S+core7 Programming Guide

V1.4 - Mar 22, 2007

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

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

Important Notice

Sunplus Technology reserves the right to change this documentation without prior notice. Information provided by Sunplus Technology is believed to be accurate and reliable. However, Sunplus Technology makes no warranty for any errors which may appear in this document. Contact Sunplus Technology to obtain the latest version of device specifications before placing your order. No responsibility is assumed by Sunplus Technology for any infringement of patent or other rights of third parties which may result from its use. In addition, Sunplus products are not authorized for use as critical components in life support devices/ systems or aviation devices/systems, where a malfunction or failure of the product may reasonably be expected to result in significant injury to the user, without the express written approval of Sunplus.

Table of Content

PAGE

Revision History

Revision	Date	By	Remark
1.0	2005/07/05	J.Y Wu	First Edition
1.1	2006/01/29	Pei-Lin Tsai	Second Edition
1.2	2006/04/21	C.Y. Lee	Third Edition: added enhanced MAC.
1.3	2006/06/05	Pei-Lin Tsai	Forth Edition: added data dependency rules
1.4	2007/03/22	Ya-min Chang Pei-Lin Tsai	Fifth Edition: revised LCB, LCW, LCE, SCB, SCW, SCE; Br{cond}! mv table Added 6.6 S+core ABI Added 6.7 S+core Relocation Added 3.4 Using Inline Assembly Added 4.3 S+core Specific Directives

1 Instruction Formats

1.1 Instruction Set Forms

The S+core processor has 32-bit and 16-bit instruction modes. The two P bits located at 31st and 15th are used to distinguish the instruction mode as shown in Fig 1-1. Therefore the instruction width is 30 bits for 32-bit instruction and 15 bits for 16-bit instruction.

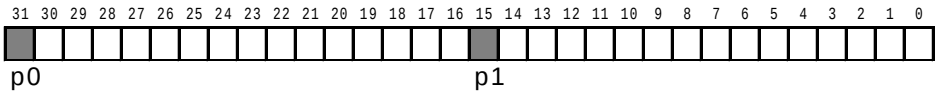


Fig 1-1 Position of P-bit

Fig 1-2 shows the instruction fields of S+core processor after the P bits are truncated. Bits 29~25 specify the main opcode in 32-bit instruction and bits 14~12 specify the main opcode in 16-bit instruction. Many instructions also have an extended sub-opcode. The remaining bits of the instruction contain one or more fields for different instruction formats.

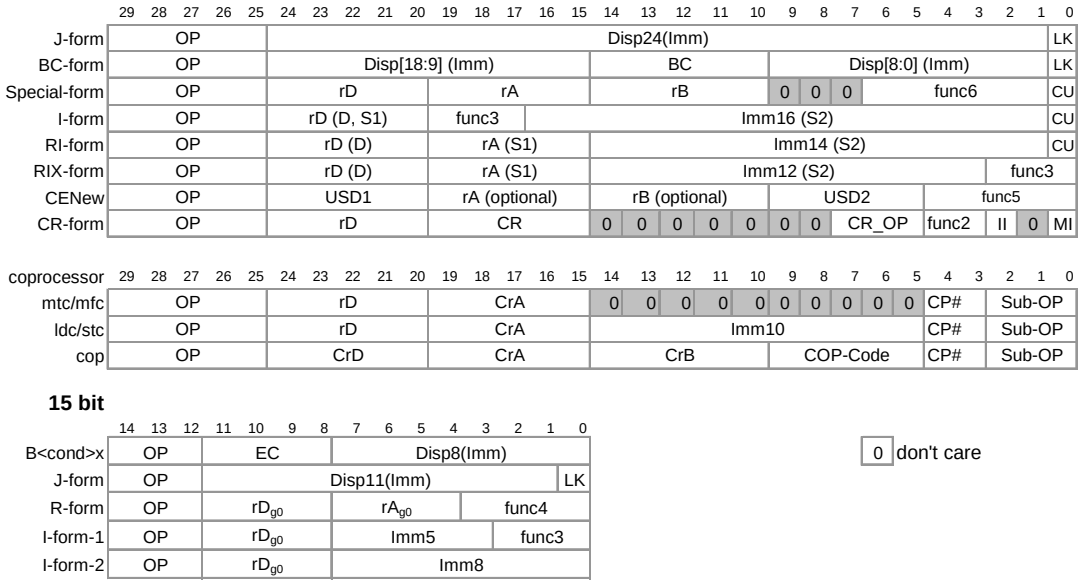


Fig 1-2 S+core Instruction Fields

1.2 Instruction Fields

Table 1-1 shows the naming convention and description of instruction fields for S+core processor.

Table 1-1 Instruction fields

Field	Naming Convention	Description
OP	Opcode	The main opcode of the instruction set.
func	Function Code	Extended function code.
CU	Condition Update	0: Don't update the condition flag 1: Update the condition flag. If the suffix ".c" is added to a 32-bit instruction, the CU bit of that instruction would be set to 1.
LK	Link	0: Don't update the link register (r3) 1: Update the link register (r3). If true, the PC of next instruction (PC+4/PC+2) would be stored into link register (r3) after a branch or a jump.
BN	Bit Number	Bit number for bit operation instructions. A 5-bit immediate that specifies the bit in rA that is to be modified by a bit operation instruction.
EC	Execution Condition	Execution condition code for conditional execution instructions. The encoding of EC field specifies the corresponding condition flag check shown in Table 2-8 (p. 85).
TC	Test Condition	Test condition code for T flag updating instructions. The 2-bit encoding of TC field specifies the corresponding condition flag check: 2'b00 specifies Z flag check; 2'b01, N flag check; 2'b10, unconditionally set T flag. The T flag is set according to the condition flag check result.
SA	Shift Amount	Shift amount for shift/rotate instructions. A 5-bit immediate that specifies the shift/rotate amount of a shift/rotate instruction.
BC	Branch Condition	Branch Condition code for conditional branch instructions. The encoding of BC field specifies the corresponding condition flag check shown in Table 2-1 (p. 26).
rD	Destination Register	Destination general-purpose register (GPR)
rA, rB	Source Register	Source general-purpose register (GPR)
g0	Lower Register	The lower registers of the general-purpose register file. Register set g0 contains r0~r15.
g1	Higher Register	The higher registers of the general-purpose register file. Register set g0 contains r16~r31.
Disp	Signed Branch Displacement	Displacement for jump and branch instructions.
Imm	Immediate	Immediate value

Field	Naming Convention	Description
CR	Control Register	
CR_OP	CR Opcode	Control register instruction opcode.
Sub-OP	Coprocessor Sub-opcode	Extended sub-opcode for distinguishing different coprocessor instructions.
USD	User-Defined Parameter	User-defined parameter field for extended custom instructions. Field specified by this notation can be either immediate or the number of a destination register (depends on user definition).
CrA, CrB	Coprocessor Source Register	
CrD	Coprocessor Destination Register	
CP#	Coprocessor Number	Represent which coprocessor instruction is used.
COP-Code	Coprocessor Opcode	Coprocessor instruction extended opcode. The coprocessor operation instruction operates according to this field.
Exp	Exponent with a radix of 2.	Represent the exponent value. This field indicates the exponent to be added in an “add exponent” instruction.
Spar (Imm)	Software Parameter	Software parameter used for trap and syscall instructions which pass it.
Srn	Special Register Number	The n^{th} special register; Sr0 is CNT; Sr1 is LCR; Sr2 is SCR.
Code		Software debug breakpoint code.
Cache_OP	Cache Opcode	The cache instruction operates according to this field.
S	Sign	This bit indicates an “add exponent” instruction (addei!) performs addition or subtraction.
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.
rD_{g0}, rD_{g1}		Destination register belonging to the register set g0 or g1.
rA_{g0}, rA_{g1}		Source register A that belongs to the register set g0 or g1.
X[n:m]		Selection of bits n through m of bit string X.

1.3 CPU Instruction Operation Notations

Symbol	Meaning	Example
k'bX	k-bit binary representation of X value	2'b11 = 3
k'hX	k-bit hexadecimal representation of X value	2'h2a = 42
X[n]	selection of bit n of bit string X	X=2'b10, X[1]=1'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
SignExtend(X)	sign-extend X to a 32-bit string	X=2'hfe, SignExtend(X) = 4'hfffe
ZeroExtend(X)	zero-extend X to a 32-bit string	X=2'hfe, ZeroExtend(X) = 4'h00fe
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 logical and	X=2'b11, Y=2'b00, X&Y = 2'b00
|	bitwise logical or	X=2'b11, Y=2'b00, X|Y =2'b11
~	bitwise logical not	X=2'b11, Y = ~X, Y=2'b00
^	bitwise logical xor	X=2'b11, Y=2'b01, X^Y = 2'b10
X^Y	X to the power of Y	10^3 = 1000
cond	*cond* is true if the icc test according to 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.
CNT	counter register	
LR	link register, generally as R3	
GPR[Rn]	Rn is a general purpose register	
C	carry flag	

2 S+core Instruction Set

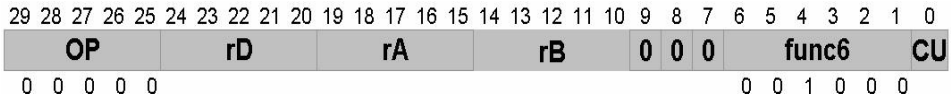
2.1 32-bit Instruction Set

The 32-bit instruction set is a high-performance instruction set that can achieve the best performance of S+core processor. It is a 3-operand operation and has the maximum immediate value.

addx

Function ADD

Form



Syntax

```
add     rD, rA, rB          (CU = 0)
add.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] + GPR[rB];
If(CU){
    N = R[31];   // R = GPR[rA] + GPR[rB]
    Z = (R==0)? 1 : 0;
    C = carry (GPR[rA] + GPR[rB]);
    V = overflow (GPR[rA] + GPR[rB]);
}
```

Usage

```
add     r4, r3, r2
add.c   r4, r3, r2
```

Description

The *addx* instruction adds the contents of general-purpose registers rA and rB, and stores the result

to general-purpose register rD.

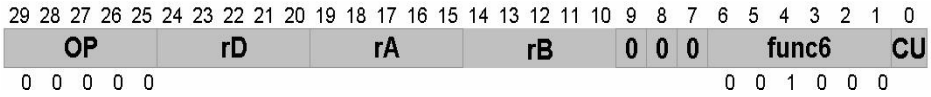
Exceptions

None

addcx

Function ADD with Carry

Form



Syntax

```
addc     rD, rA, rB          (CU = 0)
addc.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

$GPR_{rD} = GPR_{rA} + GPR_{rB} + C$

```
If (CU){
     N = R [31];   // R = GPR_rA + GPR_rB + C
     Z = (R==0)? 1: 0;
     C = carry (GPR_rA + GPR_rB + C);
     V = overflow (GPR_rA + GPR_rB + C);
}
```

Usage

```
addc     r4, r3, r2
addc.c       r4, r3, r2
```

Description

The *addcx* instruction adds the contents of general-purpose registers rA and rB with carry flag, and stores the result to general-purpose register rD.

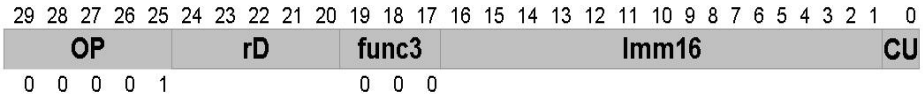
Exceptions

None

addix

Function ADD with Immediate

Form



Syntax

```
addi     rD, SImm16        (CU = 0)
addi.c   rD, SImm16        (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<SImm16>	Specifies the 16-bit signed immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] + SignExtend (Imm16);
If (CU){
     N = R [31];   // R = GPR[rA] + SignExtend (Imm16)
     Z = (R==0)? 1: 0;
     C = carry (GPR[rA] + SignExtend (Imm16));
     V = overflow (GPR[rA] + SignExtend (Imm16));
}
```

Usage

```
addi     r4, 0x1234
addi     r4, -1
```

Description

The *addix* instruction adds the content of general-purpose register rD to the sign-extended Imm16, and stores the result to general-purpose register rD.

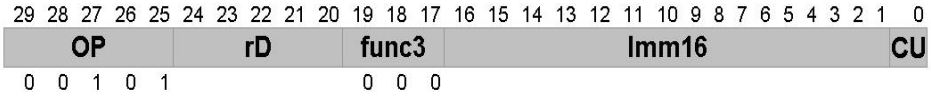
Exceptions

None

addisx

Function ADD with Immediate Shifted

Form



Syntax

```
addis    rD, Imm16              (CU = 0)
addis.c  rD, Imm16              (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies the 16-bit immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] + {Imm16, 16{0}};
If (CU){
     N = R [31];   // R = GPR[rA] + {Imm16, 16{0}}
     Z = (R==0)? 1: 0;
     C = carry (GPR[rA] + {Imm16, 16{0}});
     V = overflow (GPR[rA] + {Imm16, 16{0}});
}
```

Usage

```
addis    r4, 0x8234
```

Description

The *addisx* instruction adds the content of general-purpose register rD to the 16-bit left-shifted Imm16, and stores the result to general-purpose register rD.

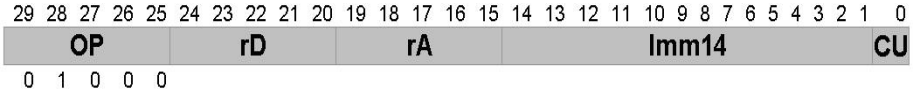
Exceptions

None

addrix

Function ADD Register with Immediate

Form



Syntax

```
addri    rD, rA, SImm14        (CU = 0)
addri.c  rD, rA, SImm14        (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SImm14>	Specifies the 14-bit signed immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] + SignExtend (Imm14);
If (CU){
     N = R [31];   // R = GPR[rA] + SignExtend (Imm14)
     Z = (R==0)? 1: 0;
     C = carry (GPR[rA] + SignExtend (Imm14));
     V = overflow (GPR[rA] + SignExtend (Imm14));
}
```

Usage

```
addri    r4, r3, 0x0123
```

Description

The *addrix* instruction adds the content of general-purpose register rA to the sign-extended Imm14, and stores the result to general-purpose register rD.

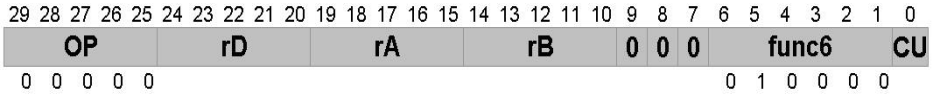
Exceptions

None

andx

Function Logical AND

Form



Syntax

```
and     rD, rA, rB          (CU = 0)
and.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] & GPR[rB];
If (CU){
     N = R [31];   // R = GPR[rA] & GPR[rB]
     Z = (R==0)? 1: 0;
}
```

Usage

```
and     r4, r3, r2
```

Description

The *andx* instruction bit-wise ands the contents of general-purpose registers rA and rB, and stores the result to general-purpose register rD.

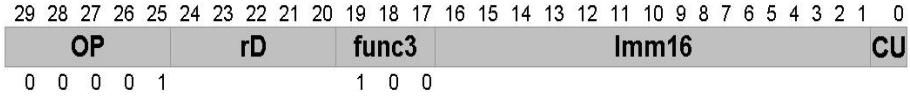
Exceptions

None

andix

Function Logical AND with Immediate

Form



Syntax

```
andi     rD, Imm16              (CU = 0)
andi.c   rD, Imm16              (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies the 16-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] & ZeroExtend(Imm16);
If(CU){
     N = R[31];   // R = GPR[rD] & ZeroExtend(Imm16)
     Z = (R==0)? 1 : 0;
}
```

Usage

```
andi     r4, 0x1234
```

Description

The *andix* instruction bit-wise ands the content of general-purpose register rD and zero-extended Imm16, and stores the result to general-purpose register rD.

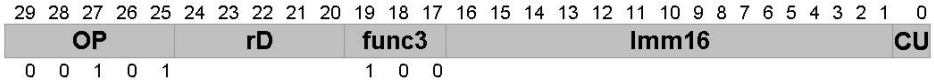
Exceptions

None

andisx

Function Logical AND with Immediate Shifted

Form



Syntax

```
andis    rD, Imm16              (CU = 0)
andis.c  rD, Imm16              (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies the 16-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] & {Imm16, 16{0}};
If(CU){
     N = R[31];   // R = GPR[rD] & {Imm16, 16{0}}
     Z = (R==0)? 1 : 0;
}
```

Usage

```
andis    r4, 0xabcd
```

Description

The *andisx* instruction bit-wise ands the content of general-purpose register rD and the 16-bit left-shifted Imm16, and stores the result to general-purpose register rD.

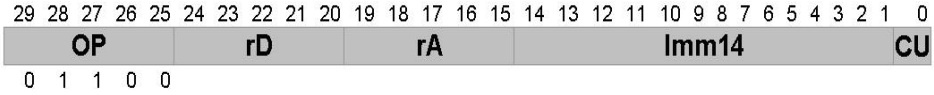
Exceptions

None

andrix

Function Logical AND Register with Immediate

Form



Syntax

```
andri    rD, rA, Imm14          (CU = 0)
andri.c  rD, rA, Imm14          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<Imm14>	Specifies the 14-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] & ZeroExtend(Imm14);
If(CU){
     N = R[31];   // R = GPR[rD] & ZeroExtend(Imm14)
     Z = (R==0)? 1 : 0;
}
```

Usage

```
andri    r4, r2, 0x0123
```

Description

The *andrix* instruction bit-wise ands the content of general-purpose register rA and zero-extended Imm14, and stores the result to general-purpose register rD.

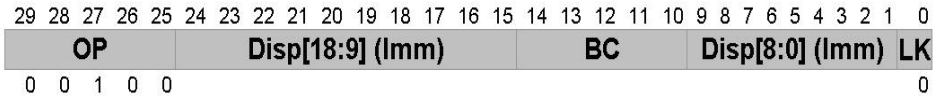
Exceptions

None

b{cond}

Function Branch Conditional

Form



Syntax

`b{cond}` `label`

`label in [-2^19 ~ 2^19 -1]` `(LK = 0)`

where

`{cond}` Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

`<label>` Specifies the branch target symbol name.

`<LK>` Specifies the link register (r3) updating bit. For *b{cond}* instruction, the LK bit is always zero.

Table 2-1 BC Field Encoding of Branch Conditional Instruction

	BC				operation	cf test	Suffix
0	0	0	0	0	branch on carry set (>=unsigned)	C	CS
1	0	0	0	1	branch on carry clear (<unsigned)	~C	CC
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

Operation

```
if(cond)
    PC = PC_current + SignExtend({Disp[18:9] | Disp[8:0], 1'b0});
else
    No operation;
```

Usage

```
beq     Label
```

Description

The *b{cond}* instruction branches to the target address according to the result of condition code test. The branch target address is the sum of the branch displacement (sign-extended 1-bit left-shifted Disp[18:9] | Disp[8:0]) and the address of current program counter.

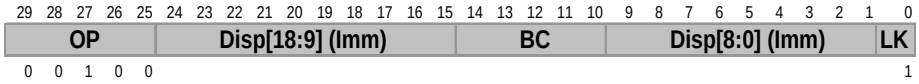
Exceptions

None

b{cond}l

Function Branch Conditional and Link

Form



Syntax

`b{cond}l` `label`

`label in [-2^19 ~ 2^19 -1]` `(LK = 1)`

where

`{cond}` Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

`<label>` Specifies the branch target symbol name.

`<LK>` Specifies the link register (r3) updating bit. For *b{cond}l* instruction, the LK bit is always one.

Operation

```
GPR_r3 = PC_current + 4;
if(cond)
  {
      PC = PC_current + SignExtend({Disp[18:9] | Disp[8:0], 1'b0});
      }
  else
      No operation;
```

Usage

```
bcsl     Label
```

Description

The *b{cond}l* instruction branches to the target address according to the result of condition code test, and saves the program counter value of the following instruction into link register. The branch target address is the sum of the branch displacement (sign-extended 1-bit left-shifted Disp[18:9] | Disp[8:0]) and the address of current program counter.

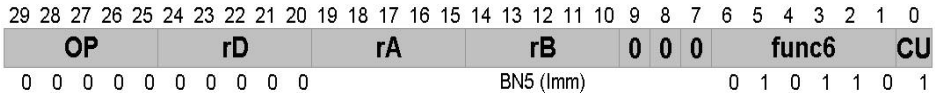
Exceptions

None

bittst.c

Function Bit Test in Register

Form



Syntax

```
bittst.c     rA, BN        (CU = 1)
```

where

<rA> Specifies the source general-purpose register.

<BN> Specifies the location of toggle bit.

<.c> Specifies that the condition flag updating bit (CU) is true. For bit operation instructions, the CU bit is always set to one and the suffix ".c" is added.

Operation

```
N = GPR_rA [31];
Z = (GPR_rA[BN]==0)? 1 : 0;      // Z = ~ GPR_rA[BN]
```

Usage

```
BITTST.c r2, 0x0a
```

Description

The *bittst.c* instruction tests the BN bit of general-purpose register rA and updates the condition flag Z with the test result. The condition flag N is also affected by this instruction.

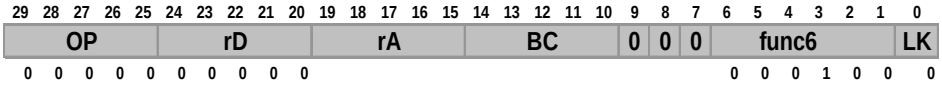
Exceptions

None

br{cond}

Function Branch Register Conditional

Form



Syntax

```
br{cond}      rA            (LK = 0)
```

where

{cond} Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

<rA> Specifies the branch target address register.

<LK> Specifies the link register (r3) updating bit. For *br{cond}* instruction, the LK bit is always zero.

Operation

```
if(cond)
    PC = GPR[rA];
else
    No operation;
```

Usage

```
breq          r3
```

Description

The *br{cond}* instruction branches to an address specified by the branch target address register, according to the result of condition code test.

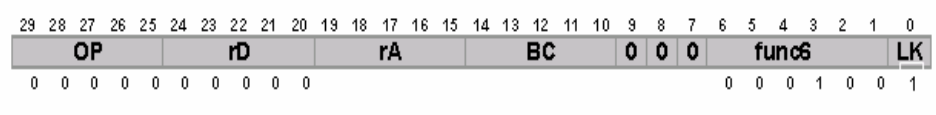
Exceptions

Instruction Address Error Exception (AdEL-instruction)

br{cond}l

Function Branch Register Conditional and Link

Form



Syntax

```
br{cond}l     rA            (LK = 1)
```

where

`{cond}` Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

`<rA>` Specifies the branch target address register.

`<LK>` Specifies the link register (r3) updating bit. For *br{cond}l* instruction, the LK bit is always one.

Operation

```
GPR[r3] = PC[current] + 4;
if(cond)
    PC = GPR[rA];
else
    No operation;
```

Usage

```
breql    r3
```

Description

The *br{cond}l* instruction branches to an address specified by the branch target address register, according to the result of condition code test; and saves the program counter of the following instruction into the link register.

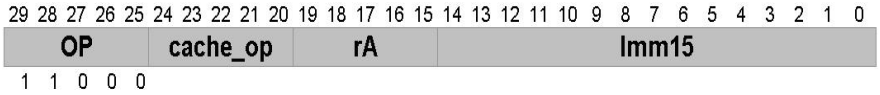
Exceptions

Instruction Address Error Exception (AdEL-instruction)

cache

Function Cache Operation Instruction

Form



Syntax

```
cache    cache_op, [rA, SImm15]
```

where

`{cache_op}` Specifies the cache operation. The encoding of this field is described in Table 2-2.

`<rA>` Specifies the base address register.

`<SImm15>` Specifies the 15-bit signed immediate value.

Table 2-2 The Encoding of Cache Operation Field

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	Set a Cache-line invalid and unlock it	VA
0x03	I-Cache	Fill LIM (local instruction memory) device	VA (PFN & Size)
0x04	I-Cache	Re-fill LIM (local instruction memory with previous PFN and Size) device	NA
0x08	D-Cache	Pre-fetch a Cache-line	VA
0x09	D-Cache	Pre-fetch and lock a Cache-line	VA
0x0A	D-Cache	Set a Cache-line invalid and unlock it	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	VA (PFN & Size)
0x0D	D-Cache	Force writing-back a Cache-line and set it valid when the cache-line is valid and dirty (Write-out)	VA
0x0E	D-Cache	Force writing-back a Cache-line and set it invalid when the cache-line is valid and dirty (Flush)	VA
0x10	I-Cache	Set entire Instruction cache invalid	NA

Cache-OP[4:0]	I-Cache/ D-Cache	Function	Data
0x11	I-Cache	Toggle Instruction Pre-fetch Buffer Function (Enable/Disable)	NA
0x18	D-Cache	Set entire Data cache invalid	NA
0x1A	D-Cache	Drain Write Buffer	NA
0x1B	D-Cache	Toggle Write Buffer Function	NA
0x1D	D-Cache	Toggle Write-back D-Cache Function (Enable/Disable)	NA
0x1E	D-Cache	Force writing-back entire D-Cache and set it valid when the cache-lines are valid and dirty (Write-out).	NA
0x1F	D-Cache	Force writing-back entire D-Cache and set it invalid when the cache-lines are valid and dirty (Flush).	NA

Operation

None

Usage

```
cache    0x00, [r2, 0x0123]
```

Description

The *cache* instruction performs cache operation according to the cache_op code. The 15-bit signed immediate value is added to the content of the base address register rA to form an effective virtual address. This virtual address is transformed to a physical one using the fixed-mapping MMU engine.

Exceptions

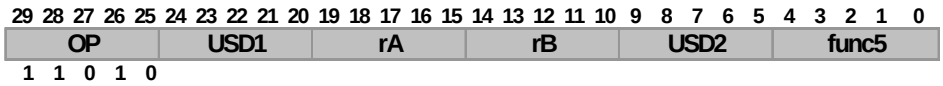
Reserved instruction exception (RI).

Bus error exception (Pre-fetch/fill Cache inst.: BusErr-Data) is a precise exception for the *cache* instruction.

ceinst

Function Custom Engine user defined Instruction

Form



Syntax

```
ceinst   fun5, rA, rB, USD1, USD2
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<USD1>	Specifies the first user-defined field.
<USD2>	Specifies the second user-defined field.

Operation

```
User-defined custom engine instruction
```

Usage

```
ceinst   0x00, r2 , r4 , 0x01, 0x02
```

Description

The *ceinst* format instruction is defined for custom engine instruction extension. In this format, func5 defines the custom engine operation. This format allows at most two source operands to be fetched from general-purpose registers. Though rA and rB fields are defined in this instruction, if one needs to specify one or two general-purpose source registers, the corresponding register indexes should be placed in these two fields. USD1 and USD2 are user-defined fields and they could be immediate for computation, parameters for operation control or destination general-purpose register index. If custom engine doesn't implement such extended instructions, reserved instruction exception will occur.

Exceptions

Reserved instruction exception (RI)

Custom engine execution exception (CeE)

cmp{TCS}.c

Function Compare

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
OP					rD					rA					rB					0	0	0	func6						CU
0	0	0	0	0	0	0	0	TCS															0	0	1	1	0	0	1

Syntax

```
cmp{TCS}.c    rA, rB        (CU = 1)
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

{TCS} Specifies the T condition flag setting bits. Table 2-3 shows the options that *{TCS}* could be, with their corresponding TCS field encoding.

Table 2-3 The TCS Field Encoding for Compare Instruction

	cond	TCS		operation
0	CMPTEQ.c	0	0	compare and if (Z=1) set T, else clear T
1	CMPTMI.c	0	1	compare and if (N=1) set T, else clear T
2		1	0	reserved (behaves like cmp.c)
3	CMP.c	1	1	compare

Operation

```
N = R[31];   // R = GPR_rA - GPR_rB
Z = (R==0)? 1 : 0;
C = ~borrow (GPR_rA - GPR_rB);
V = overflow (GPR_rA - GPR_rB);

If (TCS[1:0] == 2'b00)
    T = Z;
Else if (TCS[1:0] == 2'b01)
    T = N;
Else
    T = T;
```

Usage

```
cmp.c          r2, r3
cmptmi.c       r5, r7
```

Description

The *cmp{TCS}.c* instruction subtracts the content of general-purpose register rB from that of rA, and updates the relevant condition flags N/Z/C/V without modifying any general-purpose register. If TCS field is specified as `2'b00 or 2'b01`, the T flag will be updated accordingly.

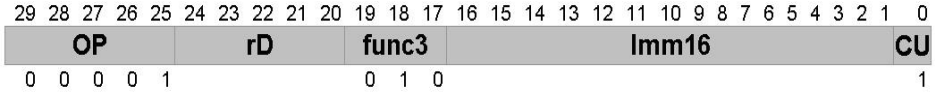
Exceptions

None

cmpi.c

Function Compare with Immediate

Form



Syntax

```
cmpi.c   rD, SImm16        (CU = 1)
```

where

<rD> Specifies the source general-purpose register.

<Simm16> Specifies the 16-bit signed immediate value.

<.c> Specifies that the condition flag updating bit CU is true.

Operation

```
N = R[31];   // R = GPR_rD - SignExtend(Imm16)
Z = (R==0)? 1 : 0;
C = ~borrow (GPR_rD - SignExtend(Imm16));
V = overflow (GPR_rD - SignExtend(Imm16));
```

Usage

```
cmpi.c       r4, 0xabcd
```

Description

The *cmpi.c* instruction subtracts the sign-extended Imm16 from the content of general-purpose register rD, and updates the relevant condition flags N/Z/C/V without modifying any general-purpose register.

Exceptions

None

cmpz{TCS}.c

Function Compare with Zero

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
OP					rD					rA					rB					0	0	0	func6						CU
0	0	0	0	0	0	0	0	TCS							0	0	0	0	0				0	0	1	1	0	1	1

Syntax

```
cmpz{TCS}.c        rA        (CU = 1)
```

where

<rA> Specifies the first source general-purpose register.

{TCS} Specifies the T condition flag setting bits. Table 2-4 shows the options that *{TCS}* could be, with their corresponding TCS field encoding.

<.c> Specifies that the condition flag updating bit CU is true.

Table 2-4 The TCS Field Encoding for Compare with Zero Instruction

	cond	TCS		operation
0	CMPZTEQ.c	0	0	compare to zero and if (Z=1) set T, else clear T
1	CMPZTMI.c	0	1	compare to zero and if (N=1) set T, else clear T
2		1	0	reserved (behaves like cmpz.c)
3	CMPZ.c	1	1	compare to zero

Operation

```
N = R[31];   // R = GPR_rA - 0
Z = (R==0)? 1 : 0;
C = ~borrow (GPR_rA - 0);
V = overflow (GPR_rA - 0);

If (TCS[1:0] == 2'b00)
     T = Z;
Else if (TCS[1:0] == 2'b01)
     T = N;
Else
     T = T;
```

Usage

```
cmpz.c             r12
cmpzteq.c          r25
```

Description

The *cmpz{TCS}.c* instruction subtracts zero from the content of general-purpose register rA, and updates the relevant condition flags N/Z/C/V without modifying any general-purpose register. If TCS field is specified as `2'b00 or 2'b01`, the T flag will be updated accordingly.

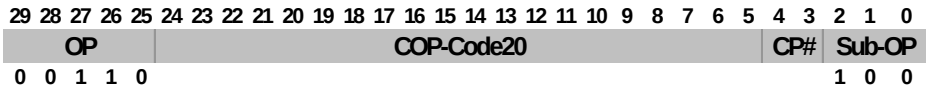
Exceptions

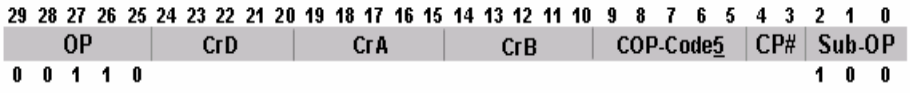
None

copx

Function Coprocessor User-defined Instruction

Form





Syntax

```
      cop1     cop_code20                (CP# = 2'b01)
      cop2     cop_code20                (CP# = 2'b10)
      cop3     cop_code20                (CP# = 2'b11)
  or
      cop1     CrD, CrA, CrB, cop_code5  (CP# = 2'b01)
      cop2     CrD, CrA, CrB, cop_code5  (CP# = 2'b10)
      cop3     CrD, CrA, CrB, cop_code5  (CP# = 2'b11)
                                         (CP# = 2'b00 is reserved)
```

where

<cop_code20>	Specifies the 20-bit coprocessor user-defined code.
<CrD, CrA, CrB>	Specifies the user-define field number.
<cop_code5>	Specifies the 5-bit coprocessor user-defined code.

Operation

```
coprocessor operation instruction
```

Usage

```
cop2     0x123
cop1     cr5,cr4,cr19,31
```

Description

This *copx* instruction is a user-defined coprocessor extension instruction. There are two types for this instruction, 20-bit parameter and 5-bit parameter forms. If the corresponding bit of CU field in processor status register (PSR) is not enabled, executing a coprocessor instruction will cause control

or coprocessor unusable (CCU) exception. The coprocessor instruction generates the reserved instruction exception when no coprocessor responds to that instruction.

Exceptions

Control or Coprocessor Unusable exception (CCU)

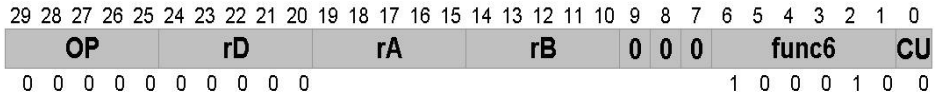
Reserved Instruction exception (RI)

Coprocessor Execution exception (CpE)

div

Function Divide

Form



Syntax

```
div     rA, rB          (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

$CEL = Q(GPR_{rA} / GPR_{rB}),$

$CEH = R(GPR_{rA} / GPR_{rB})$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
div     r2, r3
```

Description

The *div* instruction does a division on the content of general-purpose register rA. The dividend is the signed value of the content of rA and the divisor is the signed value of the content of general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The custom engine execution exception is induced when the divisor is zero. If this exception occurs, the result of this operation is undefined. The quotient word of the divided result is placed in custom engine register CEL; and the remainder word of the divided result is placed in custom engine register CEH.

The move from custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the contents of custom engine registers CEH and CEL into general-purpose registers.

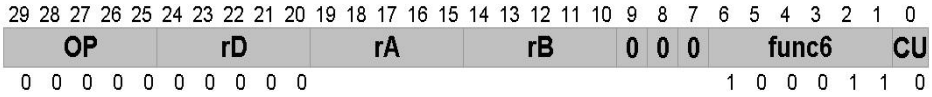
Exceptions

Custom engine execution exception: Divided by zero.

divu

Function **Divide Unsigned**

Form



Syntax

```
divu    rA, rB          (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

$CEL = Q(GPR_{rA} / GPR_{rB}),$

$CEH = R(GPR_{rA} / GPR_{rB})$

(GPR_{rA}, GPR_{rB} are treated as unsigned)

Usage

```
divu    r2, r3
```

Description

The *divu* instruction does a division on the content of general-purpose register rA. The dividend is the unsigned value of the content of rA; and the divisor is the unsigned value of the content of general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The custom engine execution exception is induced when the divisor is zero. If the result of this operation is undefined. The quotient word of the divided result is placed in custom engine register CEL, and the remainder word of the divided result is placed in custom engine register CEH.

The move from custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the contents of custom engine registers CEH and CEL into general-purpose registers.

Exceptions

Custom engine execution exception: Divided by zero.

drte

Function Return from Debug Exception

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
OP					rD					CR					0	0	0	0	0	0	0	CR_OP								
0	0	1	1	0																		1	0	1	0	0	1	0	0

Syntax

```
drte
```

Operation

```
PC = DEPC;
DM = 1'b0;
```

Usage

```
drte
```

Description

The *drte* instruction returns from debug exception by restoring the PC with DEPC, thus the program can continue from the address specified by DEPC. This instruction can only be used in Kernel mode or in User mode with PSR cra-bit set.

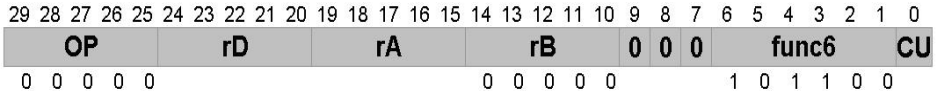
Exceptions

None

extsbx

Function Extend Sign of Byte

Form



Syntax

```
extsb    rD, rA                (CU = 0)
extsb.c  rD, rA            (CU = 1)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<.c> Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {24{ GPR[rA][7]}, GPR[rA][7:0]};
If (CU = 1){
     N = GPR[rD][31];
     Z = Undefine;
}
```

Usage

```
extsb    r4, r2
```

Description

The *extsb* instruction sign-extends the least significant byte of general-purpose register rA, and stores the result to general-purpose register rD.

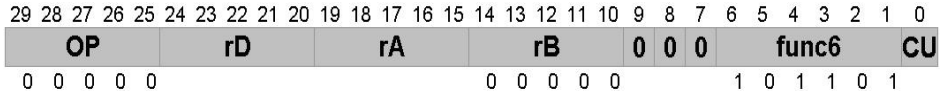
Exceptions

None

extshx

Function Extend Sign of Half-word

Form



Syntax

```
extsh    rD, rA                (CU = 0)
extsh.c  rD, rA            (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {16{ GPR[rA][15]}, GPR[rA][15:0]};
If (CU = 1){
     N = GPR[rD][31];
     Z = Undefine;
     }
```

Usage

```
extsh        r4, r2
```

Description

The *extsh* instruction sign-extends the lower half word of general-purpose register rA, and stores the result to general-purpose register rD.

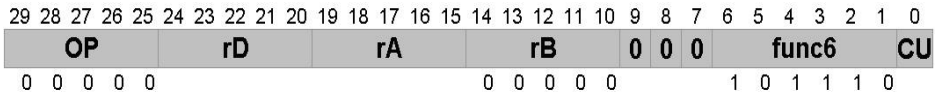
Exceptions

None

extzbx

Function Extend Zero of Byte

Form



Syntax

```
extzb    rD, rA          (CU = 0)
extzb.c  rD, rA          (CU = 1)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<.c> Specifies that the condition flag updating bit CU is true.

Operation

GPR_{rD} = {24{0}, GPR_{rA}[7:0]}

```
If (CU = 1){
    N = GPR_rD[31];
    Z = Undefine;
}
```

Usage

```
extzb    r4, r2
```

Description

The *extzb* instruction zero-extends the least significant byte of general-purpose register rA, and stores the result to general-purpose register rD.

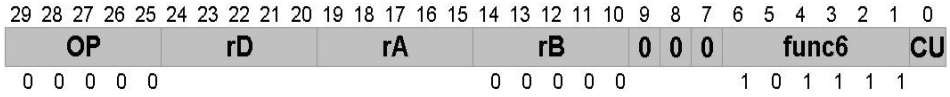
Exceptions

None

extzhx

Function Extend Zero of Half-word

Form



Syntax

```
extzh    rD, rA          (CU = 0)
extzh.c  rD, rA          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {16{0}, GPR[rA][15:0]}
If (CU = 1){
    N = GPR[rD][31];
    Z = Undefine;
}
```

Usage

```
extzh    r4, r2
```

Description

The *extzh* instruction zero-extends the lower half word of general-purpose register rA, and stores the result to general-purpose register rD.

Exceptions

None

jx

Function Jump (and Link)

Form

Syntax

```
j                                label
label  in [0 ~ 2^25 -1]      (LK = 0)
jl                               label
label  in [0 ~ 2^25 -1]      (LK = 1)
```

where

<LABEL> Specifies the symbol name of the jump target.

<LK> Specifies the link register (r3) updating bit.

Operation

```
PC = {PC_current[31:25], Disp24, 1'b0}
if(LK)
     LR = GPR_r3 = PC_current + 4;
```

Usage

```
j        Label
```

Description

The *jx* instruction jumps to a calculated target address and updates link register with the address of the following instruction according to LK bit. The target address is combined by 24-bit displacement address Disp24 shifted left by one bit and the high seven bits of the address of the jump instruction. If LK bit is set, the address of the instruction after the jump instruction is placed into the link register GPR_{r3}.

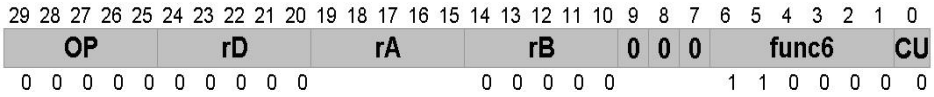
Exceptions

None

lcb

Function Load Combined Word Begin

Form



Syntax

```
lcb     [rA]+              (CU = 0)
```

where

`<rA>` Specifies the base address register.

Operation

```
LCR     = Mem32[{ GPR_rA[31:2], 2'b0}];
GPR_rA  = GPR_rA + 4;
```

Usage

```
lcb     [r2]+
```

Description

The *lcb* instruction loads word to the special register LCR (load combine register) from the memory indexed by rA. After this load operation is complete, the indexed register rA is post-incremented by 4. Since the least significant two bits of the address of rA are truncated, the address alignment error never occurs in this instruction. This instruction can also be used with *lcw* and *lce* instructions to perform un-alignment memory accesses.

Fig 2-1 shows the operation of load combine instructions when a word is given in a special register LCR and a word is given in memory respectively:

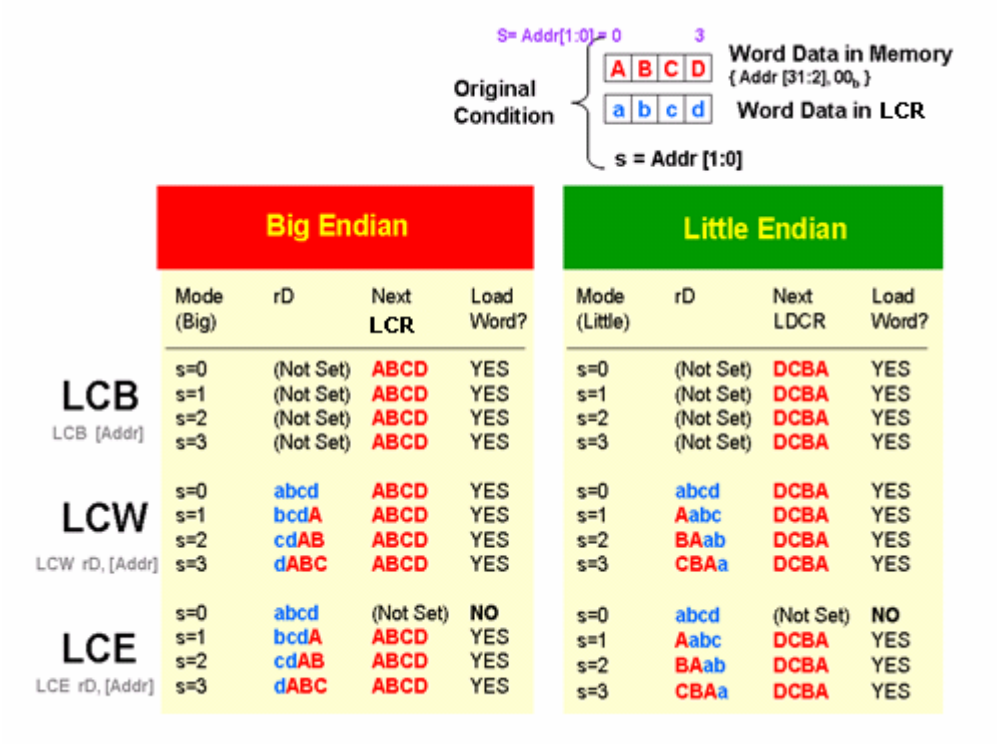


Fig 2-1 The Operation Table of Load Combine Instruction

Exceptions

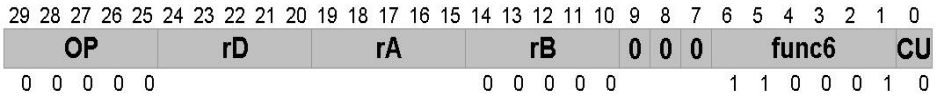
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lcw

Function Load Combined Word End

Form



Syntax

```
lcw          rD, [rA]+              (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

```
byte = GPR[rA][1:0] xor LittleEndian;
GPR[rD] = {LCR[31-8*byte:0], Mem32[{GPR[rA][31:2], 2'b0}][31: 32-8*byte]};
LCR = Mem32[{GPR[rA][31:2], 2'b0}];
GPR[rA] = GPR[rA]+4;
```

Usage

```
lcw      r4, [r2]+
```

Description

The *lcw* instruction loads word to a special register LCR (load combine register) from the memory indexed by rA. After this load operation is complete, the indexed register rA is post-incremented by 4. The loaded data is shifted and then combined with the original value of LCR register, according to processor Endian mode and the least significant two bits of the address of rA, to perform an un-alignment access. Since the least significant two bits of the address of rA are truncated, the address alignment error never occurs in this instruction. This instruction can also be used with *lcb* and *lce* instructions to implement un-alignment memory accesses. The detailed operation of *lcw* is shown in Fig 2-1.

Exceptions

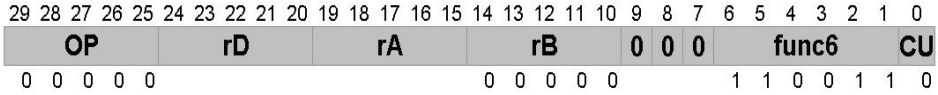
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lce

Function Load Combined Word End

Form



Syntax

```
lce     rD, [rA]+           (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

```
byte = GPR[rA][1:0] xor LittleEndian;
if(GPR[rA][1:0]==0)
{
    GPR[rD] =LCR;
    GPR[rA] = GPR[rA] + 4;
}
else
{
    GPR[rD] = {LCR[31-8*byte:0], Mem32[{GPR[rA][31:2], 2'b0}][31: 32-8*byte]};
    LCR = Mem32[{GPR[rA][31:2], 2'b0}],
}
    GPR[rA] = GPR[rA]+4
```

Usage

```
lce     r4, [r2]+
```

Description

The *lce* instruction loads word to a special register LCR (load combine register) from the memory indexed by rA. After this load operation is complete, the indexed register rA is post-incremented by 4. The loaded data is shifted and then combined with the original value of LCR register, according to processor Endian mode and the least significant two bits of the address of rA, to perform an un-alignment access. If the least significant two bits of the address of rA are zeros, the special register LCR is not changed after executing *lce* instruction. Since the least significant two bits of address of rA are truncated, the address alignment error never occurs in this instruction. This

instruction can also be used with *lcb* and *lcw* instructions to implement un-alignment memory accesses. The detailed operation of *lce* is shown in Fig 2-1.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lb

Function Load Byte signed

Form

Syntax

```
lb          rD, [rA, SImm15]
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm15> Specifies the 15-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem8[GPR_{rA} + Sign Extend(Imm15)]);

Usage

```
lb      r4, [r2, 0x0123]
```

Description

The *lb* instruction loads a byte from memory, indexed by a calculated effective memory address, sign-extends the byte and stores this result to general-purpose register rD. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm15.

Exceptions

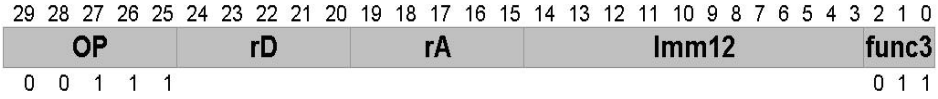
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lb (post-index)

Function Load Byte signed (post-index)

Form



Syntax

```
lb      rD, [rA]+, SImm12
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<SImm12>` Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem8[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lb      r4, [r2]+, 0x0123
```

Description

The *lb (post-index)* instruction loads a byte from memory, indexed by the content of general-purpose register rA, sign-extends the byte and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

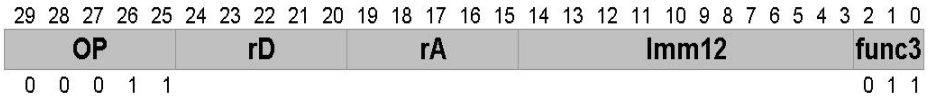
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lb (pre-index)

Function Load Byte signed (pre-index)

Form



Syntax

```
lb      rD, [rA, SImm12]+
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<SImm12>` Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem8[GPR_{rA}+SignExtend(Imm12)]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lb      r4, [r2, 0x0123]+
```

Description

The *lb (pre-index)* instruction loads a byte from memory, indexed by the sum of the content of general-purpose register rA and the sign-extended Imm12, sign-extends the byte and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

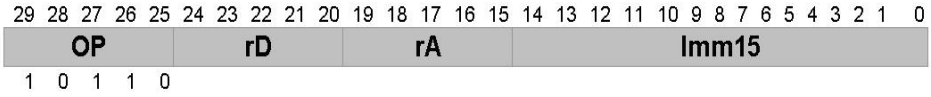
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lbu

Function Load Byte Unsigned

Form



Syntax

```
lbu             rD, [rA, SImm15]
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<SImm15>	Specifies the 15-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem8[GPR_{rA}+SignExtend(Imm15)]);

Usage

```
lbu     r4, [r2, 0x0123]
```

Description

The *lbu* instruction loads a byte from memory indexed by a calculated effective memory address, zero-extends the byte and stores this result to general-purpose register rD. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm15.

Exceptions

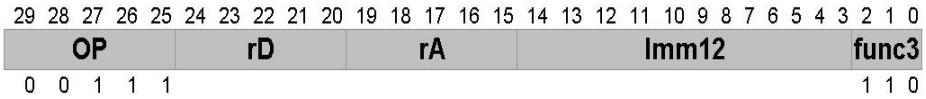
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lbu (post-index)

Function Load Byte Unsigned (post-index)

Form



Syntax

```
lbu     rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem8[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lbu     r4, [r2]+, 0x0123
```

Description

The *lbu (post-index)* instruction loads a byte from memory, indexed by the content of general-purpose register rA, zero-extends the byte and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

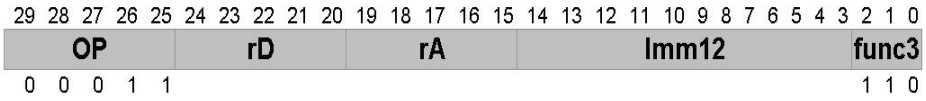
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lbu (pre-index)

Function Load Byte Unsigned (pre-index)

Form



Syntax

```
lbu      rD, [rA, SImm12]+
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<SImm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem8[GPR_{rA} + Sign Extend(Imm12)]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lbu      r4, [r2, 0x0123]+
```

Description

The *lbu (pre-index)* instruction loads a byte from memory, indexed by the sum of the content of general-purpose register rA and the sign-extended Imm12, zero-extends the byte and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

ldcx

Function Load Coprocessor data register from memory

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
OP	rD	CrA	Imm10	CP#	Sub-OP
0 0 1 1 0				0 1 0

Syntax

```
ldc1    CrA, [rD, SImm12]     (CP# = 2'b01)
ldc2    CrA, [rD, SImm12]     (CP# = 2'b10)
ldc3    CrA, [rD, SImm12]     (CP# = 2'b11)
                              (CP# = 2'b00 is reserved)
```

where

`<rD>` Specifies the base address register.

`<CrA>` Specifies the destination coprocessor (*CP#*) data register.

`<SImm12>` Specifies the 12-bit signed immediate value.

Operation

$CrA = Mem32[GPR_{rD} + Sign\ Extend(\{Imm10, 2\{0\}\})];$

Usage

```
ldc3    cr8, [r4, 0x0012]
```

Description

The *ldcx* instruction loads a word from memory, indexed by the sum of the content of general-purpose register rD and sign-extended 2-bit left-shifted Imm10, and stores this memory word to coprocessor data register CrA. If the effective address is not word aligned, address alignment error exception will occur. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

Exceptions

Reserved instruction exception

Control or coprocessor unusable exception

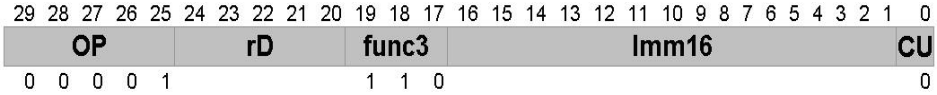
Bus error exception

Address error exception

ldi

Function Load Immediate

Form



Syntax

```
ldi       rD, SImm16        (CU = 0)
```

where

`<rD>` Specifies the destination general-purpose register.

`<SImm16>` Specifies the 16-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Imm16);

Usage

```
ldi       r4, 0xabcd
```

Description

The *ldi* instruction sign-extends the 16-bit immediate value SImm16 and stores this result into general-purpose register rD.

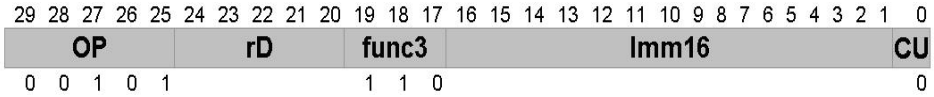
Exceptions

None

ldis

Function Load Immediate Shifted

Form



Syntax

```
ldis      rD, Imm16          (CU = 0)
```

where

`<rD>` Specifies the destination general-purpose register.

`<Imm16>` Specifies the 16-bit immediate value.

Operation

GPR_{rD} = {Imm16, 16{0}};

Usage

```
ldis      r4, 0xabcd
```

Description

The *ldis* instruction shifts the 16-bit immediate value Imm16 left by 16 bits, and stores this result into general-purpose register rD.

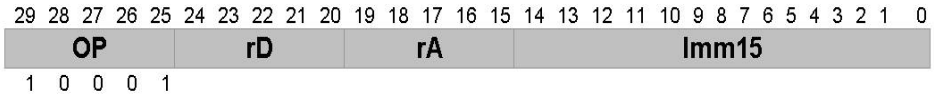
Exceptions

None

lh

Function Load Half-word signed

Form



Syntax

```
lh      rD, [rA, SImm15]
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<SImm15> Specifies the 15-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem16[GPR_{rA}+SignExtend(Imm15)]);

Usage

```
lh      r4, [r2, 0x0124]
```

Description

The *lh* instruction loads half-word from memory, indexed by a calculated effective memory address, sign-extends the half-word and stores this result to general-purpose register rD. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm15. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

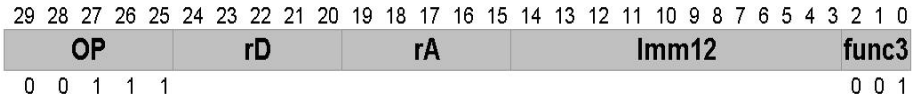
Bus error exception

Address error exception

lh (post-index)

Function Load Half-word signed (post-index)

Form



Syntax

```
lh      rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<SImm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem16[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lh      r4, [r2]+, 0x0122
```

Description

The *lh (post-index)* instruction loads half-word from memory, indexed by the content of general-purpose register rA, sign-extends the half-word and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

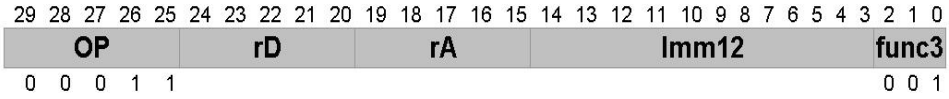
Bus error exception

Address error exception

lh (pre-index)

Function Load Half-word signed (pre-index)

Form



Syntax

```
lh      rD, [rA, SIMM12]+
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<Simm12>	Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Sign Extend(Mem16[GPR_{rA}+SignExtend(Imm12)]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lh      r4, [r2, 0x0122]+
```

Description

The *lh (pre-index)* instruction loads half-word from memory, indexed by the sum of the content of general-purpose register rA and the sign-extended Imm12, sign-extends the half-word and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

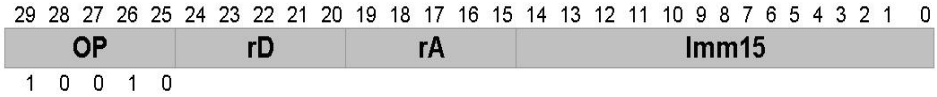
Bus error exception

Address error exception

lhu

Function Load Half-word Unsigned

Form



Syntax

```
lhu     rD, [rA, SImm15]
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<SImm15>	Specifies the 15-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem16[GPR_{rA}+SignExtend(Imm15)]);

Usage

```
lhu     r4, [r2, 0x124]
```

Description

The *lhu* instruction loads half-word from memory, indexed by a calculated effective memory address, zero-extends the half-word and stores this result to general-purpose register rD. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm15. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

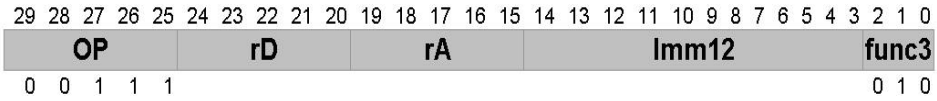
Bus error exception

Address error exception

lhu (post-index)

Function Load Half-word Unsigned (post-index)

Form



Syntax

```
lhu     rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<SImm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem16[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lhu     r4, [r2]+, 0x0122
```

Description

The *lhu (post-index)* instruction loads half-word from memory, indexed by the content of general-purpose register rA, zero-extends the half-word and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

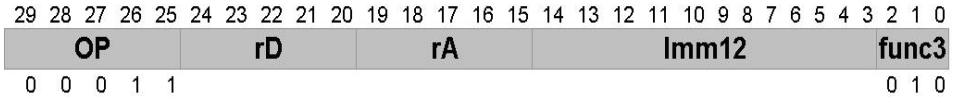
Bus error exception

Address error exception

lhu (pre-index)

Function Load Half-word Unsigned (pre-index)

Form



Syntax

```
lhu     rD, [rA, SImm12]+
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<SImm12>` Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Zero Extend(Mem16[GPR_{rA} + Sign Extend(Imm12)]);

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lhu     r4, [r2, 0x0122]+
```

Description

The *lhu (pre-index)* instruction loads half-word from memory, indexed by the sum of the content of general-purpose register rA and the sign-extended Imm12, zero-extends the half-word and stores this result to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not half-word aligned, an address alignment error exception will occur.

Exceptions

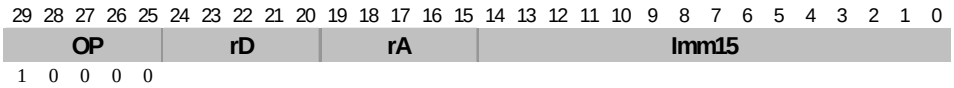
Bus error exception

Address error exception

lw

Function Load Word

Form



Syntax

```
lw      rD, [rA, SImm15]
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<Simm15>	Specifies the 15-bit signed immediate value.

Operation

GPR_{rD} = Mem32[GPR_{rA} + Sign Extend(Imm15)];

Usage

```
lw      r4, [r2, 0x0120]
```

Description

The *lw* instruction loads a word from memory, indexed by a calculated effective memory address, and stores the word to general-purpose register rD. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm15. If the load address is not word aligned, an address alignment error exception will occur.

Exceptions

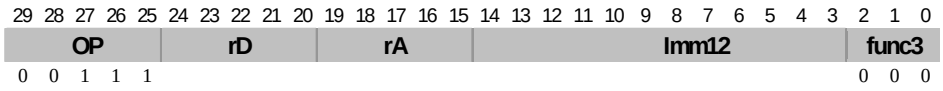
Bus error exception

Address error exception

lw (post-index)

Function Load Word (post-index)

Form



Syntax

```
lw      rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Mem32[GPR_{rA}];

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lw      r4, [r2]+, 0x0124
```

Description

The *lw (post-index)* instruction loads a word from memory, indexed by the content of general-purpose register rA, and stores the word to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not word aligned, an address alignment error exception will occur.

Exceptions

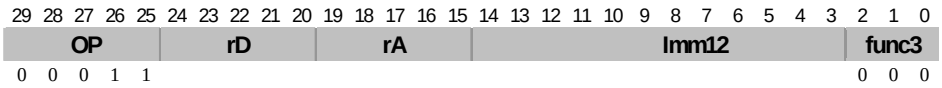
Bus error exception

Address error exception

lw (pre-index)

Function Load Word (pre-index)

Form



Syntax

```
lw      rD, [rA, SImm12]+
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm12> Specifies the 12-bit signed immediate value.

Operation

GPR_{rD} = Mem32[GPR_{rA} + Sign Extend(Imm12)];

GPR_{rA} = GPR_{rA} + Sign Extend(Imm12);

Usage

```
lw      r4, [r2, 0x0124]+
```

Description

The *lw (pre-index)* instruction loads a word from memory, indexed by the sum of the content of general-purpose register rA and the sign-extended Imm12, and stores the word to general-purpose register rD. After the load operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the load address is not word aligned, an address alignment error exception will occur.

Exceptions

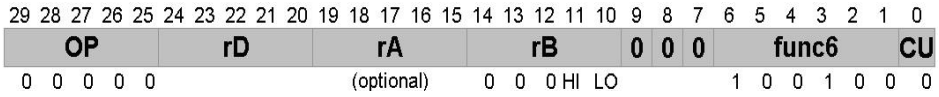
Bus error exception

Address error exception

mfcex

Function Move from Custom Engine

Form



Syntax

```
mfcel    rD          (HI = 0, LO = 1, CU = 0)
mfceh    rD          (HI = 1, LO = 0, CU = 0)
mfcehl   rD, rA      (HI = 1, LO = 1, CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the second destination general-purpose register.

Operation

```
If ({HI, LO} == 2'b01)
     GPR[rD] = CEL;
Else if ({HI, LO} == 2'b10)
     GPR[rD] = CEH;
Else if ({HI, LO} == 2'b11)
{
     GPR[rD] = CEH;
     GPR[rA] = CEL;
}
else
     Undefine;
```

Usage

```
mfcehl   r4, r2
```

Description

The *mfcex* instruction moves data from custom engine register CEH/CEL to general-purpose register according to the HI and LO bits in the instruction form. HI bit specifies the data transmission to CEH register while LO bit specifies that to CEL register. The *mfcex* instruction can transfer data from CEH and CEL simultaneously; rA is optional for this instruction. Table 2-5 lists the instruction name, HI and LO bits encoding and the corresponding operation of *mfcex* and *mtcex* instructions.

Table 2-5 The Encoding of Move to/from Custom Engine Register Instructions

HI	LO	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

Exceptions

None

mfcr

Function Move from Control Register

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
OP					rD					CR					0	0	0	0	0	0	0	CR_OP								
0	0	1	1	0																		0	0	0	0	0	0	0	1

Syntax

```
mfcr    rD, Cr_n
```

where

<rD> Specifies the destination general-purpose register.

<Cr> Specifies the source control register.

Operation

```
GPR_rD = Cr_n;
```

Usage

```
mfcr    r4, cr2
```

Description

The *mfcr* instruction moves data from control register Crn into general-purpose register rD. The encoding of field Crn is shown in Table 2-6. This instruction can only operate in Kernel mode or in User mode with cra-bit set in PSR register.

Table 2-6 The Encoding of Crn Field for *mfcr* Instruction

Encoding	Register Usage Convention	Register Description
5'b00000	*cr0*	Processor Status register (PSR)
5'b00001	*cr1*	Condition register
5'b00010	*cr2*	Exception Cause register (ECR)
5'b00011	*cr3*	Exception Vector register (EXCPvec)
5'b00100	*cr4*	Core control register (CCR)
5'b00101	*cr5*	Exception program counter (EPC)
5'b00110	*cr6*	Exception memory address (EMA)
-	-	-
5'b01111	*cr15*	LIM physical page number register (LIMPFN)
5'b10000	*cr16*	LDM physical page number register (LDMPFN)
-	-	-
5'b10010	*cr18*	Processor revision register (Prev)
-	-	-
5'b11101	*cr29*	Debug control register (DREG)
5'b11110	*cr30*	Debug exception program counter (DEPC)

Encoding	Register Usage Convention	Register Description
5'b11111	*cr31*	Debug save register (DSAVE)

Exceptions

None

mfcx

Function Move from Coprocessor Data Register

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
OP	rD	CrA	0 0 0 0 0 0 0 0 0 0	CP#	Sub-OP
0 0 1 1 0					0 0 1

Syntax

```
mfc1    rD, CrA          (CP# = 2'b01)
mfc2    rD, CrA          (CP# = 2'b10)
mfc3    rD, CrA          (CP# = 2'b11)
                         (CP# = 2'b00 is reserved)
```

where

<rD> Specifies the destination general-purpose register.

<CrA> Specifies the source coprocessor (*CP#*) data register.

Operation

GPR_{rD} = CrA

Usage

```
mfc2    r4, cr8
```

Description

The *mfcx* instruction moves data from coprocessor (*CP#*) data register into general-purpose register rD. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

Exceptions

Control or coprocessor unusable exception

mfccx

Function Move from Coprocessor Control Register

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
OP	rD	CrA	0 0 0 0 0 0 0 0 0 0	CP#	Sub-OP
0 0 1 1 0					1 1 1

Syntax

```
mfcc1   rD, CrA          (CP# = 2'b01)
mfcc2   rD, CrA          (CP# = 2'b10)
mfcc3   rD, CrA          (CP# = 2'b11)
                         (CP# = 2'b00 is reserved)
```

where

<CrA> Specifies the source coprocessor (*CP#*) control register.

Operation

GPR_{rD} = CrA

Usage

```
mfcc2    r4, cr8
```

Description

The *mfccx* instruction moves data from coprocessor (*CP#*) control register into general-purpose register rD. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

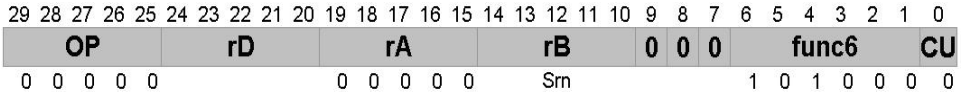
Exceptions

Control or coprocessor unusable exception

mfsr

Function Move from Special Register

Form



Syntax

```
mfsr    rD, Srn      (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<Srn> Specifies the source special register.

Operation

$GPR_{rD} = SPR_{n}$;

Usage

```
mfsr    r4, sr2
```

Description

The *mfsr* instruction moves data from special register Srn into general-purpose register rD. The encoding of field Srn is shown in Table 2-7.

Table 2-7 The Encoding of Srn Field for *mfsr* Instruction

Encoding	Register Usage Convention	Register Description
5'b00000	*sr0*	Counter register
5'b00001	*sr1*	Load combine register
5'b00010	*sr2*	Store combine register

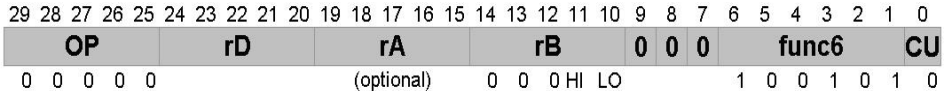
Exceptions

None

mtcex

Function Move to Custom Engine

Form



Syntax

```
mtcel   rD          (HI = 0, LO = 1, CU = 0)
mtceh   rD          (HI = 1, LO = 0, CU = 0)
mtcehl  rD, rA      (HI = 1, LO = 1, CU = 0)
```

where

<rD> Specifies the source general-purpose register.

<rA> Specifies the second source general-purpose register.

Operation

```
If ({HI, LO} == 2'b01)
    CEL = GPR[rD];
Else if ({HI, LO} == 2'b10)
    CEH = GPR[rD];
Else if ({HI, LO} == 2'b11)
{
    CEH = GPR[rD];
    CEL = GPR[rA];
}
else
    Undefine;
```

Usage

```
mtcehl   r4, r2
```

Description

The *mtcex* instruction moves data from general-purpose register to custom engine registers CEH/CEL according to the HI and LO bits in the instruction form. HI bit specifies the data transmission to CEH register while LO bit specifies that to CEL register. Table 2-5 lists the instruction name, HI and LO bits encoding and the corresponding operation of *mfcex* and *mtcex* instructions. The *mtcex* instruction can also transfer data to CEH and CEL simultaneously; rA is optional for *mtcex* instruction.

Exceptions

None

mtcr

Function Move to Control Register

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
OP					rD					CR					0	0	0	0	0	0	0	CR_OP								
0	0	1	1	0																		0	0	0	0	0	0	0	0

Syntax

```
mtcr    rD, Cr_n
```

where

<rD> Specifies the source general-purpose register.

<Cr> Specifies the destination control register.

Operation

$CR_n = GPR_{rD}$;

Usage

```
mtcr    r4, cr2
```

Description

The *mtcr* instruction moves data from general-purpose register rD into control register Crn. The encoding of field Crn is shown in Table 2-6. This instruction can only operate in Kernel mode or in User mode with cra-bit set in PSR register.

Exceptions

None

mtcx

Function Move to Coprocessor Data Register

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
OP					rD					CrA					0	0	0	0	0	0	0	0	0	0	CP#		Sub-OP			
0	0	1	1	0																							0	0	0

Syntax

```
mtc1    rD, CrA          (CP# = 2'b01)
mtc2    rD, CrA          (CP# = 2'b10)
mtc3    rD, CrA          (CP# = 2'b11)
                         (CP# = 2'b00 is reserved)
```

where

<rD> Specifies the source general-purpose register.

<CrA> Specifies the destination coprocessor (CP#) data register.

Operation

$CrA = GPR_{rD};$

Usage

```
mtc3    r14, cr15
```

Description

The *mtcx* instruction moves data from general-purpose register rD into coprocessor (*CP#*) data register. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

Exceptions

Control or coprocessor unusable exception

mtccx

Function Move to Coprocessor Control Register

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
OP	rD	CrA	0 0 0 0 0 0 0 0 0 0	CP#	Sub-OP
0 0 1 1 0					0 0 0

Syntax

```
mtcc1    rD, CrA          (CP# = 2'b01)
mtcc2    rD, CrA          (CP# = 2'b10)
mtcc3    rD, CrA          (CP# = 2'b11)
                          (CP# = 2'b00 is reserved)
```

where

`<rD>` Specifies the source general-purpose register.

`<CrA>` Specifies the destination coprocessor (CP#) control register.

Operation

$CrA = GPR_{rD};$

Usage

```
mtcc3    r14, cr15
```

Description

The *mtccx* instruction moves data from general-purpose register rD into coprocessor (*CP#*) control register. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

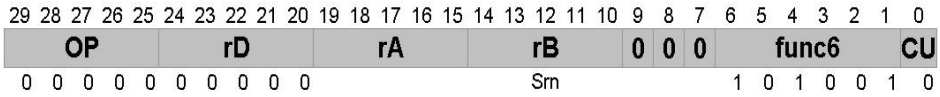
Exceptions

Control or coprocessor unusable exception

mtsr

Function Move to Special Register

Form



Syntax

```
mtsr    rA, Srn      (CU = 0)
```

where

<rD> Specifies the source general-purpose register.

<Srn> Specifies the destination special register.

Operation

$SPR_n = GPR_{rA}$;

Usage

```
mtsr    r2, sr1
```

Description

The *mtsr* instruction moves data from general-purpose register rA into special register Srn. The encoding of field Srn is shown in Table 2-7.

Exceptions

None

mv{cond}

Function Move Conditional

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
OP					rD					rA					rB					0	0	0	func6						CU
0	0	0	0	0											EC								1	0	1	0	1	1	0

Syntax

```
mv{cond} rD, rA
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

{cond} Specifies the move condition. Table 2-8 shows the 16 condition options that {*cond*} could be, with their corresponding execution condition (EC) field encoding.

Table 2-8 The Encoding of Execution Condition (EC) Field

	EC				operation	cf test	Suffix
0	0	0	0	0	execute on carry set (>=unsigned)	C	CS
1	0	0	0	1	execute on carry clear (<unsigned)	~C	CC
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 overflow	V	VS
13	1	1	0	1	execute no overflow	~V	VC
14	1	1	1	0	No operation	-	CNZ
15	1	1	1	1	set T flag always	-	AL

Operation

```
if(cond)
    GPR[rD] = GPR[rA];
else
    GPR[rD] = GPR[rD];
```

Usage

```
mveq    r4, r2
```

Description

The *mv{cond}* instruction moves data from general-purpose register rA into general-purpose register rD according to the result of condition code test. If the condition code test is true as described in Table 2-8, the data in source register rA is moved to destination register rD; otherwise, the destination register rD doesn't change. Note that *mvcnz* is not supported.

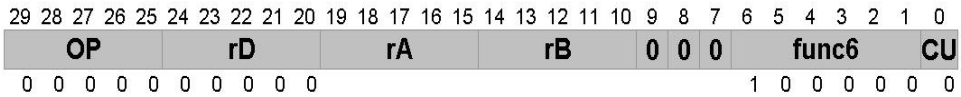
Exceptions

None

mul

Function Multiply

Form



Syntax

```
mul     rA, rB          (CU = 0)
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{HI,LO} = $GPR_{rA} * GPR_{rB}$,

GPR_{rA}, GPR_{rB} are treated as signed.

Usage

```
mul     r2, r3
```

Description

The *mul* instruction does a signed multiplication of general-purpose register rA and rB. The multiplicand is the signed value in general-purpose register rA; and the multiplier is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiplication result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

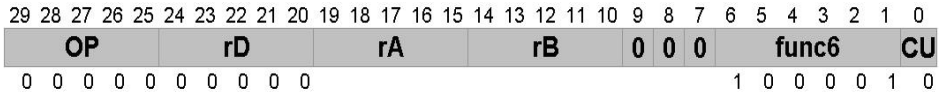
Exceptions

None

mulu

Function Multiply Unsigned

Form



Syntax

```
mulu    rA, rB          (CU = 0)
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{HI,LO\} = GPR_{rA} * GPR_{rB}$,

GPR_{rA}, GPR_{r} are treated as unsigned.

Usage

```
mulu    r2, r3
```

Description

The *mulu* instruction does an unsigned multiplication of general-purpose register rA and rB. The multiplicand is the unsigned value in general-purpose register rA while the multiplier is the unsigned value in general-purpose register rB. The low word of the multiplication result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

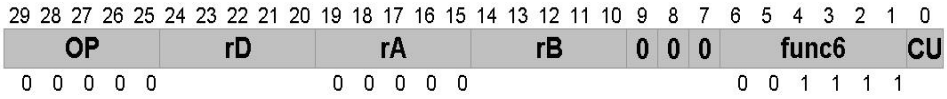
Exceptions

None

negx

Function Negative

Form



Syntax

```
neg     rD, rB          (CU = 0)
neg.c   rD, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rB>	Specifies the source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

GPR_{rD} = 0 - GPR_{rB}

```
If(CU){
    N = R[31];   // R = 0 - GPR_rB
    Z = (R==0)? 1 : 0;
    C = ~borrow (0 - GPR_rB);
    V = overflow (0 - GPR_rB);
}
```

Usage

```
neg     r4, r3
```

Description

The *negx* instruction subtracts the content of general-purpose register rB from zero to get the negative value of the content of general-purpose register rB. In other words, the negative operation performs an addition of the one's complement of register rB and one; and places the resulting 2's complement to general-purpose register rD.

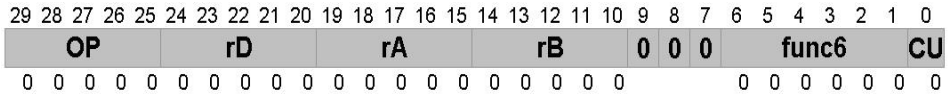
Exceptions

None

nop

Function No Operation

Form



Syntax

```
nop                (CU = 0)
```

Operation

No operation;

Usage

nop

Description

The *nop* instruction performs no operation.

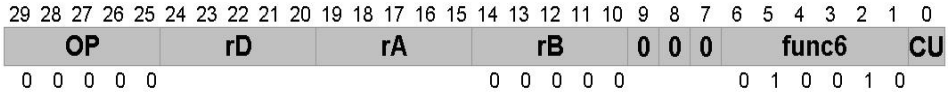
Exceptions

None

notx

Function Logical NOT

Form



Syntax

```
Not     rD, rA          (CU = 0)
not.c   rD, rA          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = ~GPR[rA];
If(CU){
    N = R[31];   // R = ~GPR[rA]
    Z = (R==0)? 1 : 0;
}
```

Usage

```
not     r4, r2
```

Description

The *notx* instruction bit-wise nots the content of general-purpose register rA, and stores the result to general-purpose register rD.

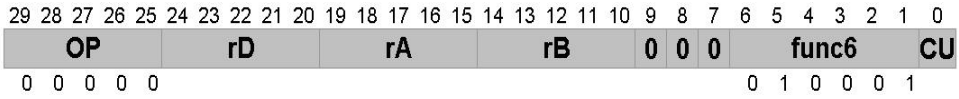
Exceptions

None

orx

Function Logical OR

Form



Syntax

```
or      rD, rA, rB              (CU = 0)
or.c    rD, rA, rB              (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] | GPR[rB];
If(CU){
     N = R[31];   // R = GPR[rA] | GPR[rB]
     Z = (R==0)? 1 : 0;
}
```

Usage

```
or   r4, r2, r3
```

Description

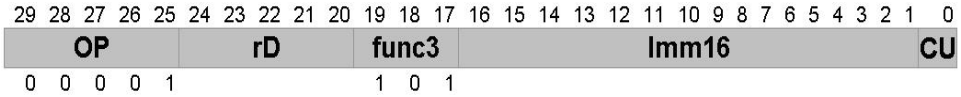
The *orx* instruction bit-wise ors the contents of general-purpose register rA and rB, and stores the result to general-purpose register rD.

Exceptions

None

orix

Function Logical OR with Immediate



Syntax

```
ori     rD, Imm16           (CU = 0)
ori.c   rD, Imm16           (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies the 16-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] | Zero Extend(Imm16);
If(CU){
    N = R[31];   // R = GPR[rD] | Zero Extend(Imm16)
    Z = (R==0)? 1 : 0;
}
```

Usage

```
ori     r4, 0x1234
```

Description

The *orix* instruction bit-wise ors the content of general-purpose register rD with the zero-extended Imm16, and stores the result to general-purpose register rD.

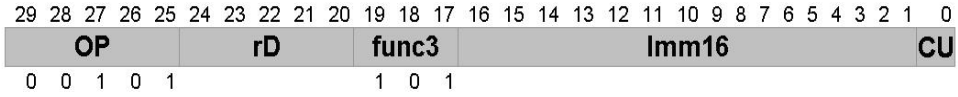
Exceptions

None

orisx

Function Logical OR with Immediate Shifted

Form



Syntax

```
oris     rD, Imm16          (CU = 0)
oris.c   rD, Imm16          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies the 16-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rD] | {Imm16, 16{0}};
If(CU){
     N = R[31];   // R = GPR[rD] | {Imm16, 16{0}}
     Z = (R==0)? 1 : 0;
}
```

Usage

```
oris     r4, 0xabcd
```

Description

The *orisx* instruction bit-wise ors the content of general-purpose register rD with the 16-bit left-shifted Imm16, and stores the result to general-purpose register rD.

Exceptions

None

orrix

Function Logical OR Register with Immediate

Form

Syntax

```
orri     rD, rA, Imm14     (CU = 0)
orri.c   rD, rA, Imm14     (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<Imm14>	Specifies the 14-bit unsigned immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] | Zero Extend(Imm14);
If(CU){
     N = R[31];   // R = GPR[rD] | Zero Extend(Imm14)
     Z = (R==0)? 1 : 0;
}
```

Usage

```
orri     r4, r2, 0x0123
```

Description

The *orrix* instruction bit-wise ors the content of general-purpose register rA with the zero-extended Imm14, and stores the result to general-purpose register rD.

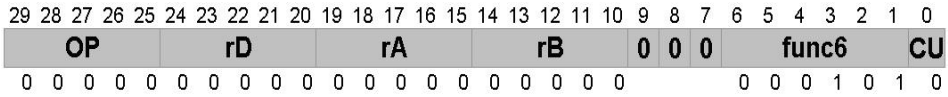
Exceptions

None

pflush

Function Pipeline Flush

Form



Syntax

```
pflush
```

Operation

```
Pipeline flush;
Restart program execution from (pflush + 4)
```

Usage

```
pflush
```

Description

In pipelined processor, some instruction sequences require software bubble insertions to prevent data hazard. For example, in the situation of a load combine instruction followed by an *mfsr* instruction with its destination LCR, three software bubbles would be required. One *pflush* instruction insertion can be applied to instead of several software bubbles to make software programming more easy.

The *pflush* instruction behaves like a jump-next instruction. It flushes the following pipeline stage and re-executes or re-fetches the next instruction from (*pflush* + *4*), like a non-pipelined processor does.

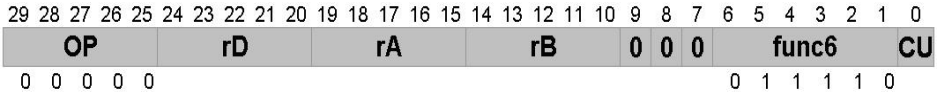
Exceptions

None

rolx

Function Rotate Left

Form



Syntax

```
rol     rD, rA, rB          (CU = 0)
rol.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register (rotate amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If ( GPR_rB[4:0] == 5'b0 )
{
        GPR_rD = GPR_rA;
        If (CU)
        N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(31-GPR_rB[4:0]):0], GPR_rA[31:(32-GPR_rB[4:0])]};
If (CU)
        {
            N = GPR_rD[31];
            C = GPR_rA[32-GPR_rB[4:0]];
        }
}
```

Usage

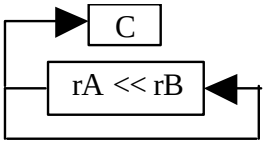
```
rol     r4, r2, r3
```

Description

The *rolx* instruction rotates left the content of general-purpose register rA by the rotate amount, the

low 5 bits of rB, and stores this result to general-purpose register rD. If CU bit is set, this instruction also updates the carry flag with the last rotated bit. Fig 2-2 shows the operation of *rolx* instruction.

Fig 2-2 The Operation of *rolx* Instruction



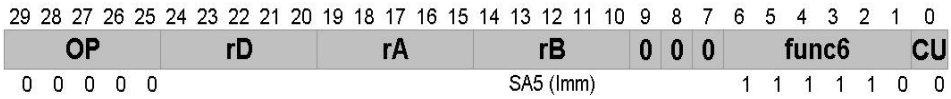
Exceptions

None

rolix

Function Rotate Left with Immediate

Form



Syntax

```
roli     rD, rA, SA          (CU = 0)
roli.c   rD, rA, SA          (CU = 0)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SA>	Specifies the rotate amount.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If ( SA == 5'b0 )
{
        GPR_rD = GPR_rA;
        If (CU)
            N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(31-SA):0], GPR_rA[31:(32-SA)]};
If (CU)
        {
            N = GPR_rD[31];
            C = GPR_rA[32-SA];
        }
}
```

Usage

```
roli     r4, r2, 0x0E
```

Description

The *rolix* instruction rotates left the content of general-purpose register rA by the rotate amount, the 5-bit SA, and stores this result to general-purpose register rD. If CU bit is set, this instruction also updates the carry flag with the last rotated bit. Fig 2-2 also shows the operation of *rolix* instruction.

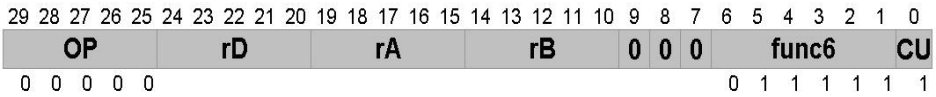
Exceptions

None

rolc.c

Function Rotate Left with Carry

Form



Syntax

```
rolc.c    rD, rA, rB            (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second general-purpose register (rotate amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If ( GPR_rB[4:0] == 5'b0 )
{
        GPR_rD = GPR_rA;
        N = GPR_rD[31];
}
Else if (GPRrB[4:0] == 5'b00001)
{
GPR_rD = {GPR_rA[30:0], C};
C = GPR_rA[31];
        N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(31-GPR_rB[4:0]):0], C, GPR_rA[31:(33-GPR_rB[4:0])]};
C = GPR_rA[32-GPR_rB[4:0]];
        N = GPR_rD[31];
}
```

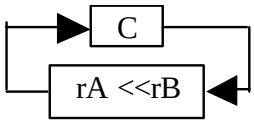
Usage

```
rolc.c    r4, r2, r5
```

Description

The *rolc.c* instruction rotates left the content of general-purpose register rA with the carry flag by the rotate amount, the low 5 bits of rB, and stores this result to general-purpose register rD. The carry flag is updated accordingly. As Fig 2-3 shows, the carry flag and register rD are in rotate chain.

Fig 2-3 The Operation of *rolc.c* Instruction



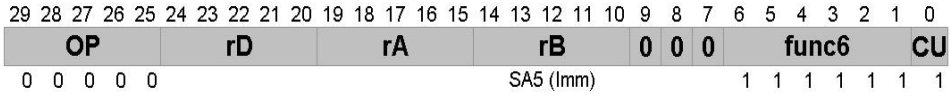
Exceptions

None

rolic.c

Function Rotate Left Immediate with Carry

Form



Syntax

```
rolic.c  rD, rA, SA        (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SA>	Specifies the rotate amount.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If (SA == 5'b0)
{
        GPR_rD = GPR_rA;
        N = GPR_rD[31];
}
Else if (SA == 5'b00001)
{
GPR_rD = {GPR_rA[30:0], C};
C = GPR_rA[31];
        N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(31-SA):0], C, GPR_rA[31:(33-SA)]};
C = GPR_rA[32-SA];
    N = GPR_rD[31];
}
```

Usage

```
rolic.c  r4, r2, 0x0E
```

Description

The *rolic.c* instruction rotates left the content of general-purpose register rA with the carry flag by the rotate amount, the 5-bit SA, and stores this result to general-purpose register rD. The carry flag is updated accordingly. As Fig 2-3 shows, the carry flag and register rD are in rotate chain.

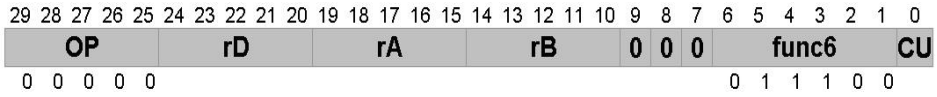
Exceptions

None

rorx

Function Rotate Right

Form



Syntax

```
ror     rD, rA, rB          (CU = 0)
ror.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second general-purpose register (rotate amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If ( GPR_rB[4:0] == 5'b0 )
{
        GPR_rD = GPR_rA;
        If (CU)
            N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(GPR_rB[4:0]-1):0], GPR_rA[31:GPR_rB[4:0]]};
If (CU)
        {
            N = GPR_rD[31];
            C = GPR_rA[(GPR_rB[4:0]-1)];
        }
}
```

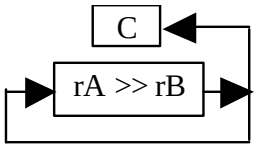
Usage

```
ror     r4, r2, r6
```

Description

The *rorx* instruction rotates right the content of general-purpose register rA by the rotate amount, the low 5 bits of rB, and stores this result to general-purpose register rD. If CU bit is set, this instruction also updates the carry flag with the last rotated bit. Fig 2-4 shows the operation of *rorx* instruction.

Fig 2-4 The Operation of *rorx* Instruction



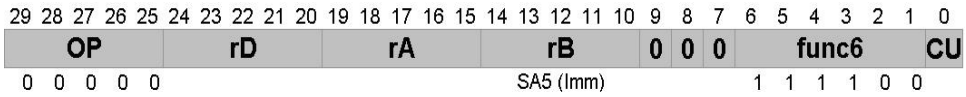
Exceptions

None

rorix

Function Rotate Right Immediate

Form



Syntax

```
rori     rD, rA, SA          (CU = 0)
rori.c   rD, rA, SA          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SA>	Specifies the rotate amount.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If (SA == 5'b0)
{
        GPR_rD = GPR_rA;
        If (CU)
            N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(SA-1):0], GPR_rA[31:SA]};
If (CU)
        {
            N = GPR_rD[31];
            C = GPR_rA[(SA-1)];
        }
}
```

Usage

```
rori     r4, r2, 0x0E
```

Description

The *rorix* instruction rotates right the content of general-purpose register rA by the rotate amount, the 5-bit SA, and stores this result to general-purpose register rD. If CU bit is set, this instruction also updates the carry flag with the last rotated bit. Fig 2-4 also shows the operation of *rorix* instruction.

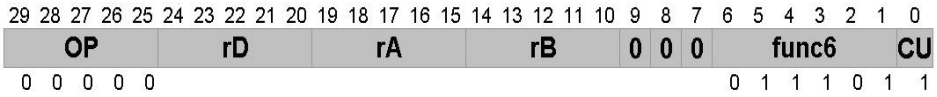
Exceptions

None

rorc.c

Function Rotate Right with Carry

Form



Syntax

```
rorc.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second general-purpose register (rotate amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If ( GPR_rB[4:0] == 5'b0 )
{
    GPR_rD = GPR_rA;
    N = GPR_rD[31];
}
Else if (GPR_rB[4:0] == 5'b00001)
{
GPR_rD = {C, GPR_rA[31:1]};
C = GPR_rA[0];
     N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(GPR_rB[4:0]-2):0], C, GPR_rA[31:GPR_rB[4:0]]};
C = GPR_rA[GPR_rB[4:0]-1];
     N = GPR_rD[31];
}
```

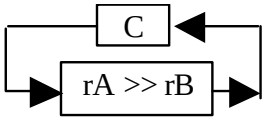
Usage

```
rorc.c   r4, r2, r4
```

Description

The *rorc.c* instruction rotates right the content of general-purpose register rA with the carry flag by the rotate amount, the low 5 bits of rB, and stores this result to general-purpose register rD. The carry flag is updated accordingly. As Fig 2-5 shows, the carry flag and register rD are in rotate chain.

Fig 2-5 The Operation of *rorc.c* Instruction



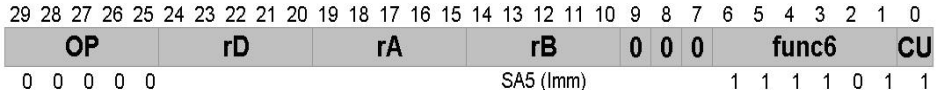
Exceptions

None

roric.c

Function Rotate Right Immediate with Carry

Form



Syntax

```
roric.c  rD, rA, SA            (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SA>	Specifies the rotate amount.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
If (SA == 5'b0)
{
        GPR_rD = GPR_rA;
        N = GPR_rD[31];
}
Else if (SA == 5'b00001)
{
GPR_rD = {C, GPR_rA[31:1]};
C = GPR_rA[0];
        N = GPR_rD[31];
}
Else
{
GPR_rD = {GPR_rA[(SA-2):0], C, GPR_rA[31:SA]};
C = GPR_rA[SA-1];
        N = GPR_rD[31];
}
```

Usage

```
roric.c  r4, r2, 0x0E
```

Description

The *roric.c* instruction rotates right the content of general-purpose register rA with the carry flag by the rotate amount, the 5-bit SA, and stores this result to general-purpose register rD. The carry flag is updated accordingly. As Fig 2-5 shows, the carry flag and register rD are in rotate chain.

Exceptions

None

rte

Function Return from Exception

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
OP					rD					CR					0	0	0	0	0	0	0	CR_OP								
0	0	1	1	0																		1	0	0	0	0	1	0	0

Syntax

```
rte
```

Operation

```
PSR.{UMs, IEs, UMc, IEc} =
     PSR.{ UMs, IEs, UMs, IEs};
Condition.{ Ts, Ns, Zs, Cs, Vs, Tc, Nc, Zc, Cc, Vc} =
Condition.{ Ts, Ns, Zs, Cs, Vs, Ts, Ns, Zs, Cs, Vs};
PC = EPC;
```

Usage

```
rte
```

Description

The *rte* instruction is used to return from interrupt or exception service routines, just like *jump register* instruction does. After executing this instruction, the program counter changes to target address pointed by exception program counter (EPC); and some fields of the processor status register (PSR) and condition register are restored to the previously saved value. The *rte* instruction can only operate in Kernel mode or in User mode with *cra*-bit set in PSR register.

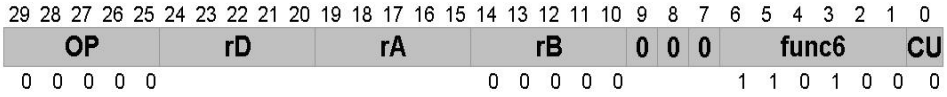
Exceptions

None

scb

Function Store Combined Word Begin

Form



Syntax

```
scb     rD, [rA]+              (CU = 0)
```

where

<rD> Specifies the source general-purpose register.

<rA> Specifies the base address register.

Operation

```
byte = GPR[rA][1:0] xor LittleEndian;
SCR = {GPR[rD][(8*byte-1):0], GPR[rD][31:8*byte]};
If (LittleEndian)
    addr[31:0] = {GPR[rA][31:2], 2'b0};
Else
    addr[31:0] = GPR[rA][31:0];

Mem[32-8*byte][addr] = GPR[rD][31:8*byte];
GPR[rA] = GPR[rA] + 4;
```

Usage

```
scb     r4, [r2]+
```

Description

The *scb* instruction stores the data in register rD and that in memory indexed by rA to a special register SCR (store combine register). After this store operation is complete, the indexed register rA is post-incremented by 4. The store size and address depend on processor Endian mode and the least significant two bits of the address of register rA. The address alignment error never occurs in this instruction. This instruction can also be used with the *scw* and *sce* instructions to implement un-alignment memory accesses.

Given a word in a special register SCR and a word in memory, the operation of store combine instructions is shown in Fig 2-6:

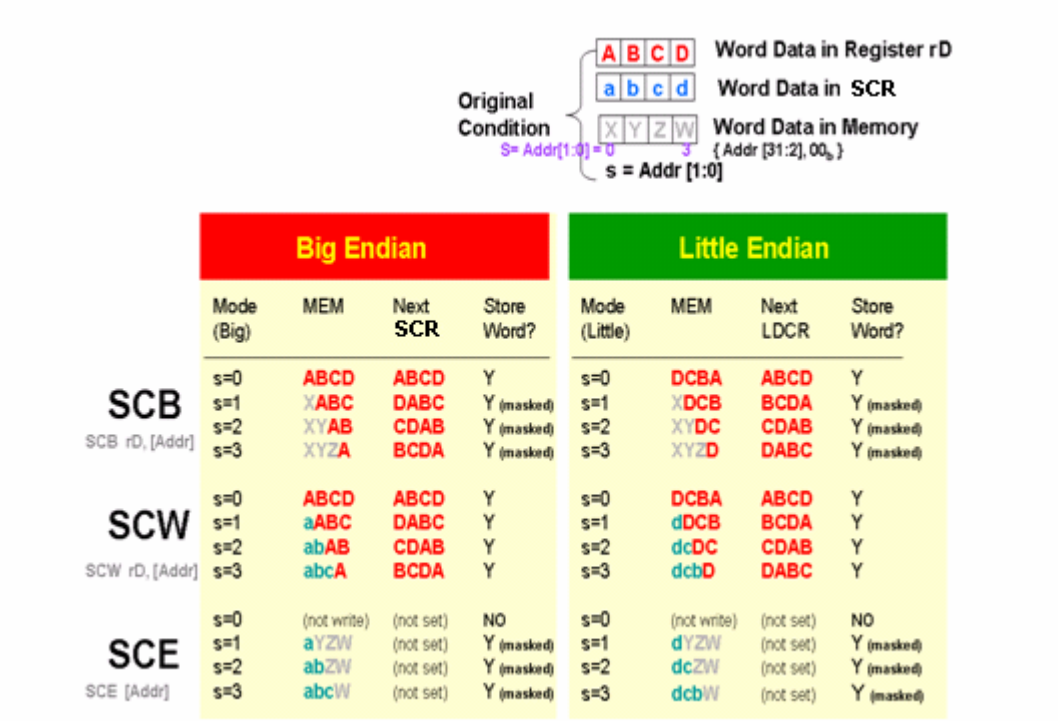


Fig 2-6 The Operation Table of Store Combine Instruction

Exceptions

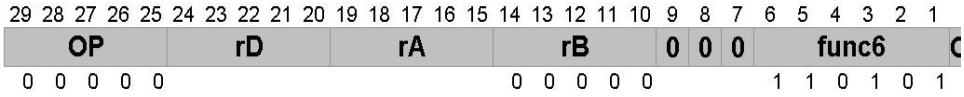
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

scw

Function Store Combined Word

Form



Syntax

```
scw     rD, [rA]+            (CU = 0)
```

where

<rD> Specifies the source general-purpose register.

<rA> Specifies the base address register.

Operation

```
byte = GPR[rA][1:0] xor LittleEndian;
Mem32[{GPR[rA][31:2],2'b0}] = {SCR[31:32-8*byte], GPR[rD][31:8*byte]};
SCR = {GPR[rD][(8*byte-1):0], GPR[rD][31:8*byte]};
GPR[rA] = GPR[rA] + 4;
```

Usage

```
scw     r4, [r2]+
```

Description

The *scw* instruction stores the combined word in special register SCR (store combine register) to the memory indexed by register rA. After this store operation is complete, the indexed register rA is post-incremented by 4. The source register rD and SCR register are combined according to processor Endian mode and the least significant two bits of the address of register rA. The store operation is always a word access. The address alignment error never occurs in this instruction. This instruction can be used with the *scb* and *sce* instructions to implement un-alignment memory accesses.

Exceptions

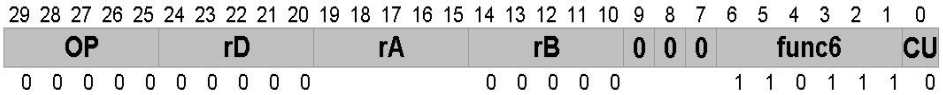
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sce

Function Store Combined Word End

Form



Syntax

```
sce     [rA]+                   (CU = 0)
```

where

<rA> Specifies the base address register.

Operation

```
byte = GPR[rA][1:0] xor LittleEndian;
    if(byte[1:0]==2'b00)
GPR[rA] = GPR[rA] + 4;
else
{
    if (LittleEndian)
addr[31:0] = {GPR[rA][31:2], (GPR[rA][1:0] + 1)};
    else
addr[31:0] = {GPR[rA][31:2], 2'b0};

Mem[8*byte][addr] = SCR[31:32-8*byte];
       GPR[rA] = GPR[rA] + 4;
    }
```

Usage

```
sce     [r2]+
```

Description

The *sce* instruction stores data in the special register SCR (store combine register) to the memory indexed by register rA. After this store operation is complete, the indexed register rA is post-incremented by 4. The store size and address depend on processor Endian mode and the least significant two bits of the address of register rA. The address alignment error never occurs in this instruction. This instruction can be used with the *scb* and *scw* instructions to implement un-alignment memory accesses.

Exceptions

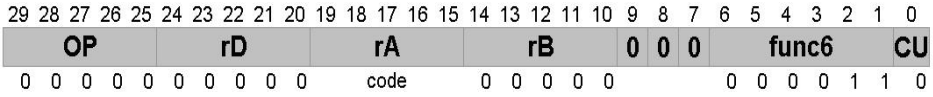
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sdbbp

Function Software Debug Breakpoint

Form



Syntax

```
sdbbp    code(Imm5)
```

where

<code> Specifies the 5-bit software parameter value.

Operation

```
if(not in Debug Mode)
{
if (ice_enable & sj_probe_en)
PC = 0xFF00_0000;
    else
        PC = {EXCPBase[31:16], 16'h1FC};
DEPC = Address of SDBBP instruction;
DM = 1'b1; DBp = 1'b1;
}
else
NOP;
```

Usage

```
sdbbp    0x0E
```

Description

The *sdbbp* instruction induces a software debug breakpoint exception, and transfers control to an exception handler. The exception vector location depends on the ICE and probe circuit. When this instruction is executed, the corresponding status bit in debug register (DREG) will be set and the debug exception program counter (DEPC) will point to the address of *sdbbp* instruction.

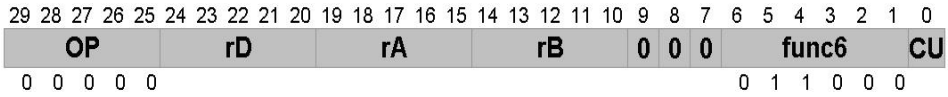
Exceptions

None

sllx

Function Shift Left Logical

Form



Syntax

```
sll     rD, rA, rB          (CU = 0)
sll.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register (shift amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {GPR[rA][(31 - GPR[rB][4:0]):0], GPR[rB][4:0]{0}};
if(CU)
{
    N = R[31];   // R = {GPR[rA][(31 -GPR[rB][4:0]):0], GPR[rB][4:0]{0}}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][32 - GPR[rB][4:0]];
}
```

Usage

```
sll     r4, r2, r3
```

Description

The *sllx* instruction left shifts the content of general-purpose register rA with zero insertion by the shift amount, the low 5 bits of register rB, and stores this result to general-purpose register rD.

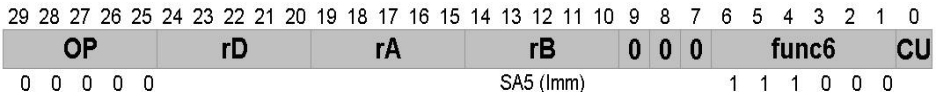
Exceptions

None

sllix

Function Shift Left Logical with Immediate

Form



Syntax

```
slli     rD, rA, SA            (CU = 0)
slli.c   rD, rA, SA            (CU = 1)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<SA> Specifies the 5-bit shift amount.

<.c> Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {GPR[rA][(31-SA):0], SA{0}};
    if(CU)
    {
        N = R[31];   // R = {GPR[rA][(31-SA):0], SA{0}}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][32 - SA];
    }
```

Usage

```
slli     r4, r2, 0x3
```

Description

The *sllix* instruction left shifts the content of general-purpose register rA with zero insertion by the shift amount, the 5-bit SA, and stores this result to general-purpose register rD.

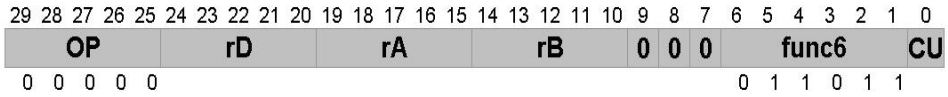
Exceptions

None

srax

Function Shift Right Arithmetic

Form



Syntax

```
sra     rD, rA, rB          (CU = 0)
sra.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register (shift amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {GPR[rB][4:0]{GPR[rA][31]}, GPR[rA][31:GPR[rB][4:0]]};
if(CU)
{
    N = R[31];   // R = {GPR[rB]{GPR[rA][31]}, GPR[rA][31:GPR[rB][4:0]]}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][GPR[rB][4:0]-1];
}
```

Usage

```
sra     r4, r2, r3
```

Description

The *srax* instruction right shifts the content of general-purpose register rA with sign insertion by the shift amount, the low 5 bits of rB, and stores this result to general-purpose register rD.

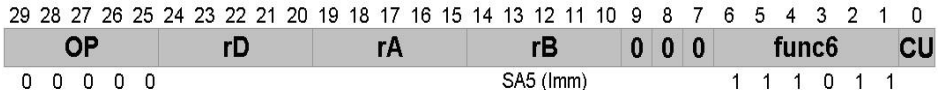
Exceptions

None

sraix

Function Shift Right Arithmetic with Immediate

Form



Syntax

```
srai     rD, rA, SA          (CU = 0)
srai.c   rD, rA, SA          (CU = 1)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<SA> Specifies the 5-bit shift amount.

<.c> Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {SA{GPR[rA][31]}, GPR[rA][31:SA]};
if(CU)
{
    N = R[31];   // R = {SA{GPR[rA][31]}, GPR[rA][31:SA]}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][SA - 1];
}
```

Usage

```
srai     r4, r2, 0x0E
```

Description

The *sraix* instruction right shifts the content of general-purpose register rA with sign insertion by the shift amount, the 5-bit SA, and stores this result to general-purpose register rD.

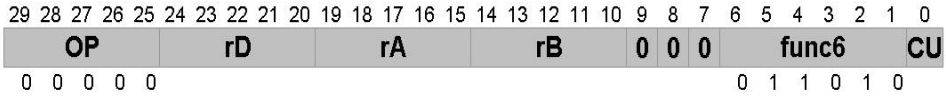
Exceptions

None

srlx

Function Shift Right Logical

Form



Syntax

```
srl     rD, rA, rB          (CU = 0)
srl.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register (shift amount).
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {GPR[rB][4:0]{0}, GPR[rA][31:GPR[rB][4:0]]};
if(CU)
{
    N = R[31];   // R = { GPR[rB] {0}, GPR[rA][31: GPR[rB][4:0]]}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][GPR[rB][4:0] - 1];
}
```

Usage

```
srl     r4, r2, r3
```

Description

The *srlx* instruction right shifts the content of general-purpose register rA with zero insertion by the shift amount, the low 5 bits of rB, and stores this result to general-purpose register rD.

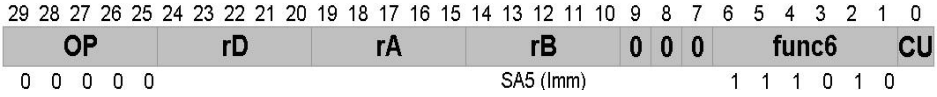
Exceptions

None

srlix

Function Shift Right Logical with Immediate

Form



Syntax

```
srli     rD, rA, SA          (CU = 0)
srli.c   rD, rA, SA          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SA>	Specifies the 5-bit shift amount.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = {SA{0}, GPR[rA][31:SA]};
if(CU)
{
    N = R[31];   // R = {SA{0}, GPR[rA][31:SA]}
    Z = (R==0)? 1 : 0;
    C = GPR[rA][SA - 1];
}
```

Usage

```
srli     r4, r2, 0x0E
```

Description

The *srlix* instruction right shifts the content of general-purpose register rA with zero insertion by the shift amount, the 5-bit SA, and stores this result to general-purpose register rD.

Exceptions

None

sb

Function Store Byte

Form

Syntax

```
sb      rD, [rA, SImm15]
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<SImm15>` Specifies the 15-bit signed immediate value.

Operation

$Mem8[GPR_{rA} + Sign\ Extend(Imm15)] = GPR_{rD}[7:0];$

Usage

```
sb      r4, [r2, 0x0123]
```

Description

The *sb* instruction stores the lowest 8 bits of general-purpose register rD to the byte memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and 15-bit signed immediate value.

Exceptions

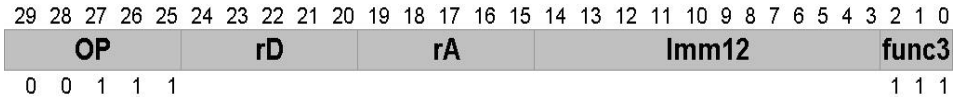
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sb (post-index)

Function Store Byte (post-index)

Form



Syntax

```
sb      rD, [rA]+, SImm12
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<Simm12>` Specifies the 12-bit signed immediate value.

Operation

$Mem8[GPR_{rA}] = GPR_{rD}[7:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sb      r4, [r2]+, 0x0123
```

Description

The *sb (post-index)* instruction stores the lowest 8 bits of general-purpose register rD to the byte memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rA. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

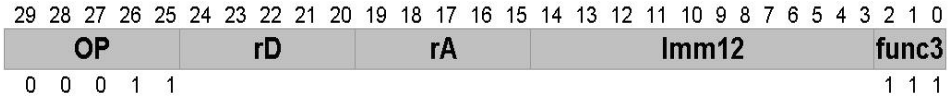
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sb (pre-index)

Function Store Byte (pre-index)

Form



Syntax

```
sb   rD, [rA, SImm12]+
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm12> Specifies the 12-bit signed immediate value.

Operation

$Mem8[GPR_{rA} + Sign\ Extend(Imm12)] = GPR_{rD}\ [7:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sb       r4, [r2, 0x0123]+
```

Description

The *sb (pre-index)* instruction stores the lowest 8 bits of general-purpose register rD to the byte memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm12. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sh

Function Store Half-word

Form

Syntax

```
sh      rD, [rA, SImm15]
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<Simm15>` Specifies the 15-bit signed immediate value.

Operation

$Mem16[GPR_{rA} + Sign\ Extend(Imm15)] = GPR_{rD}\ [15:0];$

Usage

```
sh      r4, [r2, 0x0124]
```

Description

The *sh* instruction stores the lowest 16 bits of general-purpose register rD to the half-word memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and 15-bit signed immediate value. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

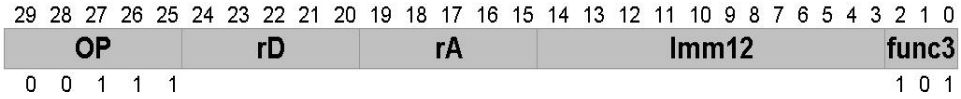
Bus error exception

Address error exception

sh (post-index)

Function Store Half-word (post-index)

Form



Syntax

```
sh      rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<SImm12> Specifies the 12-bit signed immediate value.

Operation

$Mem16[GPR_{rA}] = GPR_{rD}[15:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sh      r4, [r2]+, 0x0124
```

Description

The *sh (post-index)* instruction stores the lowest 16 bits of general-purpose register rD to the half-word memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rA. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

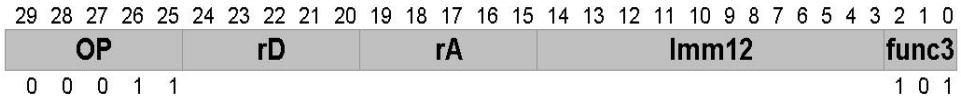
Bus error exception

Address error exception

sh (pre-index)

Function Store Half-word (pre-index)

Form



Syntax

```
sh      rD, [rA, SImm12]+
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<Simm12>` Specifies the 12-bit signed immediate value.

Operation

$Mem16[GPR_{rA} + Sign\ Extend(Imm12)] = GPR_{rD}\ [15:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sh      r4, [r2, 0x0124]+
```

Description

The *sh (pre-index)* instruction stores the lowest 16 bits of general-purpose register rD to the half-word memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm12. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

sleep

Function Sleep Instruction

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
OP					rD					CR					0	0	0	0	0	0	0	CR_OP								
0	0	1	1	0																		1	1	0	0	0	1	0	0

Syntax

```
sleep
```

Operation

```
Drive processor core into power saving standby mode
```

Usage

```
sleep
```

Description

The *sleep* instruction drives the processor core into a power saving standby mode. This instruction holds the processor clock in the low state until an external interrupt, a non-maskable interrupt, or a debug interrupt is received. Before executing the *sleep* instruction, programmers must ensure that the interrupt condition has been enabled via the IE field in PSR and IM field in ECR. This instruction can only operate in Kernel mode or in User mode with *cra*-bit set in PSR register.

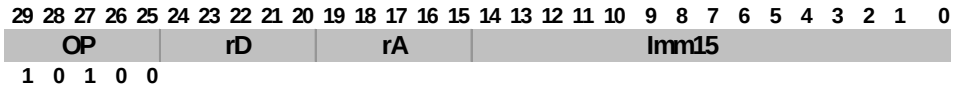
Exceptions

Control or coprocessor unusable exception (CCU)

sw

Function Store Word

Form



Syntax

```
sw      rD, [rA, SImm15]
```

where

`<rD>` Specifies the destination general-purpose register.

`<rA>` Specifies the base address register.

`<Simm15>` Specifies the 15-bit signed immediate value.

Operation

$Mem32[GPR_{rA} + Sign\ Extend(Imm15)] = GPR_{rD}\ [31:0];$

Usage

```
sw      r4, [r2, 0x0124]
```

Description

The *sw* instruction stores the content of general-purpose register rD to the word memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and 15-bit signed immediate value. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

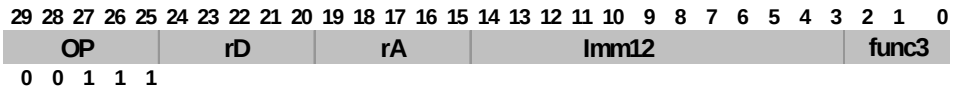
Bus error exception

Address error exception

sw (post-index)

Function Store Word (post-index)

Form



Syntax

```
sw      rD, [rA]+, SImm12
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<Simm12> Specifies the 12-bit signed immediate value.

Operation

$Mem32[GPR_{rA}] = GPR_{rD}[31:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sw      r4, [r2]+, 0x0124
```

Description

The *sw (post-index)* instruction stores the content of general-purpose register rD to the word memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rA. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

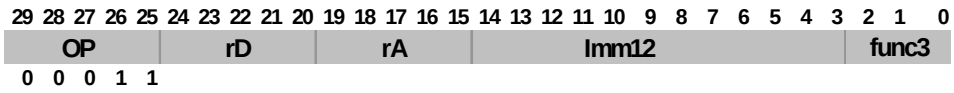
Bus error exception

Address error exception

sw

Function Store Word (pre-index)

Form



Syntax

```
sw      rD, [rA, SIMM12]+
```

Operation

$Mem32[GPR_{rA} + Sign\ Extend(Imm12)] = GPR_{rD}\ [31:0];$

$GPR_{rA} = GPR_{rA} + Sign\ Extend(Imm12);$

Usage

```
sw      r4, [r2, 0x0124]+
```

Description

The *sw (pre-index)* instruction stores the content of general-purpose register rD to the word memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rA and the sign-extended Imm12. After this store operation is complete, the content of general-purpose register rA is updated with the sum of the content of general-purpose register rA and the sign-extended Imm12. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

stcx

Function Store Coprocessor Data Register to Memory

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
OP	rD	CrA	Imm10	CP#	Sub-OP
0 0 1 1 0					0 1 1

Syntax

```
stc1    CrA, [rD, SImm12]     (CP# = 2'b01)
stc2    CrA, [rD, SImm12]     (CP# = 2'b10)
stc3    CrA, [rD, SImm12]     (CP# = 2'b11)
                              (CP# = 2'b00 is reserved)
```

where

`<rD>` Specifies the base address register.

`<CrA>` Specifies the source coprocessor (*CP#*) data register.

`<Simm12>` Specifies the 12-bit signed immediate value.

Operation

$Mem32[GPR_{rD}+SignExtend(\{Imm10, 2\{0\}\})] = CrA;$

Usage

```
stc1    cr8, [r4, 0x0012]
```

Description

The *stcx* instruction stores the content of coprocessor data register CrA to the word memory indexed by the effective memory address. The effective memory address is the sum of the content of general-purpose register rD and the sign-extended 2-bit left-shifted Imm10. If the effective address is not word aligned, an address alignment error exception will occur. If the corresponding bit of CU field in processor status register (PSR) is not set, executing a coprocessor instruction will cause control or coprocessor unusable (CCU) exception.

Exceptions

Control or coprocessor unusable exception

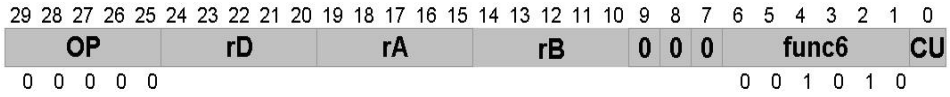
Bus error exception

Address error exception

subx

Function Subtract

Form



Syntax

```
sub     rD, rA, rB              (CU = 0)
sub.c   rD, rA, rB              (CU = 0)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] - GPR[rB];
If (CU)
{
    N = R[31];   // R = GPR[rA] - GPR[rB]
    Z = (R==0)? 1 : 0;
    C = ~borrow (GPR[rA] - GPR[rB]);
    V = overflow (GPR[rA] - GPR[rB]);
}
```

Usage

```
sub     r4, r2, r3
```

Description

The *subx* instruction subtracts the content of general-purpose register rB from that of rA, and updates general-purpose register rD with this subtraction result.

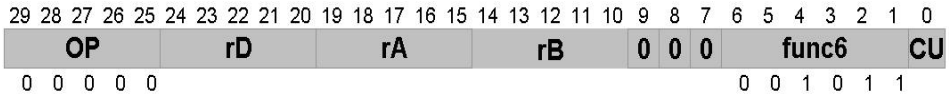
Exceptions

None

subcx

Function Subtract with Carry

Form



Syntax

```
subc     rD, rA, rB          (CU = 0)
subc.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] - GPR[rB] - (~C)
If (CU)
{
    N = R[31];   // R = GPR[rA] - GPR[rB] - (~C)
    Z = (R==0)? 1 : 0;
    C = ~borrow (GPR[rA] - GPR[rB] - (~C));
    V = overflow (GPR[rA] - GPR[rB] - (~C));
}
```

Usage

```
subc     r4, r2, r3
```

Description

The *subcx* instruction subtracts the content of general-purpose register rB and the carry flag from the content of rA, and updates general-purpose register rD with this subtraction result.

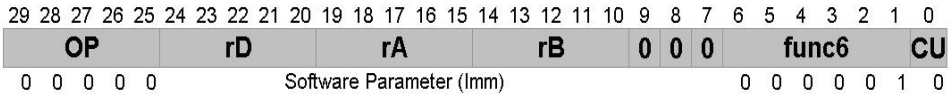
Exceptions

None

syscall

Function System Call Trap

Form



Syntax

```
Syscall       Software_Parameter(Imm15)       (CU = 0)
```

where

`<software_parameter>` Specifies the 15-bit software parameter.

Operation

```
System call trap
```

Usage

```
syscall  0x008E
```

Description

The *`syscall`* instruction induces a syscall exception unconditionally and transfers control to the exception handler. The program registers in the processor core are unchanged when this exception occurs except the following:

- The bit fields IEc and KUc in Processor Status register (PSR) are saved.
- The bit fields Tc, Nc, Zc, Cc, and Vc in Condition register are saved.
- The Exc_code in Exception Cause register (ECR) is 7.
- The EPC register points at the *`syscall`* instruction.

Exceptions

None

t{cond}

Function Test and Set Condition Flag T

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
OP					rD					rA					rB					0	0	0	func6						CU
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	EC								1	0	1	0	1	0	0

Syntax

`t{cond}` (CU = 0)

where

`t` Specifies the condition flag T. T flag is used in parallel condition execution to determine whether the execution is through the true or false path.

`{cond}` Specifies the test condition. Table 2-9 shows the 16 condition options that {*cond*} could be, with their corresponding execution condition (EC) field encoding.

Table 2-9 The EC Field Encoding for Test Condition

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

Operation

```
case(EC)
begin
    4'b0000: T = C;
    4'b0001: T = ~C;
    4'b0010: T = C & ~Z;
    4'b0011: T = ~C | Z;
    4'b0100: T = Z;
    4'b0101: T = ~Z;
```

```
        4'b0110: T = (Z==0) & (N==V);
        4'b0111: T = (Z==1) | (N!=V);
        4'b1000: T = (N==V);
        4'b1001: T = (N!=V);
        4'b1010: T = N;
        4'b1011: T = ~N;
        4'b1100: T = V;
        4'b1101: T = ~V;
        4'b1110: T = (CNT>0);
        4'b1111: T = 1;
    end
```

Usage

```
tvs
tset
tcnz
```

Description

The *t{cond}* instruction sets or clears T flag according to the result of condition flag test. The EC field specifies the corresponding condition flag test.

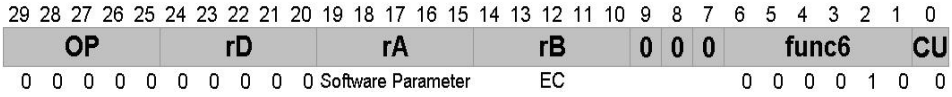
Exceptions

None

trap{cond}

Function Trap Conditional

Form



Syntax

```
trap{cond}    Software_Parameter (Imm5)       (CU = 0)
```

where

`<software_parameter>` Specifies the 5-bit software parameter.

`{cond}` Specifies the trap exception condition. Table 2-10 shows the 15 condition options that {*cond*} could be, with their corresponding execution condition (EC) field encoding.

Table 2-10 The EC Field Encoding for Trap Instruction

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

Operation

```
if(cond)
    trap exception;
else
    nop;
```

Usage

```
trapeq        0x008E
```

Description

This is a control instruction which performs conditional trap exception operation. It can only operate in Kernel mode or in User mode with cra-bit set in PSR register.

The *trap{cond}* instruction causes trap exception according to the result of condition code test. When the test result is true, this instruction transfers control to the exception handler. The program registers in the processor core are unchanged when this exception occurs except the following:

- The bit fields IEc and KUc in Processor Status register (PSR) are saved.
- The bit fields Tc, Nc, Zc, Cc, and Vc in Condition register are saved.
- The Exc_code in Exception Cause register (ECR) is 10.
- The EPC register points at the *trap{cond}* instruction.

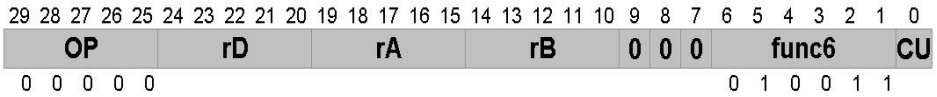
Exceptions

None

xorx

Function Logical XOR

Form



Syntax

```
xor     rD, rA, rB          (CU = 0)
xor.c   rD, rA, rB          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

```
GPR[rD] = GPR[rA] ^ GPR[rB];
If(CU){
     N = R[31];   // R = GPR[rA] ^ GPR[rB]
     Z = (R==0)? 1 : 0;
}
```

Usage

```
xor     r4, r2, r3
```

Description

The *xorx* instruction bit-wise xors the contents of general-purpose register rA and rB, and updates general-purpose register rD with this result.

Exceptions

None

2.2 16-Bit Instruction Set

The 16-bit instruction set (including PCE) is not recommended in hand coding assembly, and it is reserved for compiler for hybrid instruction mode to reduce the code size. Restricted to the instruction width, 16-bit instruction set is a 2-operand operation and has small immediate value. If the last 16-bit instruction is not word aligned, it is better to change this 16-bit instruction into the corresponding 32-bit instruction and not to insert a 16-bit *nop!* instruction to align word boundary, since this instruction increases the execution instruction count.

add!

Function Add

Form

Syntax

```
add!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and first source general-purpose registers (r0~r15).

<rAg0> Specifies the second source general-purpose register (r0~r15).

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
GPR[rD]g0 = GPR[rD]g0 + GPR[rA]g0;
```

Usage

```
add!    r4, r2
```

Description

The *add!* instruction adds up the contents of general-purpose register rDg0 and rAg0, and stores this result to general-purpose register rDg0. All the condition flags (N, Z, C, V) are clobbered by this instruction.

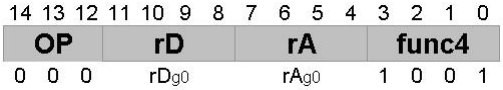
Exceptions

None

addc!

Function ADD with Carry

Form



Syntax

```
addc!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and first source general-purpose registers (r0~r15).

<rAg0> Specifies the second source general-purpose register (r0~r15).

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
GPR[rD]g0 = GPR[rD]g0 + GPR[rA]g0 + C;
```

Usage

```
addc!    r15, r7
```

Description

The *addc!* instruction adds up the contents of general-purpose register rDg0 and rAg0 with the carry flag, and stores this result to general-purpose register rDg0. All the condition flags (N, Z, C, V) are clobbered by this instruction.

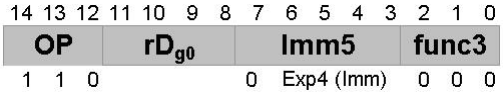
Exceptions

None

addei!

Function ADD with Exponent Immediate

Form



Syntax

```
addei!   rDg0, Imm4        (CU = 1)
```

where

<Rdg0> Specifies the destination and first source general-purpose registers (r0~r15).

<Imm4> Specifies the 4-bit exponent immediate value.

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
```

$GPR_{rD}g0 = GPR_{rD}g0 + 2^{Exp4}$;

Usage

```
addei!   r9, 1
```

Description

The *addei!* instruction adds the content of general-purpose register rDg0 to the exponent immediate 2^{Exp4}, and stores this result to general-purpose register rDg0. All the condition flags (N, Z, C, V) are clobbered by this instruction.

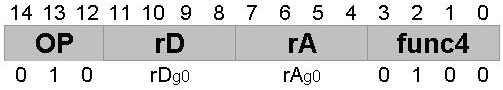
Exceptions

None

and!

Function Logical AND

Form



Syntax

```
AND!    rDg0, rAg0        (CU = 1)
```

where

<rDg0> Specifies the destination and first source general-purpose registers (r0~r15).

<rAg0> Specifies the second source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
GPRrDg0 = GPRrDg0 & GPRrAg0;
```

Usage

```
and!    r14, r2
```

Description

The *and!* instruction bit-wise ands the contents of general-purpose register rDg0 and rAg0, and stores this result to general-purpose register rDg0. The condition flags (N, Z) are always clobbered by this instruction.

Exceptions

None

b{cond}!

Function Branch Conditional

Form

Syntax

`b{cond}! label`

`label in [-2^8 ~ 2^8 -1]`

where

{cond} Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

`<label>` Specifies the branch target symbol name.

Operation

```
if (cond)
    PC = PC_current + Sign Extend({Disp8, 1'b0});
else
    NOP;
```

Usage

```
beq!    Label
```

Description

The *b{cond}!* instruction branches to the target address according to the result of condition code test. The branch target address is the sum of the branch displacement, sign-extended 1-bit left-shifted Disp8, and the address of current program counter.

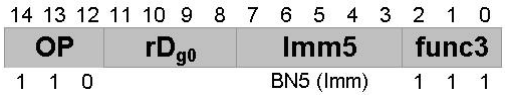
Exceptions

None

bittst!

Function Bit Test in Register

Form



Syntax

```
bittst!  rDg0, BN      (CU = 1)
```

where

<rDg0> Specifies the destination and first source general-purpose registers (r0~r15).

<BN> Specifies the location of test bit.

Operation

```
N = GPRrDg0 [31];
Z = (GPRrDg0 [BN]==0)? 1 : 0;        // Z = ~ GPRrDg0 [BN]
```

Usage

```
bittst!  r7, 0x0E
```

Description

The *bittst!* instruction tests the BN-bit of general-purpose register rDg0 and updates the condition flag Z according to the test result . The condition flag N is also affected by this instruction.

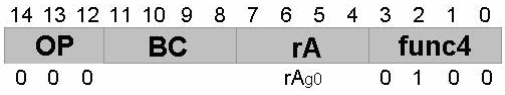
Exceptions

None

br{cond}!

Function Branch Register Conditional

Form



Syntax

`br{cond}!rAg0`

where

`{cond}` Specifies the branch condition. Table 2-1 shows the 16 condition options that {*cond*} could be, with their corresponding branch condition (BC) field encoding.

`<rAg0>` Specifies the branch target address register.

Operation

```
if (cond)
    PC = GPRrAg0;
else
    NOP;
```

Usage

```
brcc!    r12
```

Description

The *br{cond}!* instruction branches to an address specified by the branch target address register rAg0, according to the result of condition code test.

Exceptions

Instruction Address Error Exception (AdEL-instruction)

cmp!

Function Compare

Form

Syntax

```
cmp!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the source general-purpose register (r0~r15).

<rAg0> Specifies the second source general-purpose register (r0~r15).

Operation

```
N = R[31];   // R = GPR_rDg0 - GPR_rAg0
Z = (R==0)? 1 : 0;
C = ~borrow (GPR_rDg0 - GPR_rAg0);
V = overflow (GPR_rDg0 - GPR_rAg0);
```

Usage

```
cmp!    r6, r2
```

Description

The *cmp!* instruction subtracts the content of general-purpose register rAg0 from that of rDg0, and updates the condition flags N/Z/C/V accordingly without modifying any general-purpose register.

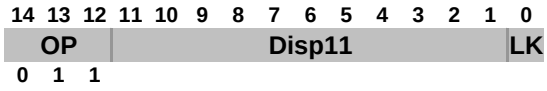
Exceptions

None

jx!

Function Jump (and Link)

Form



Syntax

```
j!      label           (LK = 0)
label  in [0 ~ 2^12 -1]
jl!     label           (LK = 1)
label  in [0 ~ 2^12 -1]
```

where

<label> Specifies the 12-bit jump label displacement.

<LK> Specifies the link register updating bit.

Operation

```
PC = {PCcurrent[31:12], Disp11, 1'b0}
if(LK)
    LR = GPR_r3 = PC_current + 4;
```

Usage

```
j!      Label
```

Description

The *jx!* instruction jumps to a calculated target address and updates link register with the address of the following instruction according to LK bit. The jump target address is formed by shifting left the 11-bit displacement address Disp11 by one bit and then combining the address with the high 20 bits of the address of the jump instruction. If LK bit is set, the address of the instruction after the jump instruction is placed into the link register GPR_{r3}.

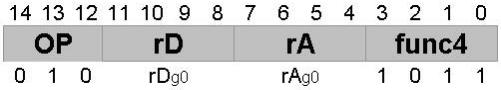
Exceptions

None

lbu!

Function Load Byte Unsigned

Form



Syntax

```
lbu!    rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$GPR_{rD}g0$ = Zero Extend(Mem8[$GPR_{rA}g0$]);

Usage

```
lbu!    r4, [r13]
```

Description

The *lbu!* instruction loads a byte from memory indexed by the content of general-purpose register rAg0, zero-extends the byte and stores this result to general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lbup!

Function Load Byte Unsigned with Base Pointer

Form

Syntax

```
lbup!    rDg0, Imm5
```

where

<Rdg0> Specifies the destination general-purpose register (r0~r15).

<Imm5> Specifies the 5-bit immediate value.

Operation

```
GPRrDg0 = Zero Extend(Mem8[BP+ZeroExtend(Imm5)]);
```

Usage

```
lbup!    R15, 0x0E
```

Description

The *lbup!* instruction loads a byte from memory indexed by the sum of base pointer register r2 and the zero-extended Imm5, zero-extends the byte and stores this result to general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

lh!

Function Load Half-word Signed

Form

Syntax

```
lh!     rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$GPR_{rD}g0 = Sign\ Extend(Mem16[GPR_{rA}g0]);$

Usage

```
lh!     r13, [r5]
```

Description

The *lh!* instruction loads half-word from memory indexed by the content of general-purpose register rAg0, sign-extends the half-word and stores this result to general-purpose register rDg0. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

lhp!

Function Load Half-word Signed with Base Pointer

Form

Syntax

```
lhp!    rDg0, Imm6
```

where

<Rdg0> Specifies the destination general-purpose register (r0~r15).

<Imm6> Specifies the 6-bit immediate value.

Operation

```
GPR[rD]g0 = Sign Extend(Mem16[BP+ZeroExtend({Imm5,1'b0})]);
```

Usage

```
lhp!    r4, 0x0E
```

Description

The *lhp!* instruction loads half-word from memory indexed by the sum of base pointer register r2 and the zero-extended 1-bit left-shifted Imm5, sign-extends the half-word and stores this result to general-purpose register rDg0. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

ldiu!

Function Load Immediate Unsigned

Form

Syntax

```
ldiu!    rDg0, Imm8
```

where

<Rdg0> Specifies the destination general-purpose register (r0~r15).

<Imm8> Specifies the 8-bit immediate value.

Operation

$GPR_{rD}g0 = \{24\{0\}, Imm8\};$

Usage

```
ldiu!    r4, 0x0E
```

Description

The *ldiu!* instruction loads an unsigned 8-bit immediate value (*Imm8)* and places the result into general-purpose register rDg0.

Exceptions

None

lw!

Function Load Word

Form

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP			rDg0				rAg0				func4			
0	1	0									1	0	0	0

Syntax

```
lw!     rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$GPR_{rD}g0 = Mem32[GPR_{rA}g0];$

Usage

```
lw!     r6, [r2]
```

Description

The *lw!* instruction loads a word from memory indexed by the content of general-purpose register rAg0, and stores this result to general-purpose register rDg0. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

lwp!

Function Load Word with Base Pointer

Form

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP			rDg0				Imm5					func3		
1	1	1										0	0	0

Syntax

```
lwp!rDg0, Imm7
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<Imm7> Specifies the 7-bit immediate value.

Operation

$GPR_{rD}g0$ = Mem32[BP + Zero Extend({Imm5,2{0}})];

Usage

```
lwp!     R7, 0x10
```

Description

The *lwp!* instruction loads a word from memory indexed by the sum of base pointer register r2 and the zero-extended 2-bit left-shifted Imm5, and stores this result to general-purpose register rDg0. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

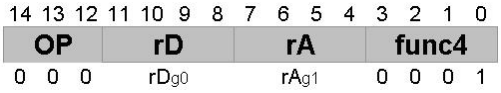
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

mlfh!

Function Move Lower Register from Higher Register

Form



Syntax

```
mlfh!    rDg0, rAg1
```

where

`<rDg0>` Specifies the destination general-purpose register (r0~r15).

`<rAg1>` Specifies the source general-purpose register (r16~r31).

Operation

$GPR_{rD}g0 = GPR_{rA}g1;$

Usage

```
mlfh!    r4, r12
```

Description

The *mlfh!* instruction moves data in a higher source register (r16~r31) to a lower destination register (r0~r15).

Exceptions

None

mhfl!

Function Move Higher Register from Lower Register

Form

Syntax

```
mhfl!    rDg1, rAg0
```

where

<rDg1> Specifies the destination general-purpose register (r16~r31).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

$GPR_{rD}g1 = GPR_{rA}g0;$

Usage

```
mhfl!    r14, r12
```

Description

The *mhfl!* instruction moves data in a lower source register (r0~r15) to a higher destination register (r16~r31).

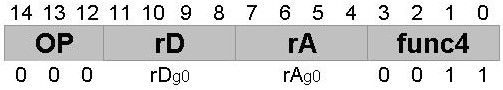
Exceptions

None

mv!

Function Move

Form



Syntax

```
mv!     rDg0, rAg0
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

$GPR_{rD}g0 = GPR_{rA}g0;$

Usage

```
mv!     r15, r6
```

Description

The *mv!* instruction moves data in a lower source register (r0~r15) to a lower destination register (r0~r15).

Exceptions

None

neg!

Function Negative

Form

Syntax

```
neg!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
GPR[rD]g0 = 0 - GPR[rA]g0;
```

Usage

```
neg!    r9, r1
```

Description

The *neg!* instruction subtracts the content of general-purpose register rAg0 from zero to get the negative value of the content of general-purpose register rAg0. In other words, the negative operation performs an addition of the one's complement of register rAg0 and one; and places the resulting 2's complement to general-purpose register rDg0. All the condition flags (N, Z, C, V) are always clobbered by this instruction.

Exceptions

None

nop!

Function No Operation

Form

Syntax

```
nop!
```

Operation

```
No operation;
```

Usage

```
nop!
```

Description

The *nop* instruction is a 16-bit no operation instruction.

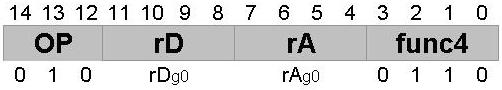
Exceptions

None

not!

Function Logical NOT

Form



Syntax

```
not!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
GPR[rD]g0 = ~GPR[rA]g0;
```

Usage

```
not!    r4g0, r2g0
```

Description

The *not!* instruction bit-wise nots the content of general-purpose register rAg0, and stores the result to general-purpose register rDg0. The condition flags (N, Z) are always clobbered by this instruction.

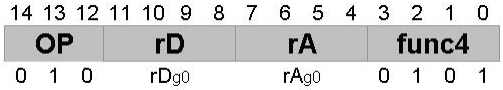
Exceptions

None

or!

Function Logical OR

Form



Syntax

```
or!     rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
GPRrDg0 = GPRrDg0 | GPRrAg0;
```

Usage

```
or!     r11, r2
```

Description

The *or!* instruction bit-wise ors the contents of general-purpose register rDg0 and rAg0, and stores the result to general-purpose register rDg0. The condition flags (N, Z) are always clobbered by this instruction.

Exceptions

None

pop!

Function Load Post-Increment

Form

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP			rDgh				H	rAg0			func4			
0	1	0									1	0	1	0

Syntax

```
pop!    rDgh, [rAg0]
```

where

<rDgh> Specifies the destination general-purpose register. When H bit field is high, rDgh represents registers r16~r31; otherwise rDgh represents registers r0~r15.

<rAg0> Specifies the base address register (r0~r7).

Operation

GPR_{rD} = Mem32[rAg0];

rAg0 = rAg0 + 4;

Notes:

if H=1, GPR_{rD} = r16~r31,

else, GPR_{rD} = r0~r15,

rAg0 = r0 ~ r7

Usage

```
pop!    r18, [r2]
```

Description

The *pop!* instruction loads a word from memory indexed by the effective address, and stores this result to general-purpose register rDgh. The effective memory address is one of the lowest eight general-purpose registers (r0~r7). After the load operation is complete, the base address register rAg0 is incremented by four. If H bit field is high, the destination register is one of the higher registers (r16~r31); otherwise, the destination register is one of the lower registers (r0~r15). If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

push!

Function Store Pre-Decrement

Form

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP			rDgh				H	rAg0			func4			
0	1	0									1	1	1	0

Syntax

```
push!    rDgh, [rAg0]
```

where

<rDgh> Specifies the source general-purpose register. When H bit field is high, rDgh represents registers r16~r31; otherwise rDgh represents registers r0~r15.

<rAg0> Specifies the base address register (r0~r7).

Operation

```
Mem32[rAg0-4] = GPR_rD;
rAg0 = rAg0 - 4;

Notes:
if H=1, GPR_rD = r16~r31,
else, GPR_rD = r0~r15,
rAg0 = r0 ~ r7
```

Usage

```
push!    r16, [r2]
```

Description

The *push!* instruction stores the content of general-purpose register rDgh as word to memory indexed by the effective address. The effective memory address is one of the lowest eight general-purpose registers (r0~r7) minus four. After the store operation is complete, the base address register is updated with the effective address. If H bit field is high, the source register is one of the higher registers (r16~r31); otherwise, the source register is one of the lower registers (r0~r15). If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

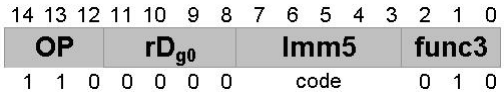
Bus error exception

Address error exception

sdbbp!

Function Software Debug Breakpoint

Form



Syntax

```
sdbbp!   Code(Imm5)
```

where

<code> Specifies the 5-bit software parameter value.

Operation

```
if(not in Debug Mode)
{
    DEPC = Address of SDBBP instruction;
    DM = 1'b1; BrkSt = 1'b1; DBp = 1'b1;
if (ice_enable & sj_probe_en)
    PC = 0xFF00_0000;
else
    PC = {EXCPBase[31:16], 16'h1FC};
}
else
    NOP;
```

Usage

```
sdbbp!   0x00
```

Description

The *sdbbp!* instruction induces a software debug breakpoint exception, and transfers control to an exception handler. The exception vector location depends on the ICE and probe circuit. When this instruction is executed, the corresponding status bit in debug register (DREG) will be set and the debug exception program counter (DEPC) will point to the address of *sdbbp* instruction.

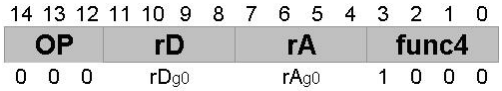
Exceptions

None

sll!

Function Shift Left Logical

Form



Syntax

```
sll!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
C = Undefine;
GPR[rD]g0 = {GPR[rD]g0[(31 - GPR[rA]g0[4:0]):0], GPR[rA]g0[4:0]{0}};
```

Usage

```
sll!    r4, r2
```

Description

The *sll!* instruction left shifts the content of general-purpose register rDg0 with zero insertion by the shift amount, the lower 5 bits of register rAg0, and stores this result to general-purpose register rDg0. The condition flags (N, Z, C) are always clobbered by this instruction.

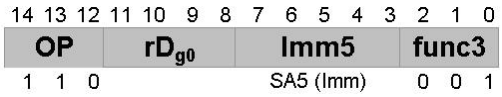
Exceptions

None

slli!

Function Shift Left Logical with Immediate

Form



Syntax

```
slli!    rDg0, SA          (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<SA5> Specifies the 5-bit shift amount.

Operation

```
N = Undefine;
Z = Undefine;
C = Undefine;
GPRrDg0 = {GPRrDg0[(31-SA):0], SA{0}};
```

Usage

```
slli!    r14, 0x0E
```

Description

The *slli!* instruction left shifts the content of general-purpose register rDg0 with zero insertion by the shift amount, the 5-bit SA5, and stores this result to general-purpose register rDg0. The condition flags (N, Z, C) are always clobbered by this instruction.

Exceptions

None

srli!

Function Shift Right Logical with Immediate

Form

Syntax

```
srli!    rDg0, SA          (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<SA5> Specifies the 5-bit shift amount.

Operation

```
N = Undefine;
Z = Undefine;
C = Undefine;
GPRrDg0 = {SA{0}, GPRrDg0[31:SA]};
```

Usage

```
srli!    r9, 0x0E
```

Description

The *srli!* instruction right shifts the content of general-purpose register rDg0 with zero insertion by the shift amount, the 5-bit SA5, and stores this result to general-purpose register rDg0. The condition flags (N, Z, C) are always clobbered by this instruction.

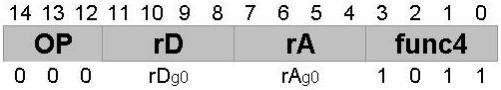
Exceptions

None

sra!

Function Shift Right Arithmetic

Form



Syntax

```
sra!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
C = Undefine;
GPRrDg0 = {GPRrAg0 [4:0]{GPRrDg0 [31]}, GPRrDg0[31:GPRrAg0 [4:0]]};
```

Usage

```
sra!    r11, r12
```

Description

The *sra!* instruction right shifts the content of general-purpose register rDg0 with sign insertion by the shift amount, the lowest 5 bits of rAg0, and stores this result to general-purpose register rDg0. The condition flags (N, Z, C) are always clobbered by this instruction.

Exceptions

None

srl!

Function Shift Right Logical

Form

Syntax

```
srl!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
C = Undefine;
GPRrDg0 = {GPRrAg0[4:0]{0}, GPRrDg0[31: GPRrAg0[4:0]]};
```

Usage

```
srl!    r6, r0
```

Description

The *srl!* instruction right shifts the content of general-purpose register rDg0 with zero insertion by the shift amount, the lowest 5 bits of rAg0, and stores this result to general-purpose register rDg0. The condition flags (N, Z, C) are always clobbered by this instruction.

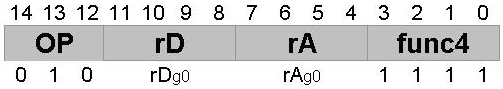
Exceptions

None

sb!

Function Store Byte

Form



Syntax

```
sb!     rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$Mem8[GPR_{rA}g0] = GPR_{rD}g0[7:0];$

Usage

```
sb!     r4, [r2]
```

Description

The *sb!* instruction stores the lowest 8 bits of general-purpose register rDg0 to the byte memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rAg0.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sbp!

Function Store Byte with Base Point

Form

Syntax

```
sbp!    rDg0, Imm5
```

where

`<rDg0>` Specifies the destination general-purpose register (r0~r15).

`<Imm5>` Specifies the 5-bit immediate value.

Operation

```
Mem8[BP+ZeroExtended(Imm5)] = GPRrDg0[7:0];
```

Usage

```
sbp!    r4, 0x0E
```

Description

The *sbp!* instruction stores the lowest 8 bits of general-purpose register rDg0 to the byte memory indexed by the effective memory address. The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate value.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sh!

Function Store Half-Word

Form

Syntax

```
sh!     rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$Mem16[GPR_{rA}g0] = GPR_{rD}g0[15:0];$

Usage

```
sh!     r13, [r5]
```

Description

The *sh!* instruction stores the lower 16-bit of general-purpose register rDg0 to the half word memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rAg0. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

shp!

Function Store Half-word with Base Pointer

Form

Syntax

```
shp!    rDg0, Imm6
```

where

`<rDg0>` Specifies the destination general-purpose register (r0~r15).

`<Imm6>` Specifies the 6-bit immediate value.

Operation

```
Mem16[BP+ZeroExtend({Imm5, 1'b0})] = GPRrDg0[15:0];
```

Usage

```
shp!    r6, 0x0e
```

Description

The *shp!* instruction stores the lowest 16 bits of general-purpose register rDg0 to the half-word memory indexed by the effective memory address. The effective memory address is the sum of base pointer register (r2) and the zero-extended 1-bit left-shifted Imm5. If the effective address is not half-word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sw!

Function Store Word

Form

Syntax

```
sw!     rDg0, [rAg0]
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

$Mem32[GPR_{rA}g0] = GPR_{rD}g0[31:0];$

Usage

```
sw!     r4, [r2]
```

Description

The *sw!* instruction stores the content of general-purpose register rDg0 to the word memory indexed by the effective memory address. The effective memory address is the content of general-purpose register rAg0. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

swp!

Function Store Word with Base Pointer

Form

Syntax

```
swp!    rDg0, Imm7
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<Imm7> Specifies the 7-bit immediate value.

Operation

```
Mem32[BP+ZeroExtend(Imm5, 2{0})] = GPRrDg0[31:0];
```

Usage

```
swp!    r8, 0x10
```

Description

The *swp!* instruction stores the content of general-purpose register rDg0 to the word memory indexed by the effective memory address. The effective memory address is the sum of base pointer register (r2) and the zero-extended 2-bit left-shifted Imm5. If the effective address is not word aligned, an address alignment error exception will occur.

Exceptions

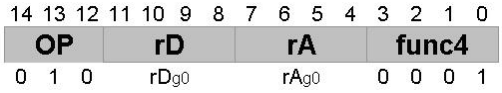
Bus error exception

Address error exception: In User mode, the address available only in Kernel/Debug mode is accessed.

sub!

Function Subtract

Form



Syntax

```
sub!    rDg0, rAg0                (CU = 1)
```

where

<rDg0> Specifies the destination general-purpose register (r0~r15).

<rAg0> Specifies the base address register (r0~r15).

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
GPR[rD]g0 = GPR[rD]g0 - GPR[rA]g0;
```

Usage

```
sub!    r14, r1
```

Description

The *sub!* instruction subtracts the content of general-purpose register rAg0 from that of rDg0 and updates general-purpose register rDg0 with this subtraction result. All the condition flags (N, Z, C, V) are always clobbered by this instruction.

Exceptions

None

subei!

Function Subtract with Exponent Immediate

Form

Syntax

```
subei!   rDg0, imm4              (CU = 1)
```

where

<rDg0> Specifies the destination and first source general-purpose register (r0~r15).

<imm4> Specifies the 4-bit exponent immediate value.

Operation

```
N = undefined
Z = undefined
C = undefined
V = undefined
```

$GPR_{rD}g0 = GPR_{rD}g0 - 2^{Exp4}$;

Usage

```
subei!   r10, 0x0E
```

Description

The *subei!* instruction subtracts the exponent immediate 2^{Exp4} from general-purpose register rDg0 and updates general-purpose register rDg0 with this subtraction result. All the condition flags (N, Z, C, V) are always clobbered by this instruction.

Exceptions

None

t{cond}!

Function Test and Set the Condition Flag T

Form

14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
OP	rD	rA	func4
0 0 0	EC	0 0 0 0	0 1 0 1

Syntax

```
t{cond}! (CU = 0)
```

where

`t` Specifies the condition flag T. T flag is used in parallel condition execution to determine whether the execution is through the true or false path.

`{cond}` Specifies the test condition. Table 2-9 shows the 16 condition options that {*cond*} could be, with their corresponding execution condition (EC) field encoding.

Operation

```
case(EC)
    begin
        4'b0000: T = C;
        4'b0001: T = ~C;
        4'b0010: T = C & ~Z;
        4'b0011: T = ~C | Z;
        4'b0100: T = Z;
        4'b0101: T = ~Z;
        4'b0110: T = (Z==0) & (N==V);
        4'b0111: T = (Z==1) | (N!=V);
        4'b1000: T = (N==V);
        4'b1001: T = (N!=V);
        4'b1010: T = N;
        4'b1011: T = ~N;
        4'b1100: T = V;
        4'b1101: T = ~V;
        4'b1110: T = (CNT>0);
        4'b1111: T = 1;
    end
```

Usage

```
tcc!
tcnz!
tset!
```

Description

The *t{cond}* instruction sets or clears T flag according to the result of condition flag test. The EC field specifies the corresponding condition flag test.

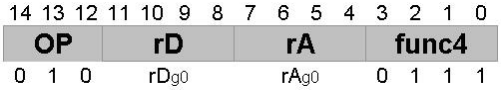
Exceptions

None

xor!

Function Logical XOR

Form



Syntax

```
xor!    rDg0, rAg0              (CU = 1)
```

where

<rDg0> Specifies the destination and source general-purpose register (r0~r15).

<rAg0> Specifies the source general-purpose register (r0~r15).

Operation

```
N = Undefine;
Z = Undefine;
GPR[rD]g0 = GPR[rD]g0 ^ GPR[rA]g0;
```

Usage

```
xor!    r9, r13
```

Description

The *xor!* instruction bit-wise xors the contents of general-purpose register rDg0 and rAg0, and updates general-purpose register rD with this result. The condition flags (N, Z) are always clobbered by this instruction.

Exceptions

None

2.3 Synthetic Instruction Set

subrix

Function ADD Register with Immediate

Syntax

```
subri    rD, rA, SImm14         (CU = 0)
subri.c  rD, rA, Simm14         (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<SImm14>	Specifies 14-bit signed immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

This instruction is equivalent to

```
addri    rD, rA, -SImm14        (CU = 0)
addri.c  rD, rA, -SImm14        (CU = 1)
```

Usage

```
subri    r4, r3, 0x0123
```

subix

Function ADD with Immediate

Syntax

```
subi     rD, SImm16        (CU = 0)
subi.c   rD, Simm16        (CU = 1)
```

where

`<rD>` Specifies the destination general-purpose register.

`<Simm16>` Specifies 16-bit signed immediate value.

`<.c>` Specifies that the condition flag updating bit CU is true.

Operation

This instruction is equivalent to

```
addi     rD, -SImm16       (CU = 0)
addi.c   rD, -Simm16       (CU = 1)
```

Usage

```
subi     r4, 0x1234
```

subisx

Function ADD with Immediate Shifted

Syntax

```
subis    rD, Imm16          (CU = 0)
subis.c  rD, Imm16          (CU = 1)
```

where

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies 16-bit immediate value.
<.c>	Specifies that the condition flag updating bit CU is true.

Operation

This instruction is equivalent to

```
addis    rD, -Imm16          (CU = 0)
addis.c  rD, -Imm16          (CU = 1)
```

Usage

```
subis    r4, 0x1234
```

li

Function Load Immediate to Register

Syntax

```
li      rD, Imm32
```

where

`<rD>` Specifies the destination general-purpose register.

`<Imm32>` Specifies 32-bit immediate value.

Operation

If `(( Imm32 value> -32768) & (Imm32 <32767))`

then `li rD, Imm32` is equivalent to

```
ldi  rD, Imm16     /* ex: li r6,0x1234 */
```

else if `((Imm32 value & 0xffff )== 0)`

then `li rD, Imm32` is equivalent to

```
ldis rD, Imm16      /* ex: li r6,0x12340000 */
```

else `li rD, Imm32` is equivalent to

```
ldis rD, HI16(Imm32)
ori  rD, LO16(Imm32)
```

Usage

```
li      r4, 0x7234
li      r5, 0x80001234
```

la

Function Load Immediate to Register

Syntax

```
la      rD, LABEL
la      rd, value
```

where

<rD> Specifies the destination general-purpose register.

<LABEL > Specifies the symbol name.

<value> Specifies the 32-bit immediate value.

Operation

1.

`la rD, LABEL` is equivalent to

```
ldis    rD, HI16(LABEL)
ori     rD, LO16(LABEL)
```

2.

`la rD, value` is equivalent to

```
ldis    rD, HI16(value)
ori     rD, LO16(value)
```

Usage

1.

```
    la      r6, label
label:
```

2.

```
   la       r7, 0x1234
```

lb

Function Load Byte Signed

Syntax

```
lb      rD, [rA]
lb      rD, label
lb      rD, value
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<label> Specifies the target symbol name

<value> Specifies the 32-bit immediate value. (for absolute addressing)

Operation

1.

This instruction is equivalent to

```
lb      rD, [rA, 0]
```

2.

`lb rD, label` is equivalent to

```
la   r1, label
lb   rD, [r1]
```

when the label is in .sbss / .sdata section, `lb rD, label` is equivalent to

```
lb   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`lb rD, value` is equivalent to

```
la   r1, value
lb   rD, [r1]
```

Usage

```
lb      r6, [r5]
lb      r6, label
lb      r7, 0 /* absolute addressing */
```

lbu

Function Load Byte Unsigned

Syntax

```
lbu     rD, [rA]
lbu     rD, label
lbu     rD, value
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

<label> Specifies the target symbol name.

<value> Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
lbu     rD, [rA,0]
```

2.

`lbu rD,label` is equivalent to

```
la   r1, label
lbu  rD, [r1]
```

when the label is in .sbss / .sdata section, `lbu rD, label` is equivalent to

```
lbu  rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`lbu rD, value` is equivalent to

```
la   r1, value
lbu  rD, [r1]
```

Usage

```
lbu     r6, [r5]
lbu     r6, label
lbu     r7, 0 /* absolute addressing */
```

lh

Function Load Half-word Signed

Syntax

```
lh      rD, [rA]
lh      rD, label
lh      rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name.
<value>	Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
lh      rD, [rA, 0]
```

2.

`lh rD, label` is equivalent to

```
la   r1, label
lh   rD, [r1]
```

when the label is in .sbss / .sdata section, `lh rD, label` is equivalent to

```
lh   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`lh rD, value` is equivalent to

```
la   r1, value
lh   rD, [r1]
```

Usage

```
lh      r6, [r5]
lh      r6, label
lh      r7, 0 /* absolute addressing */
```

lhu

Function Load Half-word Unsigned

Syntax

```
lhu     rD, [rA]
lhu     rD, label
lhu     rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name.
<value>	Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
lhu     rD, [rA, 0]
```

2.

`lhu rD, label` is equivalent to

```
la   r1, label
lhu  rD, [r1]
```

when the label is in .sbss / .sdata section, `lhu rD, label` is equivalent to

```
lhu  rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`lhu rD, value` is equivalent to

```
la   r1, value
lhu  rD, [r1]
```

Usage

```
lhu     r6, [r5]
lhu     r6, label
lhu     r7, 0 /* absolute addressing */
```

lw

Function Load Word

Syntax

```
lw      rD, [rA]
lw      rD, label
lw      rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name.
<value>	Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
lw      rD, [rA, 0]
```

2.

`lw rD,label` is equivalent to

```
la   r1, label
lw   rD, [r1]
```

when the label is in .sbss / .sdata section, `lw rD, label` is equivalent to

```
lw   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`lw rD, value` is equivalent to

```
la   r1, value
lw   rD, [r1]
```

Usage

```
lw      r6, [r5]
lw      r6, label
lw      r7, 0 /* absolute addressing */
```

sb

Function Store Byte

Syntax

```
sb      rD, [rA]
sb      rD, label
sb      rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name.
<value>	Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
sb      rD, [rA, 0]
```

2.

`sb rD,label` is equivalent to

```
la   r1,label
sb   rD,[r1]
```

when the label is in .sbss / .sdata section, `sb rD, label` is equivalent to

```
sb   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`sb rD, value` is equivalent to

```
la   r1, value
sb   rD, [r1]
```

Usage

```
sb      r6, [r5]
sb      r6, label
sb      r7, 0 /* absolute addressing */
```

sh

Function Store Half-word

Syntax

```
sh      rD, [rA]
sh      rD, label
sh      rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name .
<value>	Specifies the 32-bit immediate value.

Operation

1.

This instruction is equivalent to

```
sh      rD, [rA, 0]
```

2.

`sh rD,label` is equivalent to

```
la   r1,label
sh   rD,[r1]
```

when the label is in .sbss / .sdata section, `sh rD, label` is equivalent to

```
sh   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`sh rD, value` is equivalent to

```
la   r1, value
sh   rD,[r1]
```

Usage

```
sh      r6, [r5]
sh      r6, label
sh      r7, 0 /* absolute addressing */
```

sw

Function Store Word

Syntax

```
sw      rD, [rA]
sw      rD, label
sw      rD, value
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the base address register.
<label>	Specifies the target symbol name
<value>	Specifies the 32-bit immediate value

Operation

1.

This instruction is equivalent to

```
sw      rD, [rA, 0]
```

2.

`sw rD,label` is equivalent to

```
la   r1, label
sw   rD, [r1]
```

when the label is in .sbss / .sdata section, `sw rD, label` is equivalent to

```
sw   rD, [gp, offset]
```

where <offset> is the offset between <label> and r28.

3.

`sw rD, value` is equivalent to

```
la   r1, value
sw   rD, [r1]
```

Usage

```
sw      r6, [r5]
sw      r6, label
sw      r7, 0 /* absolute addressing */
```

mul

Function Multiply

Syntax

```
mul     rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

This is equivalent to

```
mul     rA, rB
mfcel   rD
```

Usage

```
mul r4, r6, r7
```

mulu

Function Multiply Unsigned

Syntax

```
mulu    rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

This is equivalent to

```
mulu    rA, rB
mfcel   rD
```

Usage

```
mulu r4, r6, r7
```

div

Function Divide

Syntax

```
div     rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

This is equivalent to

```
div     rA, rB
mfcel   rD
```

Usage

```
div r4, r6, r7
```

divu

Function Divide Unsigned

Syntax

```
divu    rD, rA, rB
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

This is equivalent to

```
divu    rA, rB
mfcel   rD
```

Usage

```
divu r4, r6, r7
```

rem

Function Divide

Syntax

```
rem     rD, rA, rB
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

This is equivalent to

```
div     rA, rB
mfceh   rD
```

Usage

```
rem r4, r6, r7
```

remu

Function Divide Unsigned

Syntax

```
remu    rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

This is equivalent to

```
divu    rA, rB
mfceh   rD
```

Usage

```
remu r4, r6, r7
```

3 GNU Compiler for S+core

3.1 Command Line Options

To run the C compiler, type: `gcc.`

Usage:

```
gcc [option|file]
```

Table 3-1Compiler Command Line Options

Option	Description
-mSCORE5U	S+core5U compiler option
-mSCORE5	S+core5 compiler option
-mSCORE7	S+core7 compiler option
-meb/-mel	Big / little endian option
-S	Compile code into the assembly language
-E	Run only the preprocessor on the named C programs
-o *file*	Place output in the file `*file*'
--help	Print a description of the command line options recognized by gcc
-ansi	Support all ANSI standard C programs and turn off certain features of GCC that are incompatible with ANSI C. The "-ansi" option does not cause non-ANSI programs to be rejected gratuitously. For that, "-pedantic" is required in addition to "-ansi".
-pedantic	Issue all the warnings demanded by the strict ANSI C and ISO C++
-w	Inhibit all warnings
-Wall	Enable all the warnings about constructions that users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
-Werror	Make all warnings into errors
-Q	Make the compiler print out names of each compiled function , and print the statistics about each pass when the compilation is done.
-D*macro*	Define macro *macro* with the string `1' as its definition
-D*macro=defn*	Define macro *macro* as `*defn*'. All instances of `-D' on the command line are processed before any `-U' options.
-U*macro*	Undefined macro *macro*
-gstab+	Produce debugging information for debuggers
-I*dir*	Add the directory `*dir*' to the head of the list of directories to be searched for header files.
-O0	No optimization
-O1	The compiler tries to reduce code size and execution time.
-O2	Optimize more than O1. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed.
-O3	Optimize more than O2. This will turn on all the optimizations -O2 does and turn on the function in-lining.

Option	Description
-Os	Optimize for size. This enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.
-nostartfiles	Link without crt*.o
-nostdlib	Link without standard libraries (libc.a, libm.a…)

Example 1:

In DOS command line, type:

gcc -S test.c -o test.s

The test.c is a C code file.

After this command is run, an asm file with name of test.s generates.

Example2:

gcc -S –gstabs+ test.c -o test.s

The test.c is a C code file.

Run this command to generate an asm file with name of test.s, which includes the debug information.

Example3:

gcc -S -O2 test.c -o test.s

The test.c is a c code file.

Run current command to generate an optimized asm file with name of test.s.

Example4:

gcc -S -O2 –gstabs+ test.c -o test.s

The test.c is a c code file.

This generates an optimized asm file with name of test.s, which includes the debug information.

3.2 S+core7 C Compiler Basic Data Types

Table 3-2 S+core7 C Compiler Basic Data Types

Data type	Size
char / unsigned char	8 bits
short / unsigned short	16 bits
int / unsigned int	32 bits
long / unsigned long	32 bits
long long / unsigned long long	64 bits
float	32-bit IEEE floating point format
double	64-bit IEEE floating point format

3.3 S+core7 C Compiler Calling Convention

The S+core 7 has 32 general registers, here we divide them into several register classes (listed in the following table).

Register	Usage	Preserved across function calls
r0	stack pointer	Yes
r1	temporary generally used by assembler.	No
r2	frame pointer	Yes
r3	link registers	Yes
r4 - r7	used to pass arguments and return value.	No
r8 - r11	calling-saved register	No
r12 - r21	callee-saved register; optionally used as frame pointer	Yes
r22 - r27	calling-saved register	No
r28	global pointer	Yes
r29	compiler reserved	Yes
r30 - 31	used by operating system only	Yes
hi	multiply/divide special register. Holds the most significant 32 bits of multiply or the remainder of a divide	No
lo	multiply/divide special register. Holds the least significant 32 bits of multiply or the quotient of a divide	No
sr0	special register for counter	No
sr1	special register for load combine instructions	No
sr2	special register for store combine instructions	No

The compiler will pass 1st ~ 4th parameters (if these parameters are smaller than 32 bits) using the registers r4-r7, and put the return value into the register r4. If the function has more than 4 parameters, the 5th parameter will be placed in memory [sp+16], the 6th will be placed in [sp+20], and so on.

Examples

Here we list an example of arguments passing. On the left side is the C source code, and the assembly code generated by the compiler is listed on the right side. We can find that the 5th parameter (5) will be placed in [sp+16] and the 6th parameter will be placed in [sp+20].

```
void arg6  (int a,int b,int c,int
d,int e,int f);
int main (void)
{
  arg6 (1,2,3,4,5,6);
}

void arg6  (a,b,c,d,e,f)
       int a,b,c,d,e,f;
{
}
```

```
 .text
       .align  2
       .globl  main
main:
       sw      r2,[r0,-4]+
       sw      r3,[r0,-4]+
       sw      r0,[r0,-4]+
       subi    r0,24
       mv      r2,r0
       la      r8,__main
       brl     r8
       li      r8,5
       sw      r8,[r0,16]
       li      r8,6
       sw      r8,[r0,20]
       li      r4,1
       li      r5,2
       li      r6,3
       li      r7,4
       la      r8,arg6
       brl     r8
       addi    r2,24
       lw      r0,[r2]+,4
       lw      r3,[r0]+,4
       lw      r2,[r0]+,4
       br      r3

       .align  2
       .globl  arg6
arg6:
       sw      r2,[r0,-4]+
       sw      r0,[r0,-4]+
       mv      r2,r0
       lw      r0,[r2]+,4
       lw      r2,[r0]+,4
       br      r3
```

3.4 Using Inline Assembly

This section describes how to use the inline assembly in S+core gcc.

To learn more details about inline assembly, please refer to :
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.

3.4.1 Assembler Instructions with C Expression Operands

```
asm ("andri %1,%0, 0x10" : "=r"(result): "r"(src));
```

Here ‘src’ is the C expression for the input operand, while ‘result’ is that of the output operand. Each of them has a ‘”r”’ as its operand constraint, saying that, an integer register is required. The ‘=’ in ‘=r’ indicates that the operand is an output; all output operands’ constraints must use ‘=’. The constraints use the same language used in the machine description (*note Constraints::.).

3.4.2 Basic Inline Assembly

The format for basic inline assembly is very simple,

```
asm (“statements”);
```

Examples:

```
1.asm(“nop”) ;
```

It will simply output an assembly statement “nop” in the assembly language file generated by the compiler,

```
2.asm(“mv r1,r2”);
```

This statement will move the value in r2 to r1.
Note: This statement may be harmful, since the r1 is damaged but you have not told the S+core gcc

3.4.3 Extended Inline Assembly

In basic inline assembly, only instructions can be supported. However, besides instructions, extended inline assembly can also accommodate operands . This will allow us to specify the input registers, output registers and a list of clobbered registers.

Here is the basic format:

```
asm ( “statements” : output_registers : input_registers : clobbered_list);
```

3.4.4 Assembler Template

The assembler template contains a set of assembly instructions inserted inside the C program. The general formats are each instruction should be enclosed within double quotes; the entire group of instructions should be within double quotes. Each instruction should be terminated with a delimiter; the valid delimiters are newline (\n) and semicolon (;) and ’\n’ may be followed by a tab(\t).
Operands corresponding to the C expressions are represented by %0, %1 ...

3.4.5 Operands

In an assembler instruction using ‘asm’, you can now specify the operands of the instruction using C expressions. This means no more guessing on which registers or memory locations will contain the data you want to use.

The operands behind the first colon represent the output operands; the second, represent the input

operands; and the third, to be clobbered after the inline assembly instruction.

If there are more than one output operands, they should be separated by commas. The same criterion are applied to inout and clobbered operands.

In spite of the actual operand, an operand constraint takes place; that is, a string and a pair of parentheses should enclose the actual operand.

3.4.6 Commonly Used Constraints

The operand constraints make S+core gcc recognize what kind of value to be used in the assembly instruction template.

1. Register operand constraint(r)

When operands are specified using this constraint, they are stored in General Purpose Registers(GPR)

Example:

```
int main()
{
    int a = 3;
    int b = 6;
    asm("mv %0,%1":"=r"(a):"r"(b));
    return 0;
}
```

The asm statement “asm("mv %0,%1":"=r"(a):"r"(b));” is to pass the value of b to a, so, the value of a is equal to 6 after the inline assembly statement is executed.

2. Memory operand constraint(m)

When the operands are in the memory, any operation performed on them will occur directly in the memory location, as opposed to register constraints, which first saves the value in a register to be modified and then write it back to the memory location. But register constraints are usually used only when they are absolutely necessary for an instruction or they significantly speed up the process.

Example1.

```
int main()
{
int g = 4;
int l = 3;
__asm__ ("sw %1, %a0": "=m"(g): "r"(l));
return 0;
}
```

The asm statement is to store the value of a(3) into g. Note: You must use the format “%a0”, because g is a memory address, “a” must be added in “%a0”.

The assembly code is as follows:

```
sw r4, [r2,  0]
```

Example 2.

```
int lw(){
```

```
  int l;
  int g;
  __asm__ ("lw %0, %a1": "=r"(l): "m"(g));
  return a;
}
```

The assembly code is as follows:

```
lw r4, [r2,  4]
```

3.4.7 Clobber List

Certain instructions clobber hardware registers. We have to list these registers in the clobber-list, i.e., the field after the third '**:**' in the asm function. This is to inform S+core gcc that we will use and modify them ourselves. So, S+core gcc will not assume that the values it loads into these registers are valid. We haven't listed the input and output registers in this list, because they are specified explicitly as constraints and S+core gcc knows that "asm" uses them. If the instructions use any other registers implicitly or explicitly, and the registers do not present either in input or output constraint list), then registers have to be specified in the clobbered list.

Example:

```
asm("mv r1,r2": : :"r1");
```

This statement will move the value in r2 to r1, and destroy (clobber) the value in the r1 register. The "r1" behinds the third ":" can tell the compiler the value in "r1" will be damaged.

4 GNU Assembler for S+core

4.1 Command Line Options

Table 4-1 Assembler Command Line Options

Option	Description
-SCORE5U	–FIXDD will fix data dependency for SCORE5U.
-SCORE5	–FIXDD will fix data dependency for SCORE5.
-SCORE7	Default setting, –FIXDD will fix data dependency for SCORE7.
-EB	Big Endian (default)
-EL	Little Endian
-FIXDD	Assembler will fix data dependency
-NWARN	Assembler will not generate data dependency warning messages.
-USE_R1	Assembler will not generate warning messages for using R1 (temp) register.
-G0	Gp switch size is zero, gp addressing will not be used.
--gstabs	Generate stabs debugging information for each assembler line.
- I *dir*	Add directory *dir* to the search list for .include directives
-o *objfile*	Name the object-file output from as *objfile*
-O0	Assembler will not generate optimized code.

Usage:

```
score-linux-elf-as –SCORE5U –EL –USE_R1 –gstabs –I .  –o test.o test.s
```

4.2 Assembly Language Syntax

Assembly language source files consist of a sequence of statements, one per line. Each statement has the following format, each part of which is optimal:

Label: instruction # comment

A Label allows you to identify the location of program counter (ie, its address) at that point in the program. This label then can be used, for example, as a target for branch instructions or for load or store instructions. A label can be any valid symbol, followed by a colon":". A valid symbol, in turn, is one that only uses the alphabetic characteristics A to Z and a to z, the digit 0 to 9, as well as "_", "-" and "$". Note that you cannot start a symbol with a digit.

A comment is anything that is following "#". Everything that is following "#" is ignored to the end of the line except #include and #define. C-style comments (using /* and */) are also allowed.

The instruction field is the read meat of your program: it is any valid S+core assembly language instruction that you can use. It also includes the so-called pseudo op operations or assembler directives: "instructions" that tell the assembler itself to do something. These directives will be

discussed later.

4.3 Assembler Directive

All assembler directives have names that begin with a full-stop ".". The directives presented here (in alphabetical order) are the most useful ones that you may need to use in your assembly language programs.

■ .align

Syntax .align alignment [, [fill][, max]]

Function Pad the location counter (in the current subsection) to a particular storage boundary for the next data directives. ***fill*** gives the fill value to be stored in the padding bytes, if ***fill*** is omitted, the padding bytes are normally zero. ***max*** is the maximum number of bytes that should be skipped by this alignment directive.

Example .align 8, 0x1, 16

Advances the location counter until it is a multiple of 8, but can not exceed 16, stores 0x1 in the padding bytes. If the location counter is already a multiple of 8, no change is needed.

■ .ascii

Syntax .ascii strings

Function Insert the string into the assembly, with no null character

Example .ascii "JNZ"

Inserts the byte 0x4a 0x4e 0x5a

■ .asciz

Syntax .asciz strings

Function Like ".ascii", but follows the string with a zero byte

Example .asciz "JNZ"

Inserts the byte 0x4a 0x4e 0x5a 0x00

■ .balign

Syntax .balign <power_2>[, [fill][, max]]

Function Pad the location counter (in the current subsection) to a particular storage boundary

Example .balign 8

Advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

■ .byte

Syntax .byte expressions

Function Insert the byte value of the expression into the object file. ***expressions*** are separated by comments.

Example .byte 64,'A'

Inserts the bytes 0x40 , 0x41

.byte 0x42

Inserts the byte 0x42 (0x or 0X are in hexadecimal)

.byte 0b1000011, 0104

Inserts the bytes 0x43 , 0x44 (0b or 0B are in binary, numbers starting with 0 are in octal)

■ .comm

Syntax .comm symbol, length

Function Declare a common, or un-initialized data object by specifying the object's name ***symbol*** with size ***length***

Example .comm uu, 16

Declares global common symbol uu with 16 bytes. If ld sees a definition for uu or more common symbol uu with the same size, then it will allocate 16 bytes of uninitialized memory. If ld sees multiple common symbols uu without the same size of 16 bytes, it will allocate space using the largest size.

■ .data

Syntax .data [subsection]

Function Switch the destination of the following statements into the ***data*** section of the final executable

Example .data

Switches to ***data*** section

■ .equ

Syntax .equ symbol, expression

Function Set the value of ***symbol*** to ***expression***

Example .equ zzz, (5*8)+2

zzz is absolute symbol with address to be 42; this is equivalent to zzz = 42.

■ .extern

Syntax .extern symbol

Function Specify the ***symbol*** defined in other source file

Example .extern label_zzz

Specifies label_zzz defined in other source file

■ .global

Syntax .global symbol

Function Specify the ***symbol*** to be made globally visible to all other modules which are part of executable file and visible to linker

Example .global _start

Specifies _start globally visible to all other module, including linker

■ .include

Syntax .include filename

Function Insert the contents of ***"filename"*** into the current source file. This is exactly the same as using ***#include*** in C.

Example .include "aaa.h"

Same as #include file aaa.h

■ .hword

Syntax .hword expression

Function Expect zero or more *expressions*, and emit a 16-bit number for each expression

Example .hword 0xaa55,12345

Inserts the bytes 0x55, 0xaa, 0x39, 0x30

■ .int

Syntax .int expression

Function Expect zero or more *expressions*, of any section, separated by commas. For each expression, at run time, emit a number that is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

Example .int aaa

Inserts 4 bytes in uninitialized section for symbol aaa

■ .org

Syntax .org new-lc[,fill]

Function Advance the location counter of the current section to new-lc, and the intervening bytes are filled with ***fill;*** if the comma and ***fill*** are omitted, ***fill*** defaults to zero. You can't use ***.org*** to cross sections.

Example .org 0x1234

Advances the location counter of the current section to 0x1234. The intervening bytes are filled with zero.

■ .section

Syntax .section <section-name> {,"<flags>"}

Function Assemble the following code into a section named ***section-name***

Example .section .text1,"wa"

Starts text1 section which are writable and allowable

■ .space

Syntax .space size[,fill]

Function This directive emits ***size*** bytes, each of value ***fill***. If the comma and ***fill*** are omitted, ***fill*** is assumed to be zero.

Example .space 100, 0x11

Inserts 100 bytes of 0x11 from current position

■ .text

Syntax .text

Function Switch the destination of following statements into the ***text*** section

Example .text

Switches to ***text*** section

■ .word

Syntax .word expression

Function Insert the (32-bit) word value of expression into the object file

Example .word 0xdeadbeaf

Inserts the bytes 0xaf 0xbe 0xad 0xde into the object file

4.4 S+core Specific Directives

■ .set nofixdd

Syntax .set nofixdd

Function Not to insert bubbles for data dependency instructions. .set nofixdd is default.

Example .set nofixdd

Not to insert bubbles when data dependency occurs.

■ .set fixdd

Syntax .set fixdd

Function Insert bubbles for data dependency instructions.

Example .set fixdd

Inserts bubbles when data dependency occurs.

■ .set nwarn

Syntax .set nwarn

Function Not to generate warnings if the data dependency occurs among source machine language instructions

Example .set nwarn

Not to pop up warning message when data dependency occurs

■ .set r1

Syntax .set r1

Function Not to generate warnings if the source program uses ***r1***. Allow user to use ***r1***

Example .set r1

Not to pop up warning messages if ***r1*** is used

■ .set nor1

Syntax .set nor1

Function Generate warnings if the source program uses ***r1***

Example .set nor1

Pops up warning messages if ***r1*** is used

■ .set volatile

Syntax .set volatile

Function Not to do any optimization after this directive until ***.set optimize***

Example .set volatile

Assembler will not do any optimization after this directive until ***.set optimize***.

■ .set optimize

Syntax .set optimize

Function Do optimization after this directive until ***.set volatile***

Example .set optimize

Assembler will do optimization after this directive until ***.set volatile***.

■ .sdata

Syntax .sdata

Function Switch the destination of following statements into the ***sdata*** section, which is used for small data

Example .sdata

Switches to ***sdata*** section

■ .rdata

Syntax .rdata

Function Switch the destination of following statements into the ***rdata*** section, which is used for read-only data

Example .rdata

Switches to ***rdata*** section

■ .frame

Syntax .frame frame-register, offset, return-pc-register

Function Describe a stack frame. No stack traces can be done in the debugger without ***.frame***. The first register is the frame register; offset is the distance from the frame register to the virtual frame pointer; the second register is the return program register. You must use ***.ent*** before ***.frame*** and only one ***.frame*** can be used per ***.ent***.

Example .frame r2, 0x100000, r3

Tells a stack frame, the frame register is r2, and the offset from sp to the virtual frame pointer is 0x100000, the return program register is r3.

■ .mask

Syntax .mask bit-mask, frame-offset

Function Indicate which of the integer registers are saved in the current function's stack frame; this is for the debugger to explain the frame chain.

Example .mask 0x80010000, -4

Tells debugger register r31 and r16 are saved; the offset from the top of the stack frame to the top of save area is -4.

■ .ent

Syntax .ent proc-name

Function Set the beginning of the procedure ***proc-name***; used to generate information for the debugger

Example .ent function_1

Marks the beginning of function_1

■ .end

Syntax .ent proc-name

Function Set the end of the procedure ***proc-name***; used to generate information for the debugger

Example .end function_1

Marks the end of function_1

■ .bss

Syntax .bss

Function Switch the destination of following statements to the ***bss*** section, which is used for data that is uninitialized anywhere

Example .bss

Switches to ***bss*** section

4.5 Sections and Relocations

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purposes. For example, there may be a "read only" section.

The linker ld reads many object files (partial programs) and combines their contents to form a runnable program. When as emits an object file, the partial program is assumed to start at address 0. ld assigns the final addresses for the partial program so that different partial programs do not overlap.

This is actually an oversimplification, but it suffices to explain how as uses sections.

ld moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called relocation. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by as has at least three sections, any of which may be empty. These are named text, data and bss sections.

as can also generate whatever other named sections you specify using the .section directive. If you do not use any directives that place output in the .text or .data sections, these sections still exist, but are empty. Within the object file, the text section starts at address 0, the data section follows, and the bss section follows the data section.

To let ld know which data changes when the sections are relocated, and how to change that data, as also writes to the object file details of the relocation needed. To perform relocation ld must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of (*address*) - (*start-address of section*)?
- Is the reference to an address "Program-Counter relative"?

In fact, every address as ever using is expressed as (*section*) + (*offset into section*).
Further, most expressions as computes have this section-relative nature. In this manual we use the notation {*secname N*} to mean "offset *N* into section *secname*."

Apart from text, data and bss sections, you need to know about the absolute section. When ld mixes partial programs, addresses in the absolute section remain unchanged. For example, address {absolute 0} is "relocated" to run-time address 0 by ld. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition,* their absolute sections must overlap. Address {absolute 239} in one part of a program is always the same address when the program is running as address {absolute 239} in any other part of the program.

The idea of sections is extended to the undefined section. Any address whose section is unknown at assembly time is by definition rendered {undefined *U*} - where *U* is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy, the word *section* is used to describe groups of sections in the linked program. ld puts all

partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

5 GNU Linker for S+core

5.1 Command Line Options

Table 5-1Linker Command Line Options

Option	Description
-e ADDR/ --entry ADDR	Set start address
-EB	Link big-endian objects (default)
-EL	Link little-endian objects
-larchive	Add archive file `archive` to the list of files to link
-Ldir	Add dir to library search path
-M	Print map file on standard output
-o NAME	Set output file name
-Tscriptfile	Read linker script
-v/--version	Print version information
--warn-common	Warn about duplicate common symbols
--fatal-warnings	Treat warnings as errors
-G gpsize	Set the maximum size of objects to be optimized using the GP register to size

Usage:

```
Score-linux-elf-ld –Texec.ld test.o –lc –L .–o test.exec
```

6 Appendix

6.1 Instruction Index

Contents	Instructions
ALU	32-bit ADD: addx ADD with Carry: addcx ADD immediate: addix ADD with Immediate shifted: addisx ADD register with immediate: addrix 16-bit ADD: add! 16-bit ADD with carry: addc! 16-bit ADD with immediate: addei!
	32-bit SUB: subx SUB with carry: subcx SUB immediate: subix SUB with immediate shifted: subisx SUB register with immediate: subrix 16-bit SUB: sub! 16-bit SUB with immediate: subei!
	32-bit NEG: negx 16-bit NEG: neg!
	32-bit CMP: cmp{TCS}.c CMP immediate: cmpi.c CMP register with zero: cmpz{TCS}.c 16-bit CMP: cmp!
	32-bit Mul: mul 32-bit mulu: mulu 32-bit div: div 32-bit divu: divu
Logical	32-bit AND: andx AND immediate: andix AND immediate shifted: andisx AND register with immediate: andrix 16-bit AND: and!
	32-bit OR: orx OR immediate: orix OR immediate shifted: orisx OR register with immediate: orrix 16-bit OR: or!
	32-bit XOR: xorx 16-bit XOR: xor!

Contents	Instructions
	32-bit NOT: notx 16-bit NOT: not!
	T<COND>: t{cond} 16-bit T<COND>: t{cond}!
	32 bit move: mv{cond} Move to control register: mtcr Move from control register: mfcr 16-bit move (r_1-r_{15}) to (r_1-r_{15}): mv! 16-bit move (r_1-r_{15}) to (r_{16}-r_{31}): mlfh! 16-bit move (r_{16}-r_{31}) to (r_1-r_{15}): mhfl!
Bit operation	Bit test: bittst.c 16-bit test: bittst!
Rotate operation	Rotate left: rolx Rotate left with carry: rolc.c Rotate left with immediate: rolix Rotate left immediate with carry: rolic.c
	Rotate right: rorx Rotate right with carry: rorc.c Rotate right with immediate: rorix Rotate right immediate with carry: roric.c
Shift operation	Shift left logical: sllx Shift left logical with immediate: sllix 16-bit shift-left logical: sll! 16-bit shift-left logical with immediate: slli!
	Shift right arithmetic: srax Shift right arithmetic with immediate: sraix 16-bit shift-right arithmetic: sra!
	Shift right: srlx Shift right with immediate: srlix 16-bit shift-right: srl! 16-bit shift-right with immediate: srli!
Extension operation	Extend signed byte: extsbx Extend unsigned byte: extzbx Extend signed half-word: extshx Extend unsigned half-word: extzhx
Jump / branch	32-bit jump: jx 16-bit jump: jx! 32-bit branch: b{cond} b{cond}l 32-bit branch with register: br{cond} br{cond}l 16-bit branch: b{cond}! 16-bit branch with register: br{cond}!

Contents	Instructions
Data processing	Load byte: lb Load byte post-index: lb Load byte pre-index: lb Load byte unsigned: lbu Load byte unsigned post-index: lbu Load byte unsigned pre-index: lbu Load half-word: lh Load half-word post-index: lh Load half-word pre-index: lh Load half-word unsigned: lhu Load half-word unsigned post-index: lhu Load half-word unsigned pre-index: lhu Load word: lw Load word post-index: lw Load word pre-index: lw 16-bit load unsigned byte: lbu! 16-bit load signed half-word: lh! 16-bit load word: lw! 16-bit load byte with bp: lbup! 16-bit load half-word with bp: lhp! 16-bit load word with bp: lwp!
	Store byte: sb Store byte post-index: sb Store byte pre-index: sb Store half-word: sh Store half-word post-index: sh Store half-word pre-index: sh Store word: sw Store word post-index: sw Store word pre-index: sw 16-bit store signed byte: sb! 16-bit store signed half-word: sh! 16-bit store signed word: sw! 16-bit store signed byte with bp: sbp! 16-bit store signed half-word bp: shp! 16-bit store signed word bp: swp! 16-bit push!: push! 16-bit pop!: pop!

Contents	Instructions
Load/ store combine	Load combined word begin: lcb Load combined word: lcw Load combined word end: lce Store combined word begin: scb Store combined word: scw Store combined word end: sce
Load immediate	Load immediate: ldi Load immediate shift: ldis 16-bit load immediate: ldiu!
Coprocessor control	Coprocessor user-defined instruction: copx Move to coprocessor data register: mtcx Move to coprocessor control register: mtccx Move from coprocessor data register: mfcx Move from coprocessor control register: mfccx Load data to coprocessor: ldcx Store data from coprocessor: stcx
Custom engine control	User-defined instruction: ceinst Move to custom engine: mtcex Move from custom engine: mfcex
Special register control	Move from special register: mfsr Move to special register: mtsr
Cache control	Cache
System Control	32-bit trap: trap{cond} 32-bit syscall: syscall sleep: sleep 32-bit debug: sdbbp 16-bit debug: sdbbp! rte: rte pflush: pflush drte: drte

6.2 Instruction Alphabet Table

Alphabet	Instruction
A	addx addcx addix andisx addrix andx andix andisx addrix add! addc! addei! and!
B	b{cond} b{cond}l b{cond}! bittst.c br{cond} br{cond}l bittst! br{cond}!
C	cache ceinst cmp{TCS}.c cmpi.c cmpz{TCS}.c copx cmp!
D	div divu drte
E	extsbx extshx extzbx extzhx
J	jx jx!

Alphabet	Instruction
L	lcb lce lcw Load byte: lb Load byte post-index: lb Load byte pre-index: lb Load byte unsigned: lbu Load byte unsigned post-index: lbu Load byte unsigned pre-index: lbu ldcx ldi ldis Load half-word: lh Load half-word post-index: lh Load half-word pre-index: lh Load half-word unsigned: lhu Load half-word unsigned post-index: lhu Load half-word unsigned pre-index: lhu Load word: lw Load word post-index: lw Load word pre-index: lw ldiu! 16 bit load unsigned byte: lbu! 16 bit load signed half-word: lh! 16 bit load word: lw! 16 bit load byte with bp: lbup! 16 bit load half-word with bp: lhp! 16 bit load word with bp: lwp!
M	mfcex mtcr mfcx mfccx mfsr mtcex mtcr mtcx mtccx mtsr mv{cond} mul mulu mlfh! mhfl!

Alphabet	Instruction
	mv!
N	negx nop notx neg! nop! not!
O	orx orix orisx orrix or!
P	pflush pop! push!
R	rolx rolix rolc.c rolic.c rorx rorix rorc.c roric.c rte
S	scb sce scw sdbbp sleep sllx sllix srax sraix srlx srlix Store byte: sb Store byte post-index: sb Store byte pre-index: sb Store half-word: sh Store half-word post-index: sh Store half-word pre-index: sh Store word: sw Store word post-index: sw

Alphabet	Instruction
	Store word pre-index: sw stcx subx subcx syscall sdbbp! sll! slli! srli! sra! srl! 16-bit store signed byte: sb! 16-bit store signed half-word: sh! 16-bit store signed word: sw! 16-bit store signed byte with bp: sbp! 16-bit store signed half-word bp: shp! 16-bit store signed word bp: swp! sub! subei!
T	t{cond} trap{cond} t{cond}
X	xorx xor!

6.3 Assembler Directive Index

GNU Assembly Directive	Description
.align	Pad the location counter (in the current subsection) to a particular storage boundary
.ascii	Insert the string into the assembly, with no null character
.asciz	Like “.ascii”, but follows the string with a zero byte.
.balign	Align the address to <power_2> bytes
.byte	Insert the byte value of the expression into the object file. Expressions are separated by comments.
.comm	Declare a common symbol named *symbol*
.data	Switch the destination of following statements into the data section of the final executable.
.equ	Set the value of symbol to expression
.extern	Specify the symbol is defined in other source file
.global	Specify the symbol to be made globally visible to all other modules that are part of executable files and visible to linker.

GNU Assembly Directive	Description
.hword	Insert the (16-bit) half-word value of the expression into the object file.
.include	Insert the contents of “filename” into the current source file. This is exactly the same as C’s use of #include
.int	Expects zero or more expressions, separated by comments. For each expression, emit a number that is the value of the expression.
.org	Advance the location counter of the current section to *new-lc*. You can’t use .org to cross sections.
.section	Start a new code or data section. Flags: b: bss section n: section is not loaded w: writable section d: data section r: read-only section x: executable section a: ignored
.set	Set the symbol to expression .set nor1 .set r1 .set nwarn: no pop warning message .set nofixdd: default .set fixdd: will insert bubbles for data dependency instructions .set volatile: will not modify the original assembly file .set optimize: will modify the original assembly file according to code optimization.
.space	This directive emits size bytes, each of value ***fill***. If the comma and ***fill*** are omitted, ***fill*** is assumed to be zero.
.text	Switch the destination of following statements into the text section
.word	Insert the (32-bit) word value of expression into the object file

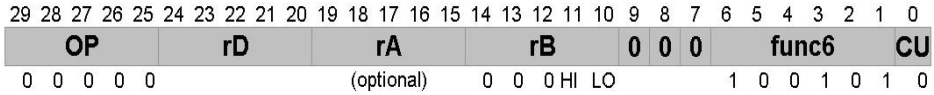
6.4 Enhanced MAC Instruction

6.4.1 32-Bit Instruction

mtcex

Function Move Data to Custom Engine Register

Form



Syntax

```
mtcel   rD          (HI = 0, LO = 1, CU = 0)
mtceh   rD          (HI = 1, LO = 0, CU = 0)
mtcehl  rD, rA      (HI = 1, LO = 1, CU = 0)
```

where

<rD> Specifies the source general-purpose register.

<rA> Specifies the second source general-purpose register.

Operation

```
If ({HI, LO} == 2'b01)
    CEL = GPR[rD];
Else if ({HI, LO} == 2'b10)
    CEH = GPR[rD];
Else if ({HI, LO} == 2'b11)
{
    CEH = GPR[rD];
    CEL = GPR[rA];
}
else
    Undefine;
```

Usage

```
mtcehl   r4, r2
```

Description

The *mtcex* instruction moves data from general-purpose register to custom engine registers CEH/CEL according to the HI and LO bits in the instruction form. HI bit specifies the data

transmission to CEH register while LO bit specifies that to CEL register. **Table** 2-5 lists the instruction name, HI and LO bits encoding and the corresponding operation of *mfcex* and *mtcex* instructions. The *mtcex* instruction can also transfer data to CEH and CEL registers simultaneously; rA is optional for *mfcex* and *mtcex* instructions.

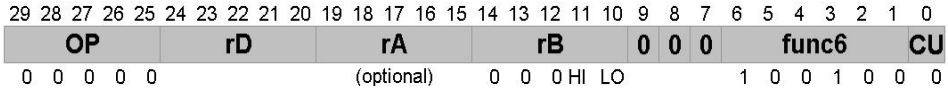
Exceptions

None

mfcex

Function Move Data from Custom Engine Register

Form



Syntax

```
mfcel    rD          (HI = 0, LO = 1, CU = 0)
mfceh    rD          (HI = 1, LO = 0, CU = 0)
mfcehl   rD, rA      (HI = 1, LO = 1, CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the second destination general-purpose register.

Operation

```
If ({HI, LO} == 2'b01)
     GPR[rD] = CEL;
Else if ({HI, LO} == 2'b10)
     GPR[rD] = CEH;
Else if ({HI, LO} == 2'b11)
{
     GPR[rD] = CEH;
     GPR[rA] = CEL;
}
else
     Undefine;
```

Usage

```
mfcehl   r4, r2
```

Description

The mfcex instruction moves data from custom engine register CEH/CEL to general-purpose register according to the HI and LO bits in the instruction form. HI bit specifies the data transmission to CEH register while LO bit specifies that to CEL register. The mfcex instruction can transfer data from CEH and CEL registers simultaneously; rA is optional for mfcex instruction. Table 2-5 lists the

instruction name, HI and LO bits encoding and the corresponding operation of mfcex and mtcex instructions.

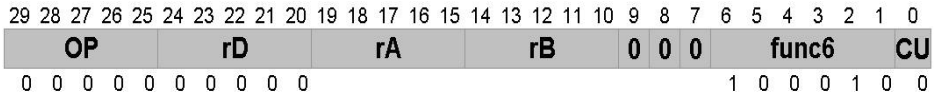
Exceptions

None

div

Function Divide

Form



Syntax

```
div     rA, rB          (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

$CEL = Q(GPR_{rA} / GPR_{rB}),$

$CEH = R(GPR_{rA} / GPR_{rB})$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
div     r2, r3
```

Description

The *div* instruction performs a division on the content of general-purpose register rA. The dividend is the signed value of the content of rA and the divisor is the signed value of the content of general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The custom engine execution exception is induced when the divisor is zero; consequently, the result of this operation is undefined. The quotient word of the divided result is placed in custom engine register CEL; and the remainder word of the divided result is placed in custom engine register CEH.

The move from custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the contents of custom engine registers CEH and CEL into general-purpose registers.

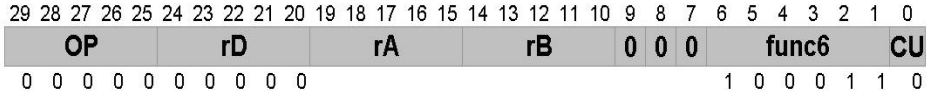
Exceptions

Custom engine execution exception: Divided by zero

divu

Function Divide Unsigned

Form



Syntax

```
divu    rA, rB          (CU = 0)
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

$CEL = Q(GPR_{rA} / GPR_{rB}),$

$CEH = R(GPR_{rA} / GPR_{rB})$

(GPR_{rA}, GPR_{rB} are treated as unsigned)

Usage

```
divu    r2, r3
```

Description

The *divu* instruction performs a division on the content of general-purpose register rA. The dividend is the unsigned value of the content of rA; and the divisor is the unsigned value of the content of general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The custom engine execution exception is induced when the divisor is zero; consequently, the result of this operation is undefined. The quotient word of the divided result is placed in custom engine register CEL, and the remainder word of the divided result is placed in custom engine register CEH. The move from custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the contents of custom engine register CEH and CEL into general-purpose registers.

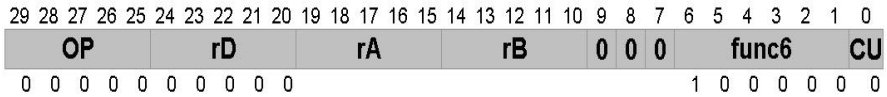
Exceptions

Custom engine execution exception: Divided by zero.

mul

Function Multiply (Signed)

Form



Syntax

```
mul     rA, rB          (CU = 0)
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{HI,LO\} = GPR_{rA} * GPR_{rB}$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mul     r2, r3
```

Description

The *mul* instruction performs a signed multiplication of general-purpose register rA and rB. The multiplicand is the signed value in general-purpose register rA; and the multiplier is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiplication result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mul.f

Function Multiply (Signed, Fractional)

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
OP					rD					rA					rB					0	0	0	func6						CU
0	0	0	0	0	0	0	0	0	0											0	0	0	1	0	0	0	0	0	1

Syntax

```
mul.f    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{CEH, CEL\} = GPR_{rA} * GPR_{rB}$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mul.f    r2, r3
```

Description

The *mul.f* instruction performs a multiplication of general-purpose register rA and rB. The multiplicand is the signed-fractional value in general-purpose register rA; and the multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's -complement values. The low word of the multiplication result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

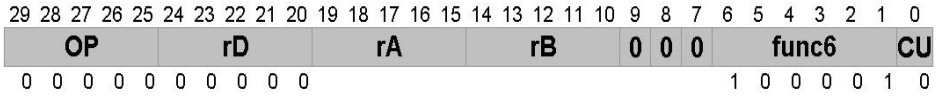
Exceptions

None

mulu

Function Multiply (Unsigned)

Form



Syntax

```
mulu    rA, rB          (CU = 0)
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{HI,LO\} = GPR_{rA} * GPR_{rB}$

(GPR_{rA}, GPR_{rB} are treated as unsigned)

Usage

```
mulu    r2, r3
```

Description

The mulu instruction performs a multiplication of general-purpose register rA and rB. The multiplicand is the unsigned value in general-purpose register rA; and the multiplier is the unsigned value in general-purpose register rB. The low word of the multiplication result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mad

Function Multiply-Add (Signed)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	0	0	0	0	0	0

Syntax

```
mad     rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} + GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mad     r2, r3
```

Description

The *mad* instruction adds custom engine registers {CEH, CEL} to the multiplication of general-purpose register rA and rB. The multiplicand is the signed value in general-purpose register rA. The multiplier is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's -complement values. The low word of the multiply-add result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madu

Function Multiply-Add (Unsigned)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	1	0	0	0	0	0

Syntax

```
madu    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} + GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPR_{rB} are treated as unsigned)

Usage

```
madu    r2, r3
```

Description

The *madu* instruction adds custom engine registers {CEH, CEL} to the multiplication of general-purpose register rA and rB. The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB. The low word of the multiply-add result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mad.f

Function Multiply-Add (Signed, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	1	0	0	0	0	0	0	0

Syntax

```
mad.f    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} + GPR_{rA} * GPR_{rB} ,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mad.f    r2, r3
```

Description

The *mad.f* instruction adds custom engine registers {CEH, CEL} to the multiplication of general-purpose register rA and rB. The multiplicand is the signed-fractional value in general-purpose register rA. The multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply-add result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msb

Function Multiply-Sub (Signed)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	0	0	0	0	0	1

Syntax

```
msb     rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} - GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
msb     r2, r3
```

Description

The *msb* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine registers {CEH, CEL}. The multiplicand is the signed value in general-purpose register rA. The multiplier is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply-sub result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbu

Function Multiply-Sub (Unsigned)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	1	0	0	0	0	1

Syntax

```
msbu    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} - GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPR_{rB} are treated as unsigned)

Usage

```
msbu    r2, r3
```

Description

The *msbu* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine registers {CEH, CEL}. The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB. The low word of the multiply-sub result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msb.f

Function Multiply-Sub (Signed, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	1	0	0	0	0	0	0	1

Syntax

```
msb.f    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} - GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
msb.f    r2, r3
```

Description

The *msb.f* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine registers {CEH, CEL}. The multiplicand is the signed-fractional value in general-purpose register rA. The multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply-sub result is placed in custom engine register CEL, and the high word is placed in custom engine register CEH.

The custom engine instructions such as *mfceh*, *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mazl

Function Multiply-Add (Lower Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	1	0	0	0	0	0	0	1	0

Syntax

```
mazl    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = 0 + GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mazl    r2, r3
```

Description

The multiplicand is the low 16-bit signed value in general-purpose register rA. The multiplier is the low 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mazl.f

Function Multiply-Add (Lower Part, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	1	1	0	0	0	0	0	1	0

Syntax

```
mazl.f   rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = 0 + GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mazl.f   r2, r3
```

Description

The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfce*l or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madl

Function Multiply-Add (Lower Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	0	0	0	0	1	0

Syntax

```
madl    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = CEL + GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
madl    r2, r3
```

Description

The *madl* instruction adds custom engine registers CEL to the multiplication of general-purpose register rA and rB. The multiplicand is the low 16-bit signed value in general-purpose register rA. The multiplier is the low 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madl.fs

Function Multiply-Add (Lower Part, Fractional, Saturation)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	1	1	0	0	0	0	1	0

Syntax

```
madl.fs  rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEL = CEL + GPR_{rA}[15:0] * GPR_{rB}[15:0]$,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
madl.fs  r2, r3
```

Description

The *madl.fs* instruction adds custom engine registers CEL to the multiplication of general-purpose register rA and rB, and then clamps the result to boundary value when overflow occurs. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mszl

Function Multiply-Sub (Lower Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	1	0	0	0	0	0	1	0	0

Syntax

```
mszl    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = 0 - GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mszl    r2, r3
```

Description

The *mszl* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the low 16-bit signed value in general-purpose register rA. The multiplier is the low 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mszl.f

Function Multiply-Sub (Lower Part, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	1	1	0	0	0	0	1	0	0

Syntax

```
mszl.f   rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

CEL = 0 - GPR_{rA}[15:0] * GPR_{rB}[15:0] ,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mszl.f   r2, r3
```

Description

The *mszl.f* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbl

Function Multiply-Sub (Lower Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	0	0	0	0	0	1	0	0

Syntax

```
msbl    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

CEL = CEL - GPR_{rA}[15:0] * GPR_{rB}[15:0] ,

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
msbl    r2, r3
```

Description

The *msbl* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine register CEL. The multiplicand is the low 16-bit signed value in general-purpose register rA. The multiplier is the low 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfce*l or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbl.fs

Function Multiply-Sub (Lower Part, Fractional, Saturation)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											0	0	1	1	0	0	0	1	0	0

Syntax

```
msbl.fs  rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = CEL - GPR_{rA}[15:0] * GPR_{rB}[15:0]$,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
msbl.fs  r2, r3
```

Description

The *msbl.fs* instruction subtracts the multiplication of general-purpose register rA and rB from the custom engine register CEL, and then clamps the result to boundary value when overflow occurs. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mazh

Function Multiply- Add (Higher Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	1	0	0	0	0	0	0	1	1

Syntax

```
mazh    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEH = 0 + GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mazh    r2, r3
```

Description

The multiplicand is the high order 16-bit signed value in general-purpose register rA. The multiplier is the high order 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mazh.f

Function Multiply-Add (Higher Part, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	1	1	0	0	0	0	0	1	1

Syntax

```
mazh.f   rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEH = 0 + GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mazh.f   r2, r3
```

Description

The multiplicand is the high order 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high order 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madh

Function Multiply-Add (Higher Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	0	0	0	0	0	0	0	1	1

Syntax

```
madh    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEH = CEH + GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
madl    r2, r3
```

Description

The *madh* instruction adds custom engine register CEH to the multiplication of general-purpose register rA and rB. The multiplicand is the high 16-bit signed value in general-purpose register rA. The multiplier is the high 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madh.fs

Function Multiply-Add (Higher part, Fractional, Saturation)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	0	1	1	0	0	0	0	1	1

Syntax

```
madh.fs  rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEH = CEH + GPR_{rA}[31:16] * GPR_{rB}[31:16]$,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
madh.fs  r2, r3
```

Description

The *madh.fs* instruction adds custom engine register CEH to the multiplication of general-purpose register rA and rB, and then clamps the result to boundary value when overflow occurs. The multiplicand is the high 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH. The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mszh

Function Multiply-Sub (Higher Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	1	0	0	0	0	0	1	0	1

Syntax

```
mszh    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEH = 0 - GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
mszh    r2, r3
```

Description

The *mszh* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the high 16-bit signed value in general-purpose register rA. The multiplier is the high 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's -complement values. The multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mszh.f

Function Multiply-Sub (Higher Part, Fractional)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	1	1	0	0	0	0	1	0	1

Syntax

```
mszh.f   rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

CEL = 0 - GPR_{rA}[31:16] * GPR_{rB}[31:16] ,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mszh.f   r2, r3
```

Description

The *mszh.f* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the high 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbh

Function Multiply-Sub (Higher Part)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	0	0	0	0	0	0	1	0	1

Syntax

```
msbh    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEH = CEH - GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed)

Usage

```
msbh    r2, r3
```

Description

The *msbh* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine register CEH. The multiplicand is the high 16-bit signed value in general-purpose register rA. The multiplier is the high 16-bit signed value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbh.fs

Function Multiply-Sub (Higher Part, Fractional, Saturation)

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
OP					0	0	0	0	0	rA					rB					H	Z	F	S	U	func5					
1	1	1	0	0	0	0	0	0	0											1	0	1	1	0	0	0	1	0	1

Syntax

```
msbh.fs  rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEH = CEH - GPR_{rA}[31:16] * GPR_{rB}[31:16]$,

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
msbh.fs  r2, r3
```

Description

The *msbh.fs* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine register CEH, and then clamps the result to boundary value when overflow occurs. The multiplicand is the high 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh* or *mfcehl* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

min

Function Minimum

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	0	0	0	0	1	1	0

Syntax

```
min     rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

```
GPR[rD] = minimum(GPR[rA], GPR[rB]);
If (rA <= rB)
    rD = rA;
Else
    rD = rB;
```

Usage

```
min     r4, r3, r2
```

Description

The *min* instruction selects the minimum content of general-purpose register rA and rB, and stores the result to general-purpose register rD.

Exceptions

None

max

Function Maximum

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	0	0	0	0	1	1	1

Syntax

```
max     rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

```
GPR[rD] = maximum(GPR[rA], GPR[rB]);
If (rA >= rB)
    rD = rA;
Else
    rD = rB;
```

Usage

```
max     r4, r3, r2
```

Description

The *max* instruction selects the maximum content of general-purpose register rA and rB, and stores the result to general-purpose register rD.

Exceptions

None

add.s

Function ADD with Saturation

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	1	0	0	1	0	0	0

Syntax

```
add.s    rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

GPR_{rD} = saturation(GPR_{rA} + GPR_{rB});

Usage

```
add.s        r4, r3, r2
```

Description

The *add.s* instruction adds the contents of general-purpose register rA and rB, and then clamps the result to boundary value when overflow occurs, and stores the result to general-purpose register rD.

Exceptions

None

sub.s

Function Subtract with Saturation

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	1	0	0	1	0	0	1

Syntax

```
sub.s    rD, rA, rB
```

where

<rD>	Specifies the destination general-purpose register.
<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

GPR_{rD} = saturation(GPR_{rA} - GPR_{rB});

Usage

```
sub.s        r4, r2, r3
```

Description

The *sub.s* instruction subtracts general-purpose register rB from rA, and then clamps the result to boundary value when overflow occurs, and stores the result to general-purpose register rD.

Exceptions

None

abs

Function Absolute

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
OP					rD					rA					0	0	0	0	0	0	0	0	S	0	func5					
1	1	1	0	0											0	0	0	0	0	0	0	0	0	0	0	1	0	1	0

Syntax

```
abs     rD, rA
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

```
GPR[rD] = absolute(GPR[rA]);
If (rA >= 0)
    rD = rA;
Else
    rD = 0 - rA;
```

Usage

```
abs     r4, r3
```

Description

The abs instruction gets the absolute value of the content of general-purpose register rA, and stores the result to general-purpose register rD.

Exceptions

None

abs.s

Function Absolute with Saturation

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
OP					rD					rA					0	0	0	0	0	0	0	0	S	0	func5					
1	1	1	0	0											0	0	0	0	0	0	0	0	1	0	0	1	0	1	1

Syntax

```
abs.s    rD, rA
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

```
GPR_rD = saturation (absolute(GPR_rA) );
If (rA >= 0)
    rD = rA;
Else
    rD = saturation ( 0 – rA );
```

Usage

```
abs.s    r4, r3
```

Description

The abs.s instruction gets the absolute value of the content of general-purpose register rA, and then clamps the result to boundary value when overflow occurs, and stores the result to general-purpose register rD.

Exceptions

None

bitrev

Function Bit Reverse with Shift Right Logical

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	0	0	0	1	1	0	0

Syntax

```
bitrev   rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register (shift amount).

Operation

```
SA[5:0]  :   Shift amount
Rev(GPR_rA)   :   bit-reverse content of general-purpose register rA
If (GPR_rB == r0)
    SA = 0;
Else
    SA = GPR_rB[4:0];
Rev(GPR_rA)[31:0] = {GPR_rA[0:31]};
GPR_rD = { SA{0}, Rev(GPR_rA)[31:SA]};
```

Usage

```
bitrev   r4, r2, r3
```

Description

If the general-purpose register rB is general-purpose register r0, the shift amount is zero, else the shift amount is the lower 5-bit of register rB. The *bitrev* instruction right shifts the bit-reverse content of general-purpose register rA with zero insertion by the shift amount, and stores this result to general-purpose register rD.

Exceptions

None

clz

Function Count Leading Zero

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0											0	0	0	0	0	0	0	0	0	0	0	1	1	0	1

Syntax

```
clz     rD, rA
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

Operation

GPR_{rD} = CLZ (GPR_{rA});

Usage

```
clz     r4, r2
```

Description

The *clz* instruction counts the leading zero bit of the content of general-purpose register rA, and stores this result to general-purpose register rD.

Exceptions

None

sll.s

Function Shift Left Logical with Saturation

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
OP					rD					rA					rB					0	0	0	S	0	func5				
1	1	1	0	0																0	0	0	1	0	0	1	1	1	0

Syntax

```
sll.s    rD, rA, rB
```

where

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<rB> Specifies the second source general-purpose register (shift amount).

Operation

GPR_{rD} = saturation ({GPR_{rA}[(31 - GPR_{rB}[4:0]):0], GPR_{rB}[4:0]{0}});

Usage

```
sll.s    r4, r2, r3
```

Description

The *sll.s* instruction left shifts the content of general-purpose register rA with zero insertion by the shift amount, the lower 5-bit of register rB, and then clamps the result to boundary value when overflow occurs, and stores this result to general-purpose register rD.

Exceptions

None

6.4.2 16-Bit Instruction

mtcel!

Function Move to Custom Engine Register CEL

Form

14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
OP	sub-fun	rA	func4
0 0 1	0 0 0 0	rA_{g0}	0 0 0 0

Syntax

```
mtcel!   rA
```

where

<rA> Specifies the source general-purpose register.

Operation

$CEL = GPR_{rA}$;

Usage

```
mtcel!   r4
```

Description

The *mtcel!* instruction moves the content of general-purpose register to custom engine register CEL.

Exceptions

None

mtceh!

Function Move to Custom Engine Register CEH

Form

14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
OP	sub-fun	rA	func4
0 0 1	0 0 0 1	rA_{g0}	0 0 0 0

Syntax

```
mtceh!   rA
```

where

<rA> Specifies the source general-purpose register.

Operation

$CEH = GPR_{rA}$;

Usage

```
mtceh!   r4
```

Description

The *mtceh!* instruction moves the content of general-purpose register to custom engine register CEH.

Exceptions

None

mfcel!

Function Move from Custom Engine Register CEL

Form

14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
OP	sub-fun	rA	func4
0 0 1	0 0 0 0	rA_{g0}	0 0 0 1

Syntax

```
mfcel!   rA
```

where

<rA> Specifies the destination general-purpose register.

Operation

GPR_{rA} = CEL;

Usage

```
mfcel!   r2
```

Description

The *mfcel!* instruction moves the content of custom engine register CEL to general-purpose register.

Exceptions

None

mfceh!

Function Move from Custom Engine Register CEH

Form

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP			sub-fun				rA				func4			
0	0	1	0	0	0	1	rA_{g0}				0	0	0	1

Syntax

```
mfceh!   rA
```

where

<rA> Specifies the destination general-purpose register.

Operation

GPR_{rA} = CEH;

Usage

```
mfceh!   r2
```

Description

The *mfceh!* instruction moves the content of custom engine register CEH to general-purpose register.

Exceptions

None

mul.f!

Function Multiply (Signed, Fractional)

Form

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

Syntax

```
mul.f!   rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{CEH, CEL\} = GPR_{rA} * GPR_{rB}$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mul.f!   r2, r3
```

Description

The *mul.f!* instruction performs a multiplication of general-purpose register rA and rB. The multiplicand is the signed-fractional value in general-purpose register rA. The multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply result is placed in custom engine register CEL, and the high order word of the multiply result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

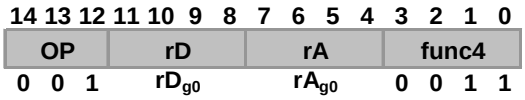
Exceptions

None

mulu!

Function Multiply (Unsigned)

Form



Syntax

```
mulu!    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = GPR_{rA} * GPR_{rB}

(GPR_{rA}, GPRrB are treated as unsigned)

Usage

```
mulu!    r2, r3
```

Description

The *mulu!* instruction performs a multiplication of general-purpose register rA and rB. The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB. The low word of the multiply result is placed in custom engine register CEL, and the high order word of the multiply result is placed in custom engine register CEH. The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mad.f!

Function Multiply-Add (Signed, Fractional)

Form

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

Syntax

```
mad.f!   rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{CEH,CEL\} = \{CEH,CEL\} + GPR_{rA} * GPR_{rB}$

($GPR_{rA,}$ GPR_{rB} are treated as signed-fractional)

Usage

```
mad.f!   r2, r3
```

Description

The *mad.f!* instruction adds custom engine registers {CEH, CEL} to the multiplication of general-purpose register rA and rB. The multiplicand is the signed-fractional value in general-purpose register rA. The multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply-add result is placed in custom engine register CEL, and the high word of the multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

madu!

Function Multiply-Add (Unsigned)

Form

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

Syntax

```
madu!    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{CEH,CEL\} = \{CEH,CEL\} + GPR_{rA} * GPR_{rB}$

($GPR_{rA,}$ GPR_{rB} are treated as unsigned)

Usage

```
madu!    r2, r3
```

Description

The *madu!* instruction adds custom engine registers {CEH, CEL} to the multiplication of general-purpose register rA and rB. The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB. The low word of the multiply-add result is placed in custom engine register CEL, and the high word of the multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msb.f!

Function Multiply-Sub (Signed, Fractional)

Form

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

Syntax

```
msb.f!   rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$\{CEH,CEL\} = \{CEH,CEL\} - GPR_{rA} * GPR_{rB}$

($GPR_{rA,}$ GPR_{rB} are treated as signed-fractional)

Usage

```
msb.f!   r2, r3
```

Description

The *msb.f!* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine registers {CEH, CEL}. The multiplicand is the signed-fractional value in general-purpose register rA. The multiplier is the signed-fractional value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low word of the multiply-sub result is placed in custom engine register CEL, and the high word of the multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

msbu!

Function Multiply-Sub (Unsigned)

Form

OP	rD	rA	func4
0 0 1	rD_{g0}	rA_{g0}	0 1 1 1

Syntax

```
msbu!    rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

{CEH,CEL} = {CEH,CEL} - GPR_{rA} * GPR_{rB}

($GPR_{rA,}$ GPR_{rB} are treated as unsigned)

Usage

```
msbu!    r2, r3
```

Description

The *msbu!* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine registers {CEH, CEL}. The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB. The low word of the multiply-sub result is placed in custom engine register CEL, and the high order word of the multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* or *mfcel!* can move the result from custom engine register CEH or CEL into general-purpose registers.

Exceptions

None

mazl.f!

Function Multiply-Add (Lower Part, Fractional)

Form

OP	rD	rA	func4
0 0 1	rD_{g0}	rA_{g0}	1 0 0 0

Syntax

```
mazl.f!  rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = 0 + GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mazl.f!  r2, r3
```

Description

The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel!* can move the result from custom engine register CEL into general-purpose registers.

Exceptions

None

mazh.f!

Function Multiply-Add (Higher part, Fractional)

Form

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

Syntax

```
mazh.f!  rA, rB
```

where

`<rA>` Specifies the first source general-purpose register.

`<rB>` Specifies the second source general-purpose register.

Operation

$CEH = 0 + GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mazh.f!  r2, r3
```

Description

The multiplicand is the high order 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high order 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* can move the result from custom engine register CEH into general-purpose registers.

Exceptions

None

madl.fs!

Function Multiply-Add (Lower part, Fractional, Saturation)

Form

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

Syntax

```
madl.fs! rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = CEL + GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
madl.fs! r2, r3
```

Description

The *madl.fs!* instruction adds custom engine register CEL to the multiplication of general-purpose register rA and rB, and then clamps the result to boundary value when overflow occurs. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel!* can move the result from custom engine register CEL into general-purpose registers.

Exceptions

sNone

madh.fs!

Function Multiply-Add (Higher Part, Fractional, Saturation)

Form

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

Syntax

```
madh.fs! rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEH = CEH + GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
madh.fs! r2, r3
```

Description

The *madh.fs!* instruction adds custom engine register CEH to the multiplication of general-purpose register rA and rB, and then clamps the result to boundary value when overflow occurs. The multiplicand is the high order 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high order 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-add result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* can move the result from custom engine register CEH into general-purpose registers.

Exceptions

None

mszl.f!

Function Multiply-Sub (Lower Part, Fractional)

Form

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

Syntax

```
mszl.f!  rA, rB
```

where

<rA> Specifies the first source general-purpose register.

<rB> Specifies the second source general-purpose register.

Operation

$CEL = 0 - GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mszl.f!  r2, r3
```

Description

The *mszl.f!* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel!* can move the result from custom engine register CEL into general-purpose registers.

Exceptions

None

mszh.f!

Function Multiply-Sub (Higher Part, Fractional)

Form

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

Syntax

```
mszh.f!  rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEL = 0 - GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
mszh.f!  r2, r3
```

Description

The *mszh.f!* instruction subtracts the multiplication of general-purpose register rA and rB from zero. The multiplicand is the high order 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high order 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfceh!* can move the result from custom engine register CEH into general-purpose registers.

Exceptions

None

msbl.fs!

Function Multiply-Sub (Lower Part, Fractional, Saturation)

Form

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

Syntax

```
msbl.fs! rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEL = CEL - GPR_{rA}[15:0] * GPR_{rB}[15:0]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
msbl.fs! r2, r3
```

Description

The *msbl.fs!* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine register CEL, and then clamps the result to boundary value when overflow occurs. The multiplicand is the low 16-bit signed-fractional value in general-purpose register rA. The multiplier is the low 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEL.

The custom engine instructions such as *mfcel!* can move the result from custom engine register CEL into general-purpose registers.

Exceptions

None

msbh.fs!

Function Multiply-Sub (Higher Part, Fractional, Saturation)

Form

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

Syntax

```
msbh.fs! rA, rB
```

where

<rA>	Specifies the first source general-purpose register.
<rB>	Specifies the second source general-purpose register.

Operation

$CEH = CEH - GPR_{rA}[31:16] * GPR_{rB}[31:16]$

(GPR_{rA}, GPR_{rB} are treated as signed-fractional)

Usage

```
msbh.fs! r2, r3
```

Description

The *msbh.fs!* instruction subtracts the multiplication of general-purpose register rA and rB from custom engine register CEH, and then clamps the result to boundary value when overflow occurs. The multiplicand is the high order 16-bit signed-fractional value in general-purpose register rA. The multiplier is the high order 16-bit signed-fractional value in general-purpose register rB. Both operands are treated as 16-bit 2's-complement values. The multiply-sub result is placed in custom engine register CEH.

The custom engine instructions such as *mfceh!* can move the result from custom engine register CEH into general-purpose registers.

Exceptions

None

6.5 Data Dependency Rules

Required software bubble means the case that users must insert sufficient bubbles (nop) for preventing from unexpected program error. In the following required software bubble table, the numbers of green grids represent the required software bubbles.

[Memo]: Currently S+core tool will generate the following warning messages for almost all required software bubbles rule except tlb/ cache data dependency, which tool can't foresee if the program will fetch new page when it is dynamically run.

```
Warning: data dependency warning : mtcr cr0 -- li   (5 bubble)
Warning: data dependency warning : mtcr cr0 -- mtcr cr3  (4 bubble)
Warning: data dependency warning : mtcr cr0 -- li   (3 bubble)
Warning: data dependency warning : mtcr cr0 -- cache   (2 bubble)
Warning: data dependency warning : mtcr cr0 -- mfcr cr4  (1 bubble)
```

Recommend software bubble means the case that cpu will automatically judge and processing stall. If user's coding meets the recommend software bubble rule, the program will not go wrong. But if user can take care and prevent from resulting recommend software bubbles, the program's performance and efficiency can be improved. In the following recommend software bubble table, the numbers of green grids represent the stall cycles that hardware process.

6.5.1 Required Software Bubbles

In the following, we will introduce all the required software bubbles cases , and use some simple examples to show in which cases tool will generate what kind of warning messages.

instruction	pipeline stages											forwarded reg
Required Software Bubble												
special register												
condition register												
mtcr(cond_reg)	I	D	S	E	A	M	W					cond reg (T)
bubble		I	D	S	E	A	M	W				
bubble			I	D	S	E	A	M	W			
pce				I	D	S	E	A	M	W		T flag
mtcr(cond_reg)	I	D	S	E	A	M	W					cond reg (NZCV)
bubble		I	D	S	E	A	M	W				
conditional branch			I	D	S	E	A	M	W			NZCV flag

Example:

```
mtcr        r14, cr1
bcc label
```

Tool will generate warning message

```
Warning: data dependency warning : mtcr cr1 -- bcc   (1 bubble)
```

〔Memo〕Conditional Branch instructions :

b{cond} / br{cond}/ b{cond}l / br{cond}l / b{cond}!/ br{cond}!

mv{cond}

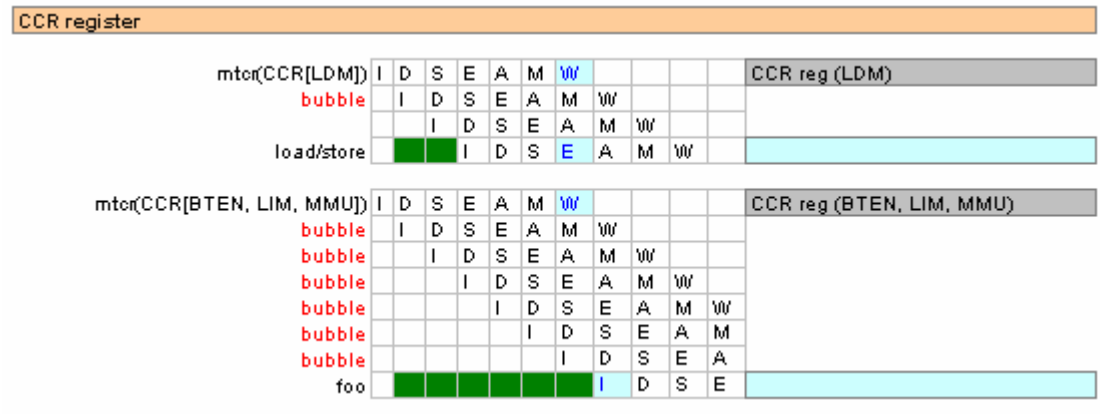
Example:

```
mtcr   r4, cr0
mv     r1,r4
```

Tool will generate warning message

```
Warning: data dependency warning : mtcr cr0 -- mv   (5 bubble)
```

〔Memo〕foo means all instructions



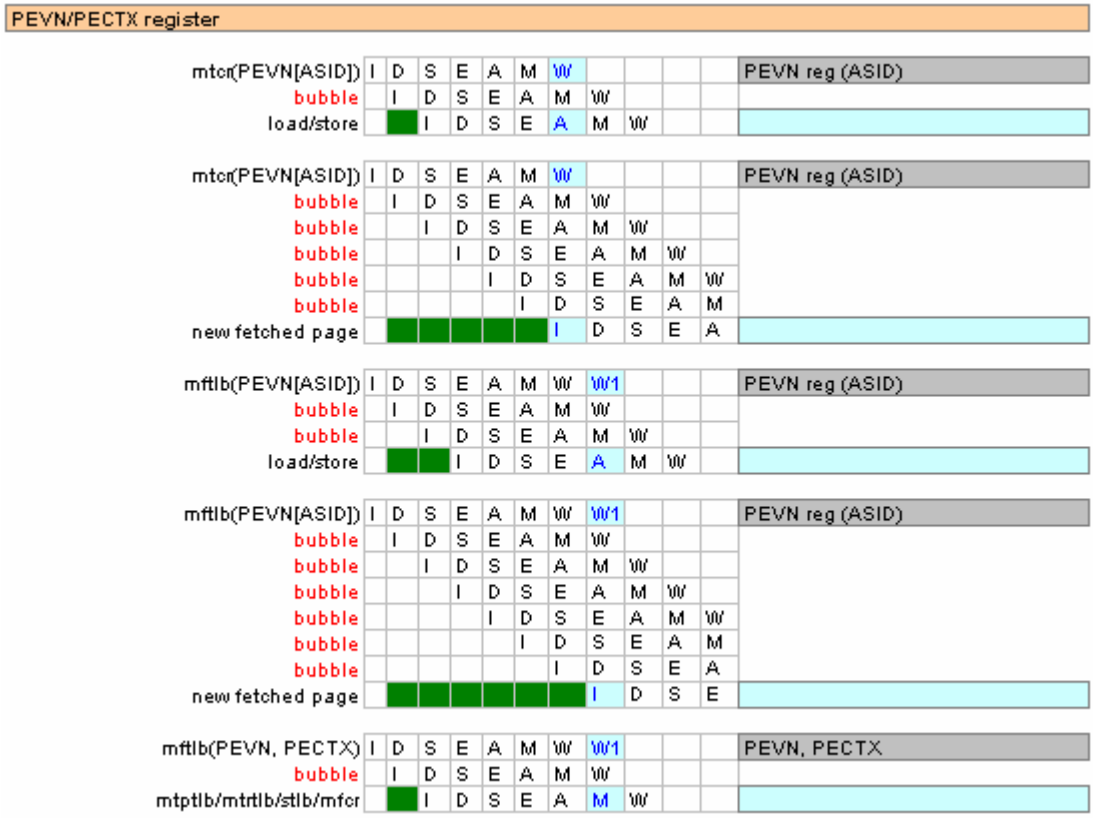
Example:

```
mtcr   r8, cr4
mv     r1,r4
```

Tool will generate warning message

```
data dependency warning : mtcr cr4 -- mv   (6 bubble)
```

〔Memo〕foo means all instructions

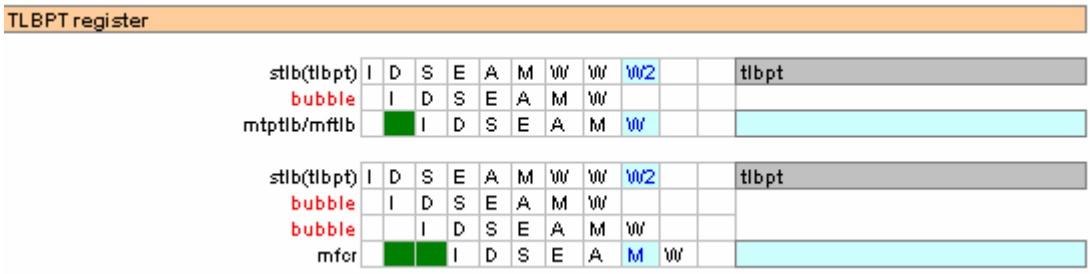


Example:

```
mftlb
mfcr r15,cr11
```

Tool will generate warning message

```
Warning: data dependency warning : mftlb  -- mfcr cr11  (1 bubble)
```

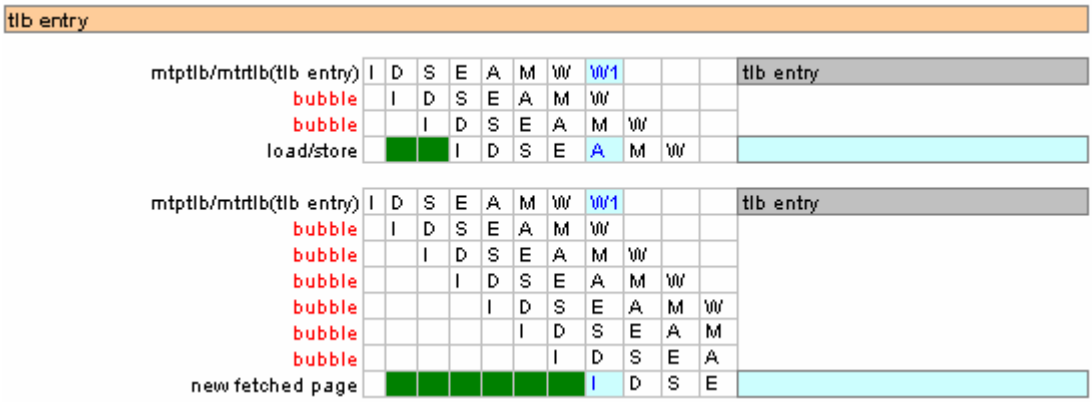


Example:

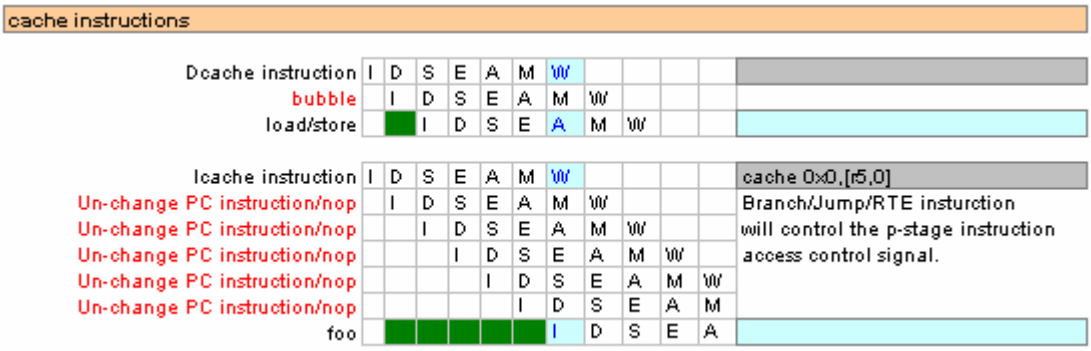
```
stlb
mfcr r5,cr8
```

Tool will generate warning message

```
Warning: data dependency warning : stlb  -- mfcr cr8  (2 bubble)
```



〔Memo〕For now on , S+coreTool will not generate warning message for this case.



Example:

/*d cache instruction*/

```
cache 8,[r4]
lw r9,[r3]
```

Tool will generate warning message

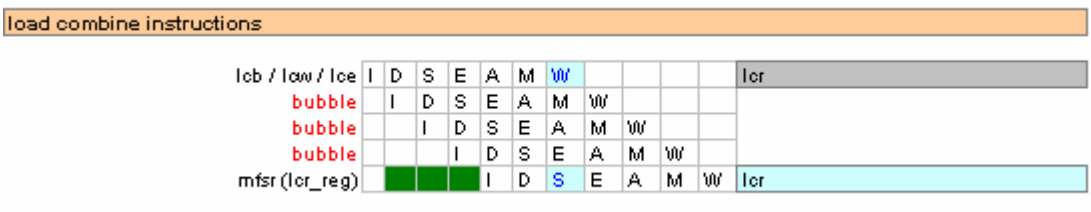
```
Warning: data dependency warning : cache 8 -- lw  (1 bubble)
```

/*i cache instruction*/

```
cache 0,[r4]
mv r1,r5
```

Tool will generate warning message

```
Warning: data dependency warning : cache 0 -- mv  (5 bubble)
```



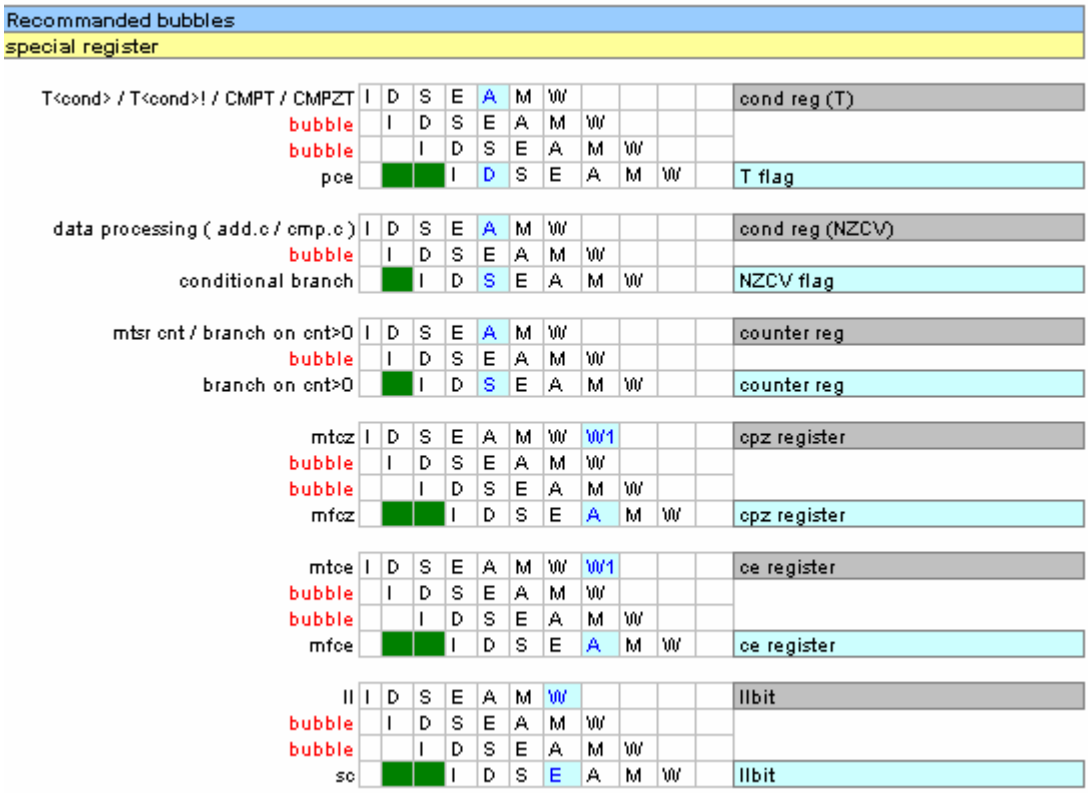
Example:

```
lcb [r3]+
mfsr r7,sr1
```

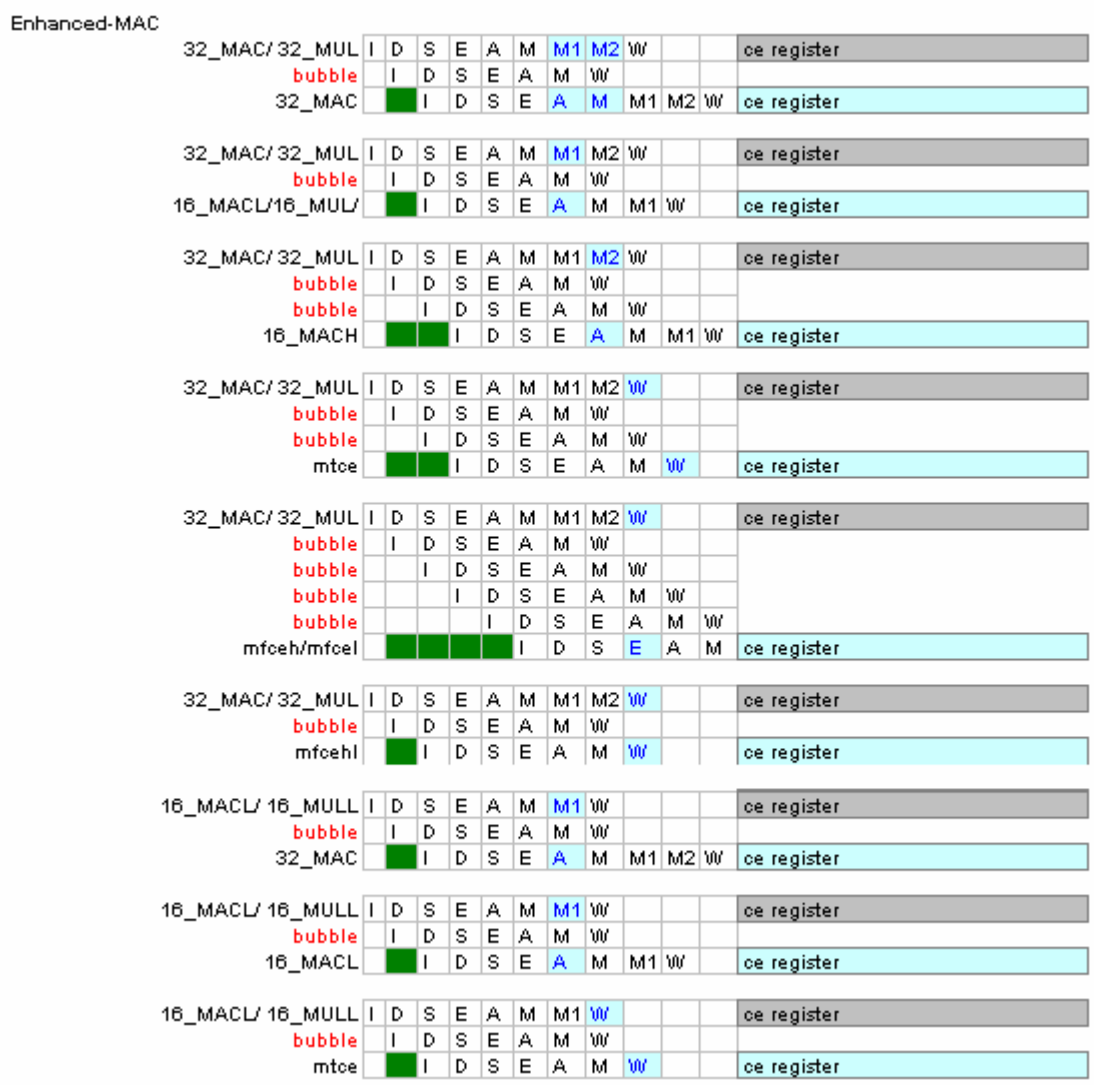
Tool will generate warning message

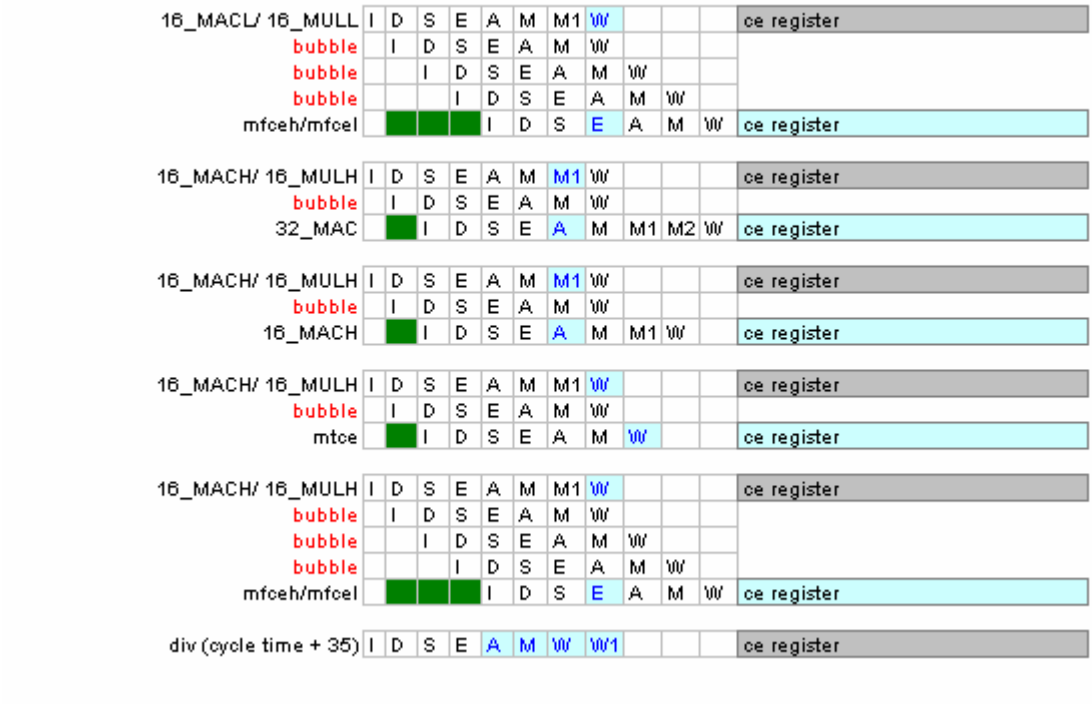
```
Warning: data dependency warning : lcb  -- mfsr sr1  (3 bubble)
```

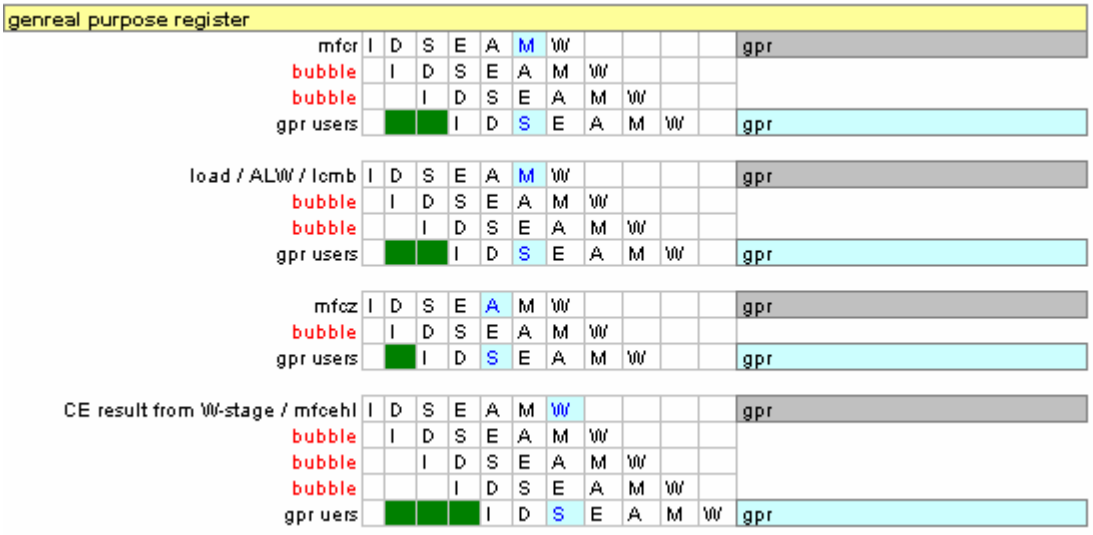
6.5.2 Recommended Bubbles



Enhanced-MAC




16_MACL/ 16_MULL I D S E A M M1 W ce register
bubble I D S E A M W
bubble I D S E A M W
bubble I D S E A M W
mfceh/mfcel I D S E A M W ce register
16_MACH/ 16_MULH I D S E A M M1 W ce register
bubble I D S E A M W
32_MAC I D S E A M M1 M2 W ce register
16_MACH/ 16_MULH I D S E A M M1 W ce register
bubble I D S E A M W
16_MACH I D S E A M M1 W ce register
16_MACH/ 16_MULH I D S E A M M1 W ce register
bubble I D S E A M W
mtce I D S E A M W ce register
16_MACH/ 16_MULH I D S E A M M1 W ce register
bubble I D S E A M W
bubble I D S E A M W
bubble I D S E A M W
mfceh/mfcel I D S E A M W ce register
div (cycle time + 35) I D S E A M W W1 ce register


genreal purpose register
mfcr I D S E A M W gpr
bubble I D S E A M W
bubble I D S E A M W
gpr users I D S E A M W gpr
load / ALW / lcmb I D S E A M W gpr
bubble I D S E A M W
bubble I D S E A M W
gpr users I D S E A M W gpr
mfcz I D S E A M W gpr
bubble I D S E A M W
gpr users I D S E A M W gpr
CE result from W-stage / mfcehl I D S E A M W gpr
bubble I D S E A M W
bubble I D S E A M W
bubble I D S E A M W
gpr uers I D S E A M W gpr

6.6 The S+core Processor and System V ABI

The System V Application Binary Interface (ABI) defines a standard binary system interface for compiled application programs on systems that implement the interfaces defined in the System V Interface Definition, Third Edition.

This document supplements the generic System V ABI and information specific to system V implementation built on the S+core processor.

6.6.1 Machine Interface

6.6.1.1 Data Representation

Within this specification, the term *byte* refers to an 8-bit object; *twobyte*, a 16-bit object; *fourbyte*, a 32-bit object; *eightbyte*, a 64-bit object.

- **Byte Ordering**

The architecture defines data types of 8-bit byte, 16-bit half-word,32-bit word, and 64-bit double-word. Byte ordering defines how the bytes that make up half words, words, double words, and quad words are ordered in memory.

The S+core processor supports either big endian or little endian byte ordering. Big endian, one of the byte ordering way, means that the most significant byte is located in the lowest addressed byte. The little endian is as the opposite of the big endian. The tables below are to illustrate the conventions for byte ordering.

- **Fundamental Types**

The following figure is to display the correspondence between scalar types of ISO C and this processor. A null pointer (for all types) has the value zero.

Table 6-1 Byte Ordering in Half-word

Big Endian		Little Endian	
hword[15:8]	hword[7:0]	hword[15:8]	hword[7:0]
byte 0	byte 1	byte 1	byte 0

Table 6-2 Byte Ordering in Word

Big Endian			
hword[31:24]	hword[23:16]	hword[15:8]	hword[7:0]
byte 0	byte 1	byte 2	byte 3

Little Endian

hword[31:24]	hword[23:16]	hword[15:8]	hword[7:0]
byte 3	byte 2	byte 1	byte 0

Table 6-3 Byte Ordering in Double Word

Big Endian			
hword[63:56]	hword[55:48]	hword[47:40]	hword[39:32]
byte 0	byte 1	byte 2	byte 3
hword[31:24]	hword[23:16]	hword[15:8]	hword[7:0]
byte 4	byte 5	byte 6	byte 7

Little Endian			
hword[63:56]	hword[55:48]	hword[47:40]	hword[39:32]
byte 7	byte 6	byte 5	byte 4
hword[31:24]	hword[23:16]	hword[15:8]	hword[7:0]
byte 3	byte 2	byte 1	byte 0

Table 6-4 Scalar Types

Type	C	Size	Alignment (bytes)	S+core Architecture
Integral	_Bool	1	1	Boolean
	char signed char	1	1	Signed byte
	unsigned char	1	1	Unsigned byte
	short signed short	2	2	Signed twobyte
	unsigned short	2	2	Unsigned twobyte
	int signed int enum†††	4	4	Signed fourbyte Integral
	unsigned int	4	4	Unsigned fourbyte
	long signed long	4	4	Signed fourbyte
	long long signed long long	8	8	Signed eightbyte

	unsigned long unsigned long long	4 8	4 8	Unsigned fourbyte Unsigned eightbyte
Pointer	any-type * any-type (*)()	4	4	Unsigned fourbyte
Floating-point	Float	4	4	Single-precision
	double	8	8	Double-precision
	long double	8	8	Double-precision

Fig 6-1 Structure Smaller than a Word

```
struct S{
char c;
};
```

Byte aligned, sizeof(S) is 1

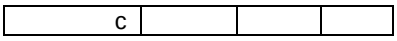
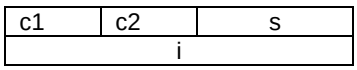


Fig 6-2 No Padding

```
struct S{
char c1;
char c2;
short s;
int i;
};
```

Word aligned, sizeof(S) is 8



■ **Aggregates and Unions**

Aggregates, union aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned components. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object. An array uses the same alignment as its elements. Structure and union objects requires padding, content in it is undefined, to meet the size and alignment constraints.

• An entire structure or union object is aligned on the same boundary as its most strictly aligned member.

• Each of members is assigned to the lowest available offset with the appropriate alignment. This may

require the internal padding, depending on the previous member.

• If necessary, a structure size is increased to make it a multiple of the alignment.

This may require the tail padding, depending on the last member.

Fig 6-3 Internal Padding

```
struct S{
char c;
short s;
};
```

Halfword aligned, sizeof(S) is 4

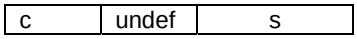
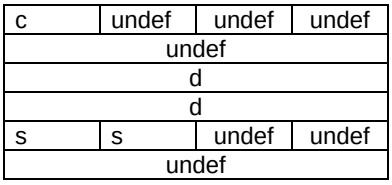


Fig 6-4 Internal and Tail Padding

```
struct S{
char c;
double d;
short s;
};
```

Doubleword align, sizeof(S) is 24



■ Bit Fields

C struct and union definitions may include bit fields that define integral values of specified sizes.

Table 6-5 Bit-Field Ranges

Bit-field Type	Width w	Range
signed char	1 to 8	$-2^{7} \sim +2^{7}-1$
char		$0 \sim +2^{8}-1$
unsigned char		$0 \sim +2^{8}-1$
signed short	1 to 16	$-2^{15} \sim +2^{15}-1$
short		$0 \sim +2^{16}-1$
unsigned short		$0 \sim +2^{16}-1$
signed int	1 to 32	$-2^{31} \sim +2^{31}-1$
int		$0 \sim +2^{32}-1$
unsigned int		
signed long	1 to 32	$-2^{31} \sim +2^{31}-1$

long	0 ~ $+2^{32}$-1
unsigned long	

Bit-fields that are neither signed nor unsigned always have non-negative values. Although featuring different types, char, short, int, or long (which can have negative values), these bit-fields have the same range as a bit-field of the same size with the corresponding unsigned type. Bit-fields obey the same size and alignment rules as other structure and union members.

Also:

• Bit fields are allocated from right to left

• Bit fields must be contained in a storage unit appropriate for its declared type

• Bit fields may share a storage unit with other struct / union members

The types of unnamed bit fields do not affect the alignment of a structure or union.

6.6.2 Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, parameter passing and so on. The standard calling sequence applies only to global functions. Local functions, not reachable from other compilation units, may use other conventions. Nevertheless, it is recommended that all functions use the standard calling sequence if possible.

6.6.2.1 Registers

The S+core architecture provides 32 general-purpose 32-bit registers. All these registers are global to all procedures active for a given thread.

This subsection discusses the usage of each register. Registers r0(stack pointer), r2(frame pointer), r3(link register) and r12 through r15 “belong” to a calling function; and a called function is required to preserve their values for its caller. Remaining registers “belong” to the called function. If a calling function wants to preserve such a register value through a function call, it must save the value in its stack frame.

6.6.2.2 Stack Frame

Similar to registers, each function has a frame on the run-time stack. This stack grows downwards from high addresses. Table 6-7 shows the stack organization.

Each called function in a program allocates its own stack frame on the run-time stack, if necessary. A frame is allocated for each non-leaf function and for each function that requires stack storage.

A stack frame is composed of:

- local variables
- saved general registers. Space is allocated the registers need to be saved.

For non-leaf function, r3 (link register) must be saved. If any of r2, r29, r30, or r12 r19 is changed within the called function, it must be saved in the stack frame before being used and restored before returning from the function. Registers are saved in numerical order, with higher numbered registers saved in

higher memory addresses.

Table 6-6 Register Usage

Register	Usage	Preserved across Register Usage function calls
r0	Stack pointer	Yes
r1	Temporary generally used by assembler.	No
r2	Frame pointer	Yes
r3	Link register	Yes
r4 - r7	Used to pass arguments and return value.	No
r8 - r11	Calling-saved register	No
r12 - r21	Callee-saved register; optionally used as frame pointer	Yes
r22 - r27	Calling-saved register	No
r28	Global pointer	Yes
r29	Compiler reserved	Yes
r30 - r31	Used by the OS only. Compiler must not use them.	Yes
HI	Multiply/divide special register. Holds the most significant 32 bits of multiply or the remainder of a divide	No
LO	Multiply/divide special register. Holds the least significant 32 bits of multiply or the quotient of a divide	No
sr0	Special register for countering	No
sr1	Special register for loading combine instructions	No
sr2	Special register for storing combine instructions	No

Table 6-7 Stack Frame

Position	Contents	Frame
old r0 + 4 * n **old r0 + 0**	Argument spill area arg[n] . . . Argument spill area arg[0]	Previous
new r0 + 0	General registers save area Local variables Allocations Outgoing arguments save area	Current

- function call argument area

In a non-leaf function, the maximum number of bytes of arguments used to call other functions from the non-leaf function must be allocated. However, at least one word must always be reserved.

A function allocates a stack frame by subtracting the frame size from the stack pointer on entry to the

function. The pointer adjustment occurs before the pointer is used within a function. Corresponding restoration of stack pointer at the end of a function must occur after any jump or branch instructions except that prior to the jump instruction that returns from the function.

6.6.2.3 Parameter Passing

Arguments are passed to a function in a combination of integer general registers and the stack. The number of arguments, their types, and their relative positions in the argument list of the calling function determines the mix of registers and memory used to pass arguments. General registers 4..7 pass the first few arguments in registers. Double-word arguments require r4 + r5 or r6 + r7, but not r5 + r6. Thus if a function has a int as its first argument, and a double as its second, the first argument is passed in r4, and second in r6 + r7, leaving a hole in the r5 register. Unnamed arguments are passed on the stack. In determining into which registers, if any, arguments go, take the following considerations into account:

- All integer-valued arguments are passed as 32-bit words, with signed or unsigned bytes and halfwords expanded (promoted) as necessary.
- If the called function returns a structure or union, the caller passes the address of an area that is large enough to hold the structure to the function in r4. The called function copies the returned structure into this area before it returns. This address becomes the first argument to the function for the purposes of argument register allocation and all user arguments are shifted down by one.
- Despite the fact that some or all of the arguments to a function are passed in registers, always allocate space on the stack for all arguments. This stack space should be a structure large enough to contain all the arguments, aligned according to normal structure rules (after promotion and structure return pointer insertion). The locations within the stack frame used for arguments are called the home locations.
- Structures are passed as if they were very wide integers with their size rounded up to an integral number of words. The fill bits necessary for rounding up are undefined.
- A structure can be split so a portion is passed in registers and the remainder passed on the stack. In this case, the first words are passed in r4 r7 as needed, with additional words passed on the stack.
- Unions are considered structures.
- Unnamed arguments corresponding to the variable argument declaration in the function prototyping are all passed on stack.

Example:

int f(int i, ...)

6.6.2.4 Function Return Values

The returning of values is done according to the following algorithm:

- A function can return no value, an integral or pointer value, a double word value (long long or double),

or a structure; unions are treated as structures.

- A function that returns no value puts no particular value in any register.
- A function that returns an integral or pointer value places its result in register r4.
- A function that returns a double word value places its result in r4 + r5.

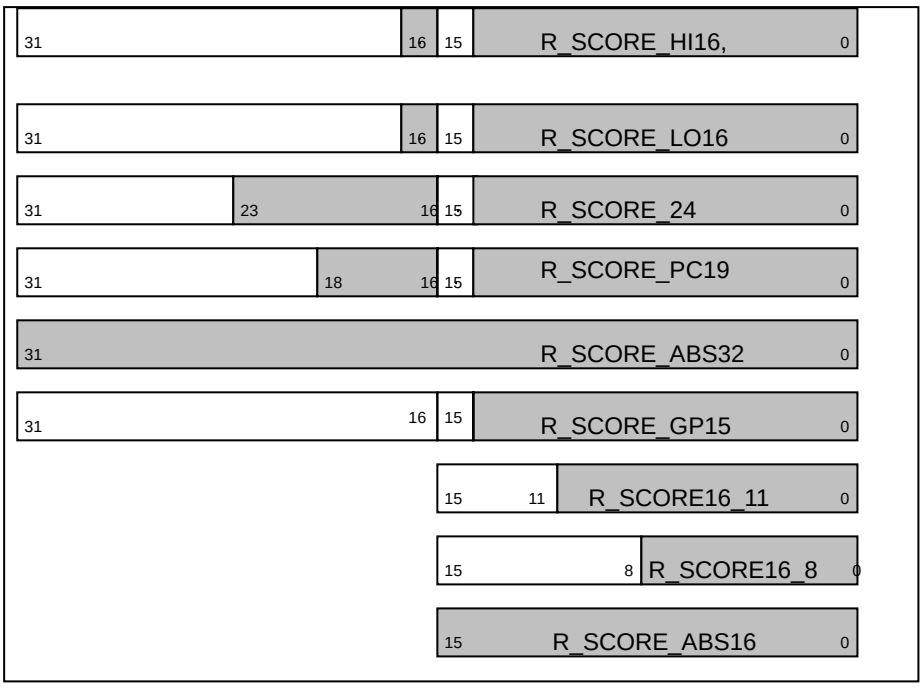
The caller to a function that returns a structure or a union passes the address of an area large enough to hold the structure in register r4. Before returning to its caller, a function will copy the return structure to the area in memory pointed to by r4; the function also returns a pointer to the returned structure in register r4. Having caller supply the return object's space allows re-entrance.

6.7 Relocation

6.7.1 Relocation Fields

Relocation entries describe how to alter the following instructions and data fields shown in Table 6-8; the bit numbers appear in the lower box corners.

Table 6-8 Relocatable Fields



Calculations below assume that a relocatable file is transforming to either an executable or a shared object file. Conceptually, linker merges one or more relocatable files to form the output. It first determines how to combine and locate the input files; then it updates the symbol values and performs the relocation.

Relocations applied to the executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

EA Represents **the effective addend** used to compute the value of the relocatable field.

AHL Identifies another type of addend used to compute the value of the relocatable field.
AHL = ((R_SCORE_HI16)'s EA <<16) | (R_SCORE_LO16)'s EA)

P Represents the place (**the section offset or address**) of the storage unit being relocated (computed using r_offset).

S Represents **the value of the symbol** whose index resides in the relocation entry, unless the symbol is STB_LOCAL and is of type STT_SECTION in which case S represents the original sh_addr minus the final sh_addr.

G Represents the offset into the global offset table where the address of the relocation entry symbol resides during execution.

The value of **`r_offset`**, a relocation entry , designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to be changed and how to calculate their values. The relocated field holds the addend.

The AHL addend is a composite computed from the addends of two consecutive relocation entries. An associated `R_SCORE_LO16` entry is required to immediately follow each relocation type of `R_SCORE_HI16` in the list of relocations.

These relocation entries are always processed as a pair and both addend fields contribute to the AHL addend. If AHI and ALO are the addends from the paired `R_SCORE_HI16` and `R_SCORE_LO16` entries, then AHL should be computed as (AHI << 16) + (short)ALO. `R_SCORE_LO16` entries without an `R_SCORE_HI16` entry immediately preceding are orphaned and the previously defined `R_SCORE_HI16` is used to comput the addend.

If an `R_SCORE_GOT15` refers to a locally defined symbol, it must be followed immediately by an `R_SCORE_GOT_LO16` relocation. The AHL addend is extracted and the section in which the referenced data item resides is determined. From this address, the final address of the data item is calculated. `R_SCORE_CALL15` only can use r29 register, this will be processed as calling function. The relocation will be treated as valid.

6.7.2 Relocation Types

Name	Value	Usage	Calculation
R_SCORE_HI16	1	`ldis rd, HI(symbol)`	(AHL+S)>>16
R_SCORE_LO16	2	`ori rd,LO(symbol)`	(AHL+S)
R_SCORE_24	4	`j label`	$Sign_{24}$(EA)+S
R_SCORE_PC19	5	`b{cond} label`	$Sign_{19}$(EA)+S-P
R_SCORE16_11	6	`j! label`	$Sign_{11}$(EA)+S
R_SCORE16_PC8	7	`b! label`	$Sign_{8}$(EA)+S-P
R_SCORE_GP15	11	`lw r4,label`	$Sign_{16}$(EA)+S-G
R_SCORE_GOT15	14	`lw r4,label`	G
R_SCORE_GOT_LO16	15	`Addi rd,LO(local_label)`	AHL+S
R_SCORE_CALL15	16	`lw r29,extern label`	G

R_SCORE_ABS16	9		S + $Sign_{16}$(EA)
R_SCORE_ABS32	8		S + EA

6.8 Reference

S+core Binutils is ported from gnu Binutils 2.13.2.1 version .

Using compiler: http://gcc.gnu.org, http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc

Using *as*: http://www.gnu.org/software/binutils

Using *ld* The GNU linker : http://www.gnu.org/software/binutils