TGEU!	set T flag on <Carry set/unsigned greater than or equal>	
TCC!/TLSU!	set T flag on <carry clear/unsigned less>	
TGTU!	set T flag on <unsigned greater than>	
TLEU!	set T flag on <unsigned less than or equal>	
TEQ!	set T flag on <equal>	
TNE!	set T flag on <not equal>	
TGT!	set T flag on <signed greater than>	
TLE!	set T flag on <signed less than or equal>	
TGE!	set T flag on <signed greater than or equal>	
TLT!	set T flag on <signed less than>	
TMI!	set T flag on <minus/negative>	
TPL!	set T flag on <plus/positive or zero>	
TVS!	set T flag on <overflow>	
TVC!	set T flag on <no overflow>	
TCNZ!	set T flag on <CNT not zero>	
TSET!	set T flag	

10.4.2 Debug Instructions

Table 10-12 lists the S^+core 16-bit debug instructions.

Table 10-12 Debug Instructions

Mnemonic	Operation	Action

SDBBP! code	Software Debug Break Point	if(**DREG**$_{DM}$ ==0) then
		DEPC ← **PC**
		DREG$_{DM}$ ← 1
		DREG$_{DBP}$ ← 1
		if (**DREG**$_{ProbEn}$ == 1) then
		  PC ← *0xFF00_0000*
		else then
		  PC ← {EXCPVec$_{Base}$, 16'h0} + *0x1FC*

11 Instruction List by Type

11.1 32-Bit Instructions

Type	Instruction	Section	Table	Appendix A
Load Instructions	LB	9.1.1 (P.101)	Table 9-4 (P.101)	
	LBU	9.1.1 (P.101)	Table 9-4 (P.101)	
	LB.B	9.1.1 (P.101)	Table 9-4 (P.101)	
	LBU.B	9.1.1 (P.101)	Table 9-4 (P.101)	
	LB.A	9.1.1 (P.101)	Table 9-4 (P.101)	
	LBU.A	9.1.1 (P.101)	Table 9-4 (P.101)	
	LH	9.1.1 (P.101)	Table 9-4 (P.101)	
	LHU	9.1.1 (P.101)	Table 9-4 (P.101)	
	LH.B	9.1.1 (P.101)	Table 9-4 (P.101)	
	LHU.B	9.1.1 (P.101)	Table 9-4 (P.101)	
	LH.A	9.1.1 (P.101)	Table 9-4 (P.101)	
	LHU.A	9.1.1 (P.101)	Table 9-4 (P.101)	
	LW	9.1.1 (P.101)	Table 9-4 (P.101)	
	LW.B	9.1.1 (P.101)	Table 9-4 (P.101)	
	LW.A	9.1.1 (P.101)	Table 9-4 (P.101)	
	LCB	9.1.1 (P.101)	Table 9-4 (P.101)	
	LCW	9.1.1 (P.101)	Table 9-4 (P.101)	
	LCE	9.1.1 (P.101)	Table 9-4 (P.101)	
	ALW	9.1.1 (P.101)	Table 9-4 (P.101)	
Store Instructions	SB	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SB.B	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SB.A	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SH	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SH.B	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SH.A	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SW	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SW.B	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SW.A	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SCB	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SCW	9.1.2 (P. 103)	Table 9-5 (P.103)	
	SCE	9.1.2 (P. 103)	Table 9-5 (P.103)	
	ASW	9.1.2 (P. 103)	Table 9-5 (P.103)	
Arithmetic Instructions	ADD	9.2.1 (P. 104)	Table 9-6 (P.104)	
	ADDC	9.2.1 (P. 104)	Table 9-6 (P.104)	
	ADDI	9.2.1 (P. 104)	Table 9-6 (P.104)	
	ADDIS	9.2.1 (P. 104)	Table 9-6 (P.104)	
	ADDRI	9.2.1 (P. 104)	Table 9-6 (P.104)	

Type	Instruction	Section	Table	Appendix A
	SUB	9.2.1 (P. 104)	Table 9-6 (P.104)	
	SUBC	9.2.1 (P. 104)	Table 9-6 (P.104)	
	CMP<cond>.c	9.2.1 (P. 104)	Table 9-6 (P.104)	
	CMPZ<cond>.c	9.2.1 (P. 104)	Table 9-6 (P.104)	
	CMPI.c	9.2.1 (P. 104)	Table 9-6 (P.104)	
	NEG	9.2.1 (P. 104)	Table 9-6 (P.104)	
Logical Instructions	AND	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ANDI	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ANDIS	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ANDRI	9.2.2 (P. 105)	Table 9-7 (P.105)	
	BITCLR	9.2.2 (P. 105)	Table 9-7 (P.105)	
	BITSET	9.2.2 (P. 105)	Table 9-7 (P.105)	
	BITTGL	9.2.2 (P. 105)	Table 9-7 (P.105)	
	BITTST	9.2.2 (P. 105)	Table 9-7 (P.105)	
	LDI	9.2.2 (P. 105)	Table 9-7 (P.105)	
	LDIS	9.2.2 (P. 105)	Table 9-7 (P.105)	
	NOP	9.2.2 (P. 105)	Table 9-7 (P.105)	
	NOT	9.2.2 (P. 105)	Table 9-7 (P.105)	
	OR	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ORI	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ORIS	9.2.2 (P. 105)	Table 9-7 (P.105)	
	ORRI	9.2.2 (P. 105)	Table 9-7 (P.105)	
	XOR	9.2.2 (P. 105)	Table 9-7 (P.105)	
Shift / Rotate Instructions	SRA	9.2.3(P.106)	Table 9-8 (P.106)	
	SRL	9.2.3(P.106)	Table 9-8 (P.106)	
	SLL	9.2.3(P.106)	Table 9-8 (P.106)	
	SRAI	9.2.3(P.106)	Table 9-8 (P.106)	
	SRLI	9.2.3(P.106)	Table 9-8 (P.106)	
	SLLI	9.2.3(P.106)	Table 9-8 (P.106)	
	ROR	9.2.3(P.106)	Table 9-8 (P.106)	
	ROL	9.2.3(P.106)	Table 9-8 (P.106)	
	RORC.c	9.2.3(P.106)	Table 9-8 (P.106)	
	ROLC.c	9.2.3(P.106)	Table 9-8 (P.106)	
	RORI	9.2.3(P.106)	Table 9-8 (P.106)	
	ROLI	9.2.3(P.106)	Table 9-8 (P.106)	
	RORIC.c	9.2.3(P.106)	Table 9-8 (P.106)	
	ROLIC.c	9.2.3(P.106)	Table 9-8 (P.106)	
Extension Instructions	EXTSB	9.2.4 (P. 107)	Table 9-9 (P.107)	
	EXTSH	9.2.4 (P. 107)	Table 9-9 (P.107)	
	EXTZB	9.2.4 (P. 107)	Table 9-9 (P.107)	
	EXTZH	9.2.4 (P. 107)	Table 9-9 (P.107)	
Move Instructions	MV<cond>	9.2.5 (P. 107)	Table 9-10 (P.107)	

Type	Instruction	Section	Table	Appendix A
	MFSR	9.2.5 (P. 107)	Table 9-10 (P.107)	
	MTSR	9.2.5 (P. 107)	Table 9-10 (P.107)	
Custom Engine Instructions	MUL	9.3 (P. 108)	Table 9-11 (P.108)	
	MULU	9.3 (P. 108)	Table 9-11 (P.108)	
	DIV	9.3 (P. 108)	Table 9-11 (P.108)	
	DIVU	9.3 (P. 108)	Table 9-11 (P.108)	
	MFCEH	9.3 (P. 108)	Table 9-11 (P.108)	
	MFCEL	9.3 (P. 108)	Table 9-11 (P.108)	
	MFCEHL	9.3 (P. 108)	Table 9-11 (P.108)	
	MTCEH	9.3 (P. 108)	Table 9-11 (P.108)	
	MTCEL	9.3 (P. 108)	Table 9-11 (P.108)	
	MTCEHL	9.3 (P. 108)	Table 9-11 (P.108)	
	CE	9.3 (P. 108)	Table 9-11 (P.108)	
Jump Instructions	J	9.4.1 (P. 109)	Table 9-12 (P.109)	
	JL	9.4.1 (P. 109)	Table 9-12 (P.109)	
Branches Instructions	B<cond>	9.4.2 (P. 109)	Table 9-13 (P.109)	
	B<cond>L	9.4.2 (P. 109)	Table 9-13 (P.109)	
	BR<cond>	9.4.2 (P. 109)	Table 9-13 (P.109)	
	BR<cond>L	9.4.2 (P. 109)	Table 9-13 (P.109)	
System Control Instructions	SYSCALL	9.5.1 (P. 112)	Table 9-14 (P.112)	
	TRAP<cond>	9.5.1 (P. 112)	Table 9-14 (P.112)	
	T<cond>	9.5.1 (P. 112)	Table 9-14 (P.112)	
Cache Instructions	CACHE	9.5.2 (P. 113)	Table 9-15 (P.113)	
Debug Instructions	SDBBP	9.5.3 (P.114)	Table 9-16 (P.114)	
	DRTE	9.5.3 (P.114)	Table 9-16 (P.114)	
Control Register Instructions	DRTE	9.5.4 (P.114)	Table 9-17(P.114)	
	RTE	9.5.4 (P.114)	Table 9-17(P.114)	
	MTCR	9.5.4 (P.114)	Table 9-17(P.114)	
	MFCR	9.5.4 (P.114)	Table 9-17(P.114)	
Coprocessor Register Transfer Instructions	MTCz	9.6.1 (P. 115)	Table 9-18 (P.115)	
	MFCz	9.6.1 (P. 115)	Table 9-18 (P.115)	
Coprocessor Memory Access Instructions	LDCz	9.6.3 (P. 116)	Table 9-19 (P.115)	
	STCz	9.6.3 (P. 116)	Table 9-19 (P.115)	
Coprocessor Operation Instructions	COPz	9.6.3 (P. 116)	Table 9-20 (P. 116)	

11.2 16-Bit Instructions

Type	Instruction	Section	Table	Appendix A
Load Instructions	LBU!	10.1.1 (P.119)	Table 10-3 (P. 119)	

Type	Instruction	Section	Table	Appendix A
	LBUP!	10.1.1 (P.119)	Table 10-3 (P. 119)	
	LH!	10.1.1 (P.119)	Table 10-3 (P. 119)	
	LHP!	10.1.1 (P.119)	Table 10-3 (P. 119)	
	LW!	10.1.1 (P.119)	Table 10-3 (P. 119)	
	LWP!	10.1.1 (P.119)	Table 10-3 (P. 119)	
	POP!	10.1.1 (P.119)	Table 10-3 (P. 119)	
Store Instructions	SB!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	SBP!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	SH!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	SHP!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	SW!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	SWP!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
	PUSH!	10.1.2 (P. 120)	Table 10-4 (P. 120)	
Arithmetic Instructions	ADD!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	ADDC!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	ADDEI!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	SUB!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	SUBEI!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	CMP!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
	NEG!	10.2.1 (P. 120)	Table 10-5 (P. 120)	
Logical Instructions	AND!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	BITCLR!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	BITSET!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	BITTGL!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	BITTST!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	LDIU!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	NOP!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	NOT!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	OR!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
	XOR!	10.2.2 (P. 121)	Table 10-6 (P. 121)	
Shift Instructions	SRA!	10.2.3 (P. 121)	Table 10-7 (P. 121)	
	SRL!	10.2.3 (P. 121)	Table 10-7 (P. 121)	
	SLL!	10.2.3 (P. 121)	Table 10-7 (P. 121)	
	SLLI!	10.2.3 (P. 121)	Table 10-7 (P. 121)	
Move Instructions	MV!	10.2.4 (P. 122)	Table 10-8 (P. 122)	
	MLFH!	10.2.4 (P. 122)	Table 10-8 (P. 122)	
	MHFL!	10.2.4 (P. 122)	Table 10-8 (P. 122)	
Jump Instructions	J!	10.3.1 (P. 122)	Table 10-9 (P. 122)	
	JL!	10.3.1 (P. 122)	Table 10-9 (P. 122)	
Branches Instructions	B<cond>!	0 (P. 122)	Table 10-10 (P. 123)	
	BR<cond>!	0 (P. 122)	Table 10-10 (P. 123)	
	BR<cond>L!	0 (P. 122)	Table 10-10 (P. 123)	

Type	Instruction	Section	Table	Appendix A
System Control Instructions	T<cond>!	10.4.1 (P. 124)	Table **10-11** (P. 126)	
Debug Instructions	SDBBP!	10.4.2 (P. 126)	Table 10-12 (P. 126)	