SUNPLUS

S+core7 Programming Guide

V1.0 – July. 05, 2005

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

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

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 Lin	First Edition

1 Instruction Formats

1.1 Instruction Set Forms

The S+core processor has 32-bit and 16-bit instruction mode. 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.

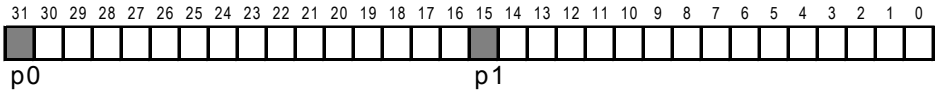


Fig 1-1 Position of P-bit

The instruction fields of S+core processor after truncating the P-bits are shown in Fig 1-2. 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 the different instruction formats.

30 bit

29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7	6 5 4 3 2 1	0
J-form: OP	Disp24(Imm)					LK
BC-form: OP	Disp[18:9] (Imm)		BC	Disp[8:0] (Imm)		LK
Special-form: OP	rD	rA	rB	0 0 0	func6	CU
syscall: OP	SPar(Imm)			0 0 0	func6	CU
MV<cond>: OP	rD (D)	rA (S1)	EC	0 0 0	func6	CU
shift/rotate imm: OP	rD (D)	rA (S1)	SA5 (Imm)	0 0 0	func6	CU
bit operation: OP	rD (D)	rA (S1)	BN5 (Imm)	0 0 0	func6	CU
bittst: OP	0 0 0 0 0	rA (S1)	BN5 (Imm)	0 0 0	func6	CU
neg: OP	rD (D)	0 0 0 0 0	rB (S2)	0 0 0	func6	CU
ext: OP	rD (D)	rA (S1)	0 0 0 0 0	0 0 0	func6	CU
alw/asw, SCx, LCx: OP	rD (D)	rA (addr)	0 0 0 0 0	0 0 0	func6	CU
MFSR: OP	rD (D)	0 0 0 0 0	Srn	0 0 0	func6	CU
MTSR: OP	0 0 0 0 0	rA (S1)	Srn	0 0 0	func6	CU
MFCE/MTCE: OP	rD (D)	rA (S1) (optional)	0 0 0 H L	0 0 0	func6	CU
CMP<cond>: OP	0 0 0 TC	rA (S1)	rB (S2)	0 0 0	func6	CU
CMPZ<cond>: OP	0 0 0 TC	rA (S1)	0 0 0 0 0	0 0 0	func6	CU
mul/div: OP	0 0 0 0 0	rA (S1)	rB (S2)	0 0 0	func6	CU
TRAP<cond>: OP	0 0 0 0 0	SPar (Imm)	EC	0 0 0	func6	CU
SDBBP: OP	0 0 0 0 0	code	0 0 0 0 0	0 0 0	func6	CU
BR<cond>: OP	0 0 0 0 0	rA (S1)	BC	0 0 0	func6	LK
T<cond>: OP	0 0 0 0 0	0 0 0 0 0	EC	0 0 0	func6	CU
nop: OP	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0	func6	CU
I-form: OP	rD (D, S1)	func3	Imm16 (S2)			CU
RI-form: OP	rD (D)	rA (S1)	Imm14 (S2)			CU
RIX-form: OP	rD (D)	rA (S1)	Imm12 (S2)			func3
CEINST: OP	USD1	rA (optional)	rB (optional)	USD2		func5
CEINST: OP	func25					
CR-form: OP	rD	CR	0 0 0 0 0 0 0	CR_OP		

coprocessor 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12 11 10	9 8 7 6 5	4 3	2 1 0
mtc/mfc: OP	rD	CrA	0 0 0 0 0	0 0 0 0 0	CP#	Sub-OP
ldc/stc: OP	rD	CrA	Imm10		CP#	Sub-OP
cop: OP	CrD	CrA	CrB	COP-Code	CP#	Sub-OP
cop: OP	COP-Code20					

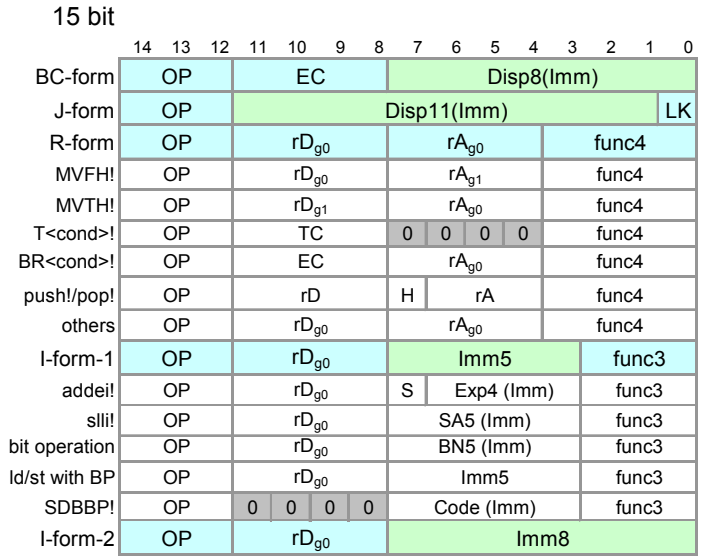


Fig 1-2 S+core Instruction Fields

1.2 Instruction Fields

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

Table 1-1 Instruction Fields

Field	Description
OP	Main opcode.
func	Extended function code.
CU	Condition flag updating bit. 0: Don't update condition flag. 1: Update the condition flag to reflect the result of the operation.
LK	Link bit. 0: Don't update the link register (r3). 1: Update the link register (r3). If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed to link register (r3).
BN	Bit number for bit operation instruction.
EC	Execution condition code for conditional execution instruction.
TC	Test condition code for T flag updating instruction.
SA	Shift amount for shift/rotate instruction.
rD	Target general-purpose register (GPR).
rA, rB	Source general-purpose register (GPR).
g0	Represents GPR 0~15
g1	Represents GPR 16~31.
Disp	Displacements for jump and branch instruction.
Imm	Immediate value.
CR	Control register.
Sub-OP	Extended sub-opcode for coprocessor instruction.

Field	Description
USD	User defined field for extended custom instruction.
CrA	Source coprocessor register.
CrD	Target coprocessor register
CP#	Represents which coprocessor instruction is used.
COP-code	Coprocessor instruction extended opcode.
Exp	Represent the exponent value.
Spar (Imm)	Software parameter used for trap and syscall instructions to pass information.
Srn	Special register number.

1.3 Instruction Naming Convention and Notation

Table 1-2 Convention and Notations

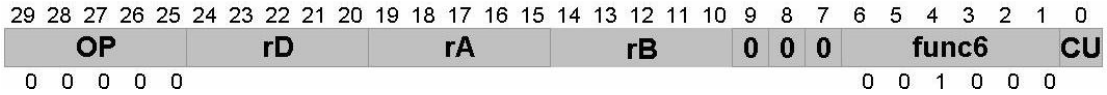
symbol	meaning	example
k'bX	k-bit binary representation of X value	2'b11 = 3
k'hX	k-bit hexadecimal representation of X value	2'h2a = 42
X[n]	selection of bit n of bit string X	X=2'b10, X[1]=1'b1
X[n:m]	selection of bits n through m of bit string X	X = 6'b111001, X[3:0] = 4'b1001
k{A}	replication of 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)	Quatient of X divided by Y	Q(5/2) = 2
R(X/Y)	Remainder of X divided by Y	R(5/2) = 1
&	bitwise logic and	X=2'b11, Y=2'b00, X&Y = 2'b00
|	bitwise logic or	X=2'b11, Y=2'b00, X|Y =2'b11
~	bitwise logic not	X=2'b11, Y = ~X, Y=2'b00
^	bitwise logic xor	X=2'b11, Y=2'b01, X^Y = 2'b10
X^Y	this represents 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 is R3	
GPR[Rn]	this indicates that Rn is a general purpose register	
C	carry flag	

2 S+core Instruction Set

2.1 32-bit Instruction

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

addx **ADD**



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 contents of the general purpose register rA and rB are added to generate a 32-bit result. This result is then placed in the general purpose register rD.

Exceptions

None

addcx **ADD with Carry**

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
```

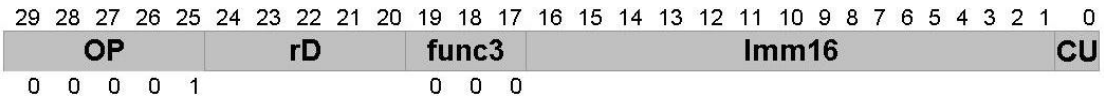
Description

The contents of general-purpose register rA and rB are added with the *Carry flag* to generate a 32-bit result. This result is then placed in general-purpose register rD.

Exceptions

None

addix **ADD with Immediate**



Syntax

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

GPR_{rD} = GPR_{rD} + Sing Extend (Imm16);

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

Usage

```
addi     r4, 0x1234
```

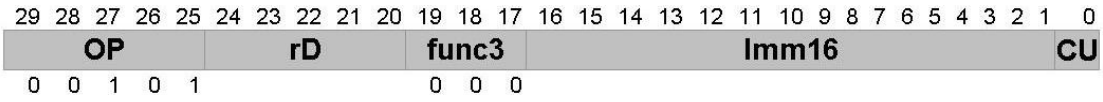
Description

The 16-bit signed immediate is sign extended to 32 bits and then added with the contents of general-purpose register rD to generate a 32-bit result. This result is then placed in general-purpose register rD.

Exceptions

None

addisx **ADD with Immediate Shifted**



Syntax

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

```
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 16-bit immediate is shifted left 16 bits and then is added with the contents of general-purpose register rD to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

addrix **ADD Register with Immediate**

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 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
```

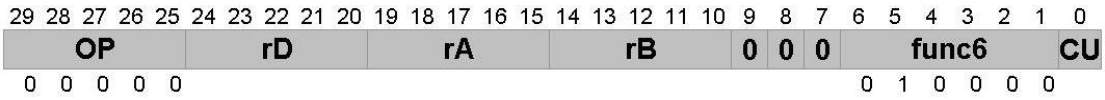
Description

The 14-bit signed immediate is sign extended to 32 bits and then is added with the contents of general-purpose register rA to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

andx **Logical AND**



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 = GPRrA & GPRrB
     Z = (R==0)? 1: 0;
}
```

Usage

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

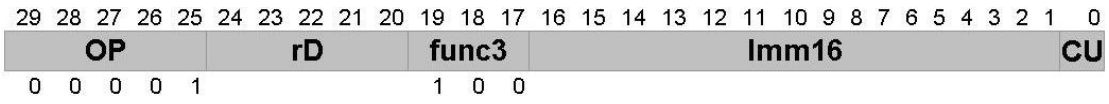
Description

The contents of general purpose register rA and rB are combined in a bit-wise AND operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

andix **Logical AND with Immediate**



Syntax

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

where:

<Rd> Specifies the destination general-purpose register.

<Imm16> Specifies 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
```

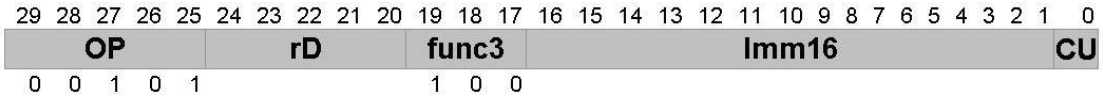
Description

The 16-bit immediate is zero extended to 32 bits and then is combined with the contents of general-purpose register rD in a bit-wise AND operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

andisx **Logical AND with Immediate shifted**



Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<Imm16> Specifies 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
```

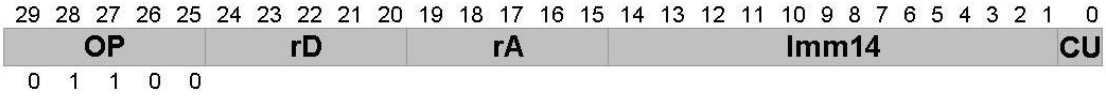
Description

The 16-bit immediate is shifted left 16 bits and then is combined with the contents of general-purpose register rD in a bit-wise AND operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

andrix **Logical AND Register with Immediate**



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 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
```

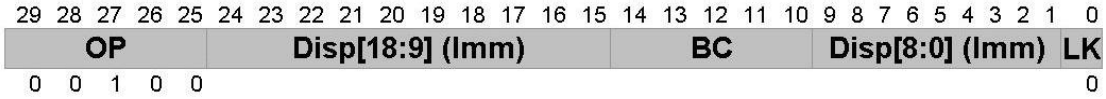
Description

The 14-bit immediate is zero extended to 32 bits and then is combined with the contents of general-purpose register rA in a bit-wise AND operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

b{cond} **Branch conditional**



Syntax

```
b{cond}  Disp19                (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) encoding.

<Disp19> Specifies the branch target address.

<LK> Specifies the link register (r3) updating bit. For b{cond} instruction, the LK bit 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({Disp19, 1'b0});
else
    No operation;
```

Usage

```
beq     Label
```

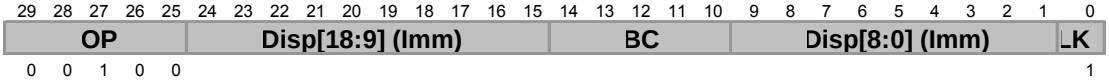
Description

A branch target address is computed from the sum of address of the branch instruction and the 19-bit branch displacement, shifted left one bit and sign-extended to 32 bit. If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address.

Exceptions

None

b{cond}l **Branch conditional and Link**



Syntax

b*{cond}*l Disp19 (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) encoding.

<Disp19> Specifies the branch target address.

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

Operation

$GPR_{r3} = PC_{current} + 4;$

```
if(cond)
  {
       PC = PC_current + SignExtend({Disp19, 1'b0});
       }
  else
       No operation;
```

Usage

```
bcsl          Label
```

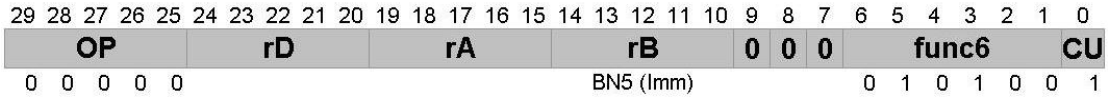
Description

A branch target address is computed from the sum of address of the branch instruction and the 19-bit branch displacement, shifted left one bit and sign-extended to 32 bits. The effective address of the instruction following the branch instruction is place in link register of general-purpose register (r3). If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address and update the link register.

Exceptions

None

bitclr.c **Bit Clear in register**



Syntax

```
bitclr.c rD, rA, BN        (CU = 1)
```

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<BN> Specifies the clear bit location.

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

Operation

GPR_{rD} = {GPR_{rA}[31:BN+1], 1'b0, GPR_{rA}[BN-1:0]};

N = R[31]; // R = {GPR_{rA}[31:BN+1], 1'b0, GPR_{rA}[BN-1:0]}

Z = (GPR_{rA}[BN]==0)? 1 : 0;

Usage

```
bitclr.c r4, r2, 0x0a
```

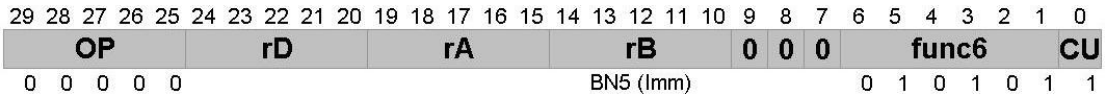
Description

This instruction clears the BN-bit of general-purpose register rA to zero and places the result to general-purpose register rD. The condition flags of N and Z also affected by this instruction.

Exceptions

None

bitset.c **Bit Set in register**



Syntax

```
bitset.c rD, rA, BN        (CU = 1)
```

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<BN> Specifies the set bit location.

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

Operation

GPR_{rD} = {GPR_{rA}[31:BN+1], 1'b1, GPR_{rA}[BN-1:0]};

N = R[31]; // R = {GPR_{rA}[31:BN+1], 1'b1, GPR_{rA}[BN-1:0]}

Z = (GPR_{rA}[BN]==0)? 1 : 0;

Usage

```
bitset.c r4, r2, 0x0a
```

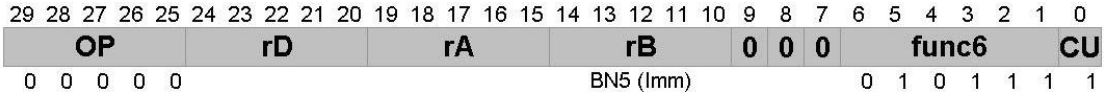
Description

This instruction sets the BN-bit of general-purpose register rA to one and places the result to general-purpose register rD. The condition flags of N and Z also affected by this instruction.

Exceptions

None

bittgl.c **Bit Toggle in register**



Syntax

```
bittgl.c rD, rA, BN        (CU = 1)
```

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

<BN> Specifies the toggle bit location.

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

Operation

GPR_{rD} = {GPR_{rA}[31:BN+1], ~GPR_{rA}[BN], GPR_{rA}[BN-1:0]};

N = R[31]; // R = {GPR_{rA}[31:BN+1], ~ GPR_{rA}[BN], GPR_{rA}[BN-1:0]}

Z = (GPR_{rA}[BN]==0)? 1 : 0;

Usage

```
bittgl.c r4, r2, 0x0a
```

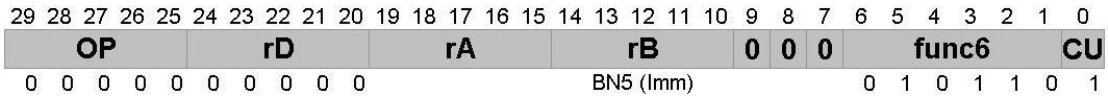
Description

This instruction toggles the BN-bit of general-purpose register rA and places the result to general-purpose register rD. The condition flags of N and Z also affected by this instruction.

Exceptions

None

bittst.c **Bit Test in register**



Syntax

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

where:

<rA> Specifies the source general-purpose register.

<BN> Specifies the toggle bit location.

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

Operation

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

Usage

```
BITTGL.c r2, 0x0a
```

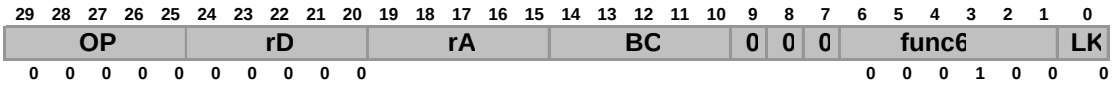
Description

This instruction tests the BN-bit of general-purpose register rA and places the test result to conditional Z flag. The conditional flag of N also affected by this instruction.

Exceptions

None

br{cond} **Branch Register conditional**



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) encoding.
<rA>	Specifies the branch target address register.
<LK>	Specifies the link register (r3) updating bit. For br{cond} instruction, the LK bit always zero.

Operation

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

Usage

```
breq        r3
```

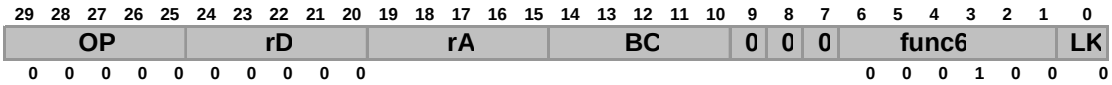
Description

A branch target address is stored in general-purpose register (rA). If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address.

Exceptions

None

br{cond}l **Branch Register conditional and Link**



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) encoding.

<rA> Specifies the branch target address register.

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

Operation

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

Usage

```
breql    r3
```

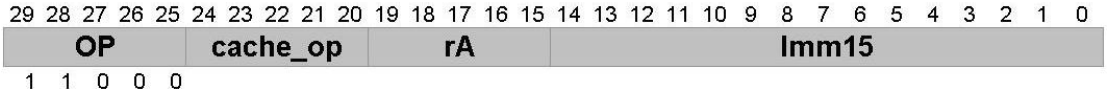
Description

A branch target address is stored in general-purpose register rA. The effective address of the instruction following the branch instruction is place in link register of general-purpose register (r3). If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address and update the link register.

Exceptions

None

cache **Cache operation instruction**



Syntax

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

where:

{cache_op}	Specifies the cache operation. The encoding field of cache operation descript in Table 2-2.
<rA>	Specifies the base address register.
<Simm15>	Specifies the 15-bit signed immediate value.

Table 2-2 Cache operation field encode

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

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

Operation

Usage

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

Description

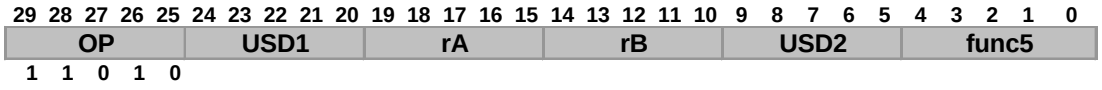
The 15-bit signed immediate value is added to the contents of the base address register rA to form an effective virtual address. The virtual address is translated to a physical address using the fixed-mapping MMU engine. The 5-bit cache_op field specifies a cache operation.

Exceptions

Reserved instruction exception (RI).

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

ceinst **Custom Engine user defined Instruction**



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
```

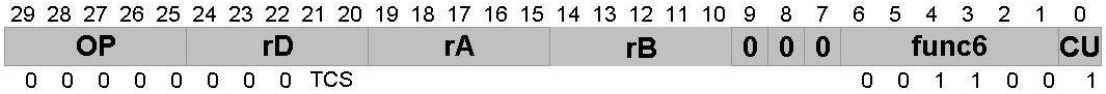
Description

For custom engine instruction extension, CEINST format is defined. In this format, func5 defines the custom engine operation. This format allows maximum two source operands fetched from general-purpose registers. Though rA and rB fields are needed, if one need to specify one or two general purpose source registers, the register indexes should be placed in these two fields. USD1 and USD2 are user-defined fields. These two fields could be either immediate for computation, parameters for operation control or destination general-purpose register index. If custom engine don't implement such extended instruction, reserved instruction exception will occur.

Exceptions

Reserved instruction exception (RI)

cmp{TCS}.c **Compare**



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 conditional flag setting bit and the field encoding is shown in Table 2-3.

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
```

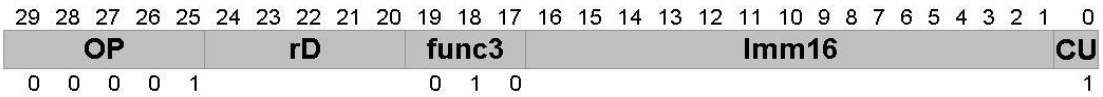
Description

The compare instruction subtracts the general-purpose register rA from rB and updates the conditional flags *N/Z/C/V* according the result and without modify any general-purpose register. By using *TCS* field, user can control the setting and clearing the *T* flag additionally.

Exceptions

None

cmpi.c **Compare with Immediate**



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
```

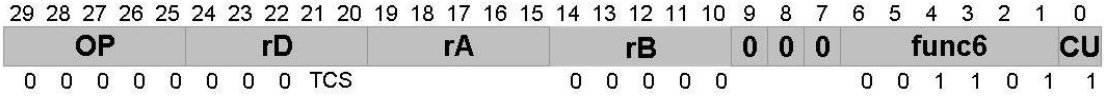
Description

The compare with immediate instruction subtracts the general-purpose register rD from 16-bit signed-extended immediate value and updates the conditional flags *N/Z/C/V* according the result and without modify any general-purpose register.

Exceptions

None

cmpz{TCS}.c **Compare with Zero**



Syntax

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

where:

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

{TCS} Specifies the T conditional flag setting bit and the field encoding is shown in Table 2-4.

<.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
```

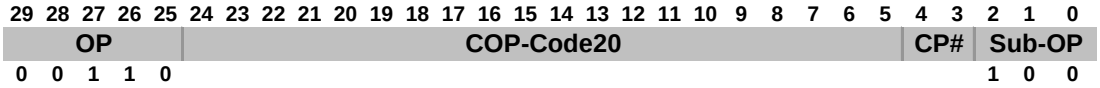
Description

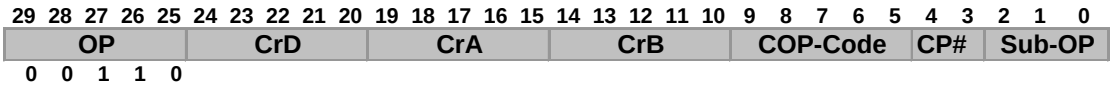
The compare instruction compares the general-purpose register rA and zero, then updates the conditional flags *N/Z/C/V* according the result and without modify any general-purpose register. By using *TCS* field, user can control the setting and clearing the *T* flag additionally.

Exceptions

None

copx **Coprocessor user defined instruction**





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

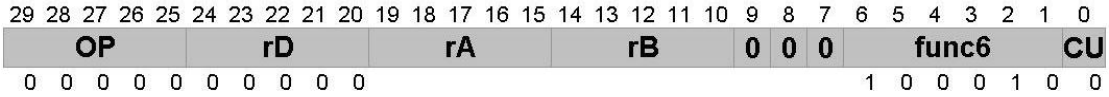
This instruction is a user defined coprocessor extension instruction. There are two types for this instruction, 20-bit parameter or 5-bit parameter form. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception. The coprocessor instruction generates the reserved instruction exception when non-coprocessor responsed the instruction.

Exceptions

Coprocessor Usable exception (CpU)

Reserved Instrcution exception (RI)

div **divide**



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

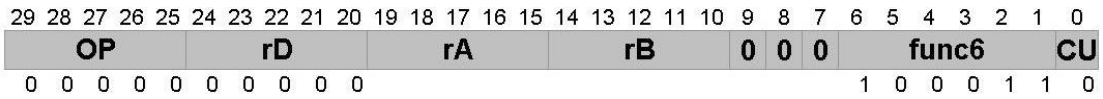
The dividend is the signed value in general-purpose register rA. The divisor is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values, and the custom engine execution exception is induced when the divisor is zero, also result of this operation is undefined.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

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

Exceptions

Custom execution exception: Divided by zero

divu **Divide Unsigned**



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

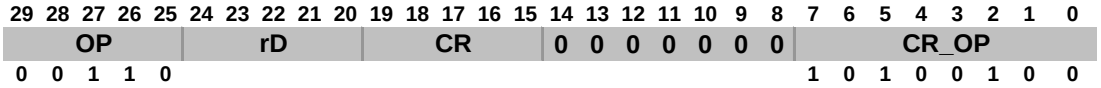
The dividend is the 32-bit unsigned value in general-purpose register rA. The divisor is the 32-bit unsigned value in general-purpose register rB. The custom engine execution exception is induced when the divisor is zero; also result of this operation is undefined. The quotient word of the divided result is placed in custom register CEL, and the remainder word of the divided result is placed in custom register CEH.

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

Exceptions

Custom execution exception: Divided by zero.

drte **Return from Debug Exception**



Syntax

```
drte
```

Operation

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

Usage

```
drte
```

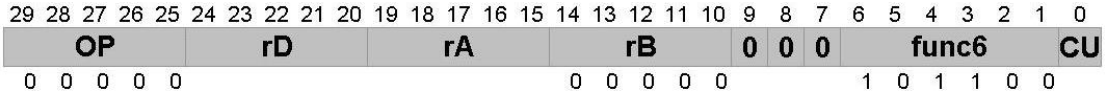
Description

The instruction returns from debug exception by restoring the PC from DEPC. Program continues from the address specified by DEPC. It can only operate in Kernel mode or User mode with *cra*-bit set in PSR register.

Exceptions

None

extsbx **Extend Sign of Byte**



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];

Usage

```
extsb    r4, r2
```

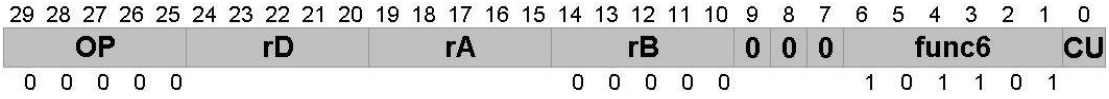
Description

This instruction moves the lower byte (8-bit) of general-purpose register rA into the lower byte (8-bit) of general-purpose register rD. Bit 7 of general-purpose register rA is placed into the remaining of general-purpose register rD.

Exceptions

None

extshx **Extend Sign of Half-word**



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];
```

Usage

```
extsh    r4, r2
```

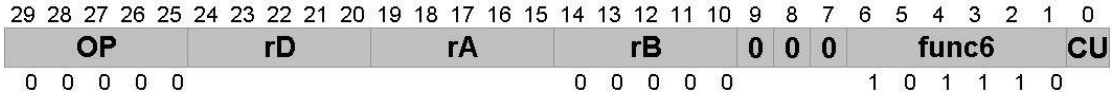
Description

This instruction moves the lower half-word (16-bit) of general-purpose register rA into the lower half-word (16-bit) of general-purpose register rD. Bit 15 of general-purpose register rA is placed into the remaining of general-purpose register rD.

Exceptions

None

extzbx **Extend Zero of Byte**



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];
```

Usage

```
extzb    r4, r2
```

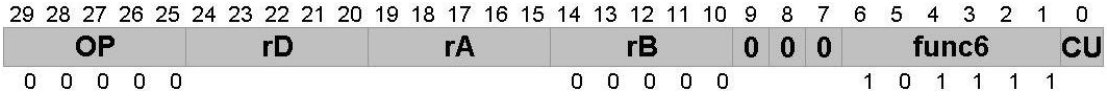
Description

This instruction moves the lower byte (8-bit) of general-purpose register rA into the lower byte (8-bit) of general-purpose register rD. 24-bits of zero is placed into the remaining of general-purpose register rD.

Exceptions

None

extzhx **Extend Zero of Half-word**



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];
```

Usage

```
extzh    r4, r2
```

Description

This instruction moves the lower half-word (16-bit) of general-purpose register rA into the lower half-word (16-bit) of general-purpose register rD. 16-bits of zero is placed into the remaining of general-purpose register rD.

Exceptions

None

jx **Jump (and Link)**

Syntax

```
j       Disp24          (LK = 0)
jl      Disp24          (LK = 1)
```

where:

<Disp24> Specifies the 24-bit jump displacement.

<LK> Specifies the link register (r3) updating bit. For jx instruction, the LK bit always one.

Operation

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

Usage

```
j       Label
```

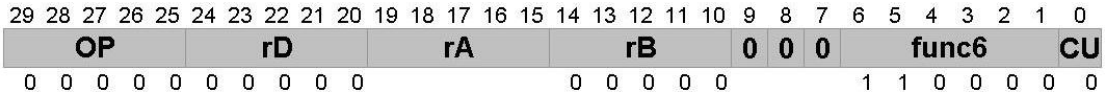
Description

The 24-bit bits displacement address Disp24 is shifted left one bit and combined with the high order seven bits of the address of jump instruction. The program unconditionally execute from the calculated target address. If LK bit sets, the address of the instruction after the jump instruction is placed into the link register GPR_{r3}.

Exceptions

None

lcb **Load Combined word Begin**



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

This instruction loads word data to special register LCR (load combine register) from memory indexed by address rA, after load operation complete the indexed register rA is post-increment by 4. Due to truncate the least significant two bits of address of rA, the address alignment error never occurs in this instruction. This instruction can be used in combined with the lcw and lce instructions to achieve un-alignment memory access.

Given a word in a special register LCR and a word in memory, the operation of load combine instructions is as follows:

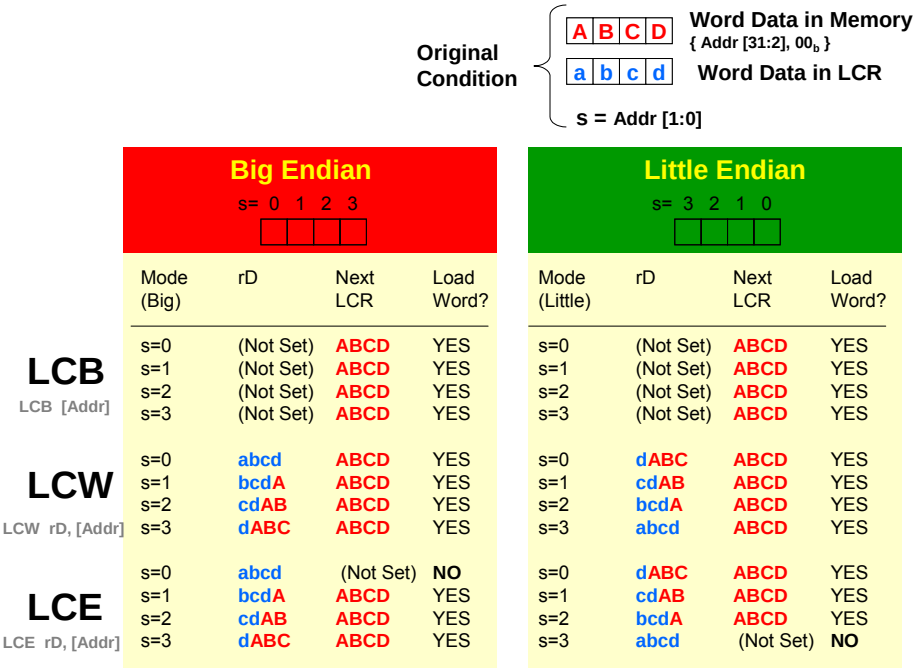


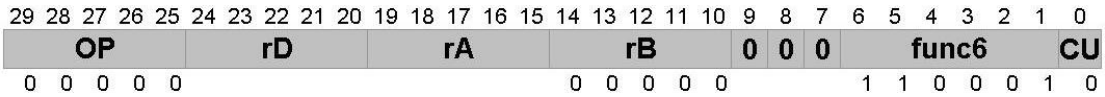
Fig 2-1 The operation table of load combine instruction

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

lcw **Load Combined Word**



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

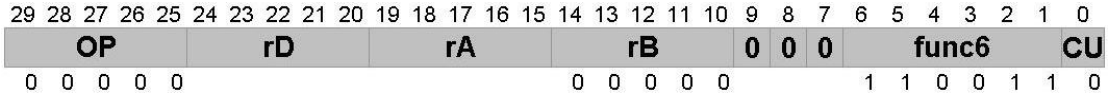
This instruction loads word data to special register LCR (load combine register) from memory indexed by address rA, after load operation complete the indexed register rA is post-increment by 4. Also the load data is shifted and combined with original LCR register according processor endian and the least significant two bits of the indexed address to perform un-alignment access. Due to truncate the least significant two bits of address of rA, the address alignment error never occurs in this instruction. This instruction can be used in combined with the lcb and lce instructions to achieve un-alignment memory access. The detailed operation of lcw is show in Fig 2-1.

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

lce **Load Combined word End**



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

This instruction loads word data to special register LCR (load combine register) from memory indexed by address rA, after load operation complete the indexed register rA is post-increment by 4. Also the load data is shifted and combined with original LCR register according processor endian and the least significant two bits of the indexed address to perform un-alignment access. If the least significant two bits of indexed address rA are zeros, the special register LCR is not changed after executing the lce instruction. Due to truncate the least significant two bits of address of rA, the address alignment error

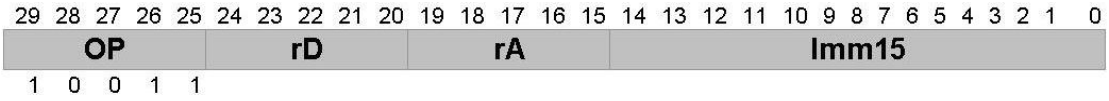
never occurs in this instruction. This instruction can be used in combined with the lcb and lcw instructions to achieve un-alignment memory access. The detailed operation of lce is show in Fig 2-1.

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

lb **Load Byte signed**



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

Usage

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

Description

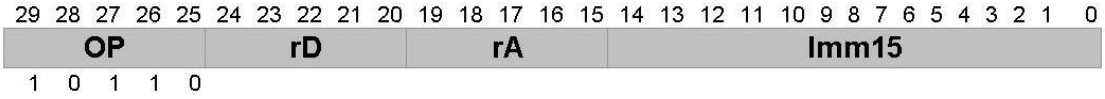
The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The byte in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

lbu **Load Byte Unsigned**



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} = ZeroExtend(Mem8[GPR_{rA}+SignExtend(Imm15)]);

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

lbu **Load Byte Unsigned (post-index)**

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} = ZeroExtend(Mem8[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the general-purpose register rA. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD. After load operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

lbu **Load Byte Unsigned (pre-index)**

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} = ZeroExtend(Mem8[GPR_{rA} + SignExtend(Imm12)]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD. Also the base register rA is updated with the effective address.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

lb **Load Byte signed (post-index)**

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} = SignExtend(Mem8[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the general-purpose register rA. The byte in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD. After load operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

lb **Load Byte signed (pre-index)**

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

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

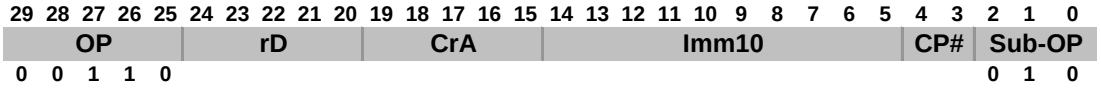
The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The byte in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD. Also the base register rA is updated with the effective address.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

ldcx **Load Coprocessor data register from memory**



Syntax

```
ldc1    CrA, [rD, SImm10]    (CP# = 2'b01)
ldc2    CrA, [rD, SImm10]    (CP# = 2'b10)
ldc3    CrA, [rD, SImm10]    (CP# = 2'b11)
                             (CP# = 2'b00 is reserved)
```

where:

`<rD>` Specifies the base address register.

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

`<Simm10>` Specifies the 10-bit signed immediate value.

Operation

```
CrA = Mem32[GPR_rD +SignExtend({Imm10, 2{0}})];
```

Usage

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

Description

The effective memory address is the sum of general-purpose register rD and 10-bit signed immediate value with shifting left 2 bits. The word in memory addressed by this effective address loads into coprocessor (*CP#*) data register. If the effective address is not word alignment, address alignment error exception will occur. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

Exceptions

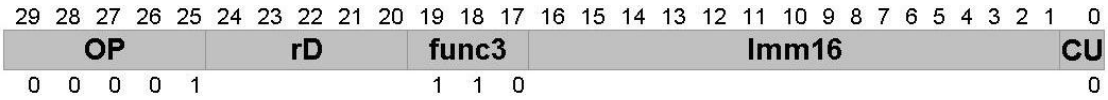
Reserved instruction exception

Coprocessor usable exception

Bus error excetion

Address error exception

ldi **Load Immediate**



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} = SignExtend(SImm16);

Usage

```
ldi       r4, 0xabcd
```

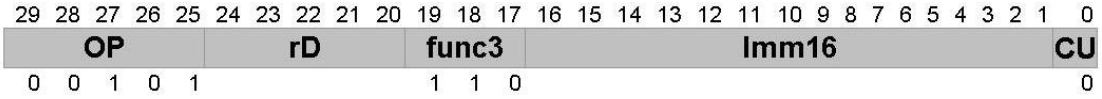
Description

This instruction sign extend the 16-bit immediate value SImm16 and places into general-purpose register rD.

Exceptions

None

ldis **Load Immediate Shifted**



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
```

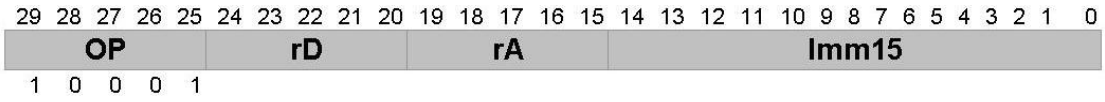
Description

The 16-bit immediate is shifted left 16 bits and inserted zeros into the lower 16-bits to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

lh **Load Half-word signed**



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

Usage

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

Description

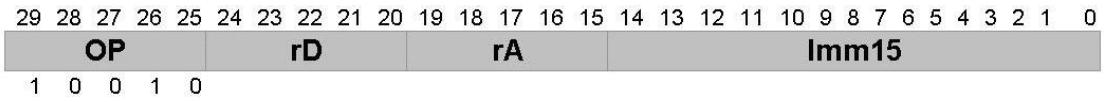
The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The half-word in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lhu **Load Half-word Unsigned**



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} = ZeroExtend(Mem16[GPR_{rA}+SignExtend(Imm15)]);

Usage

```
lhu       r4, [r2, 0x123]
```

Description

The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The half-word in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lhu **Load Half-word Unsigned (post-index)**

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} = ZeroExtend(Mem16[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the general-purpose register rA. The half-word in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD. After load operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lhu **Load Half-word Unsigned (pre-index)**

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} = ZeroExtend(Mem16[GPR_{rA} + SignExtend(Imm12)]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rD. Also the base register rA is updated with the effective address. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lh **Load Half-word signed (post-index)**

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} = SignExtend(Mem16[GPR_{rA}]);

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the general-purpose register rA. The half-word in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD. After load operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lh **Load Half-word signed (pre-index)**

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

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

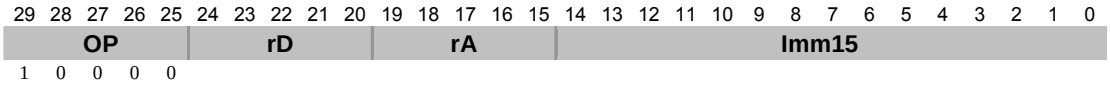
The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The half-word in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rD. Also the base register rA is updated with the effective address. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lw **Load Word**



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} + SignExtend(Imm15)];

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The word in memory addressed by this effective address loads into general-purpose register rD. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lw **Load Word (post-index)**

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} + SignExtend(Imm12);

Usage

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

Description

The effective memory address is the general-purpose register rA. The word in memory addressed by this effective address loads into general-purpose register rD. After load operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

lw **Load Word (pre-index)**

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} + SignExtend(Imm12)];

GPR_{rA} = GPR_{rA} + SignExtend(Imm12);

Usage

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

Description

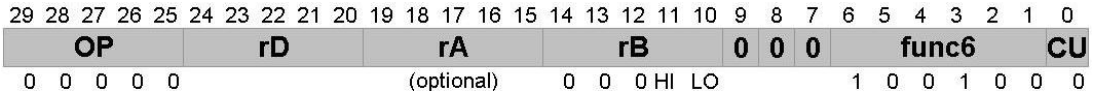
The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The word in memory addressed by this effective address loads into general-purpose register rD. Also the base register rA is updated with the effective address. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

mfcex **Move From Custom Engine**



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

This instruction moves custom engine register CEH/CEL to general-purpose register according to the H and L bit in the instruction form. H bit specifies the transfer of CEH register while L bit specifies the transfer of CEL register. Table 2-5 lists the instruction name, encoding and the corresponding operation of mfc*ex* and mtc*ex* instructions.

Table 2-5 The encoding of custom engine register transfer instruction

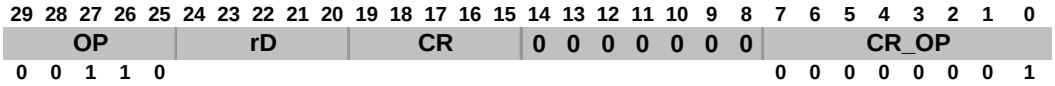
H	L	Instruction	Operations
0	0	Reserved	Reserved
0	1	mfcel	rD = CEL
		mtcel	CEL = rD
1	0	mfceh	rD = CEH
		mtceh	CEH = rD
1	1	mfcehl	rD = CEH, rA = CEL
		mtcehl	CEH = rD, CEL = rA

It may transfer one or two registers at same time; rA is optional for mfce*x* and mtce*x* instructions.

Exceptions

None

mfcr **Move From Control Register**



Syntax

```
mfcr      rD, Cr
```

where:

<Cr> Specifies the source control register.

<rD> Specifies the destination general-purpose register.

Operation

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

Usage

```
mfcr      r4, cr2
```

Description

This instruction moves control register *Crn* into general-purpose register rD. The encoding of field *Crn* is shown in Table 2-6. It can only operate in Kernel mode or 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	*cr1*	Processor Status register (PSR)
5'b00001	*cr2*	Condition register
5'b00010	*cr3*	Exception Cause register (ECR)
5'b00011	*cr4*	Core control register (CCR)
5'b00100	*cr5*	Exception program counter (EPC)
5'b00101	*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)
5'b11111	*cr31*	Debug saved register (DSAVE)

Exceptions

None

mfcx **Move From Coprocessor data register**

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

This instruction moves coprocessor (*CP#*) data register into general-purpose register rD. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

Exceptions

Coprocessor usable exception

mfccx **Move From Coprocessor Control register**

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:

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

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

Operation

GPR_{rD} = CrA

Usage

```
mfcc2    r4, cr8
```

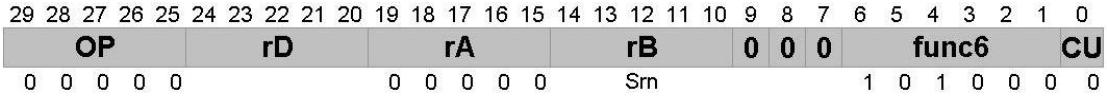
Description

This instruction moves coprocessor (*CP#*) control register into general-purpose register rD. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

Exceptions

Coprocessor usable exception

mfsr **Move From Special Register**



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

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

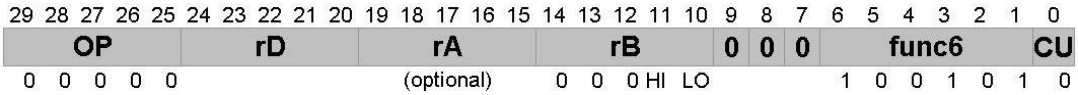
Table 2-7 The encoding of *Srn* field for mfsr instruction

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 **Move To Custom Engine**



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
```

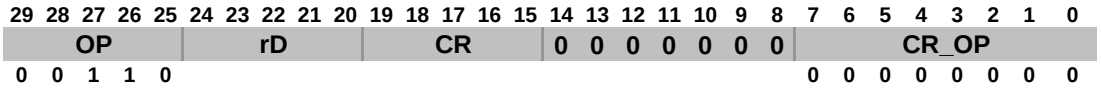
Description

This instruction moves general-purpose register to custom engine register CEH/CEL according to the H and L bit in the instruction form. H bit specifies the transfer of CEH register while L bit specifies the transfer of CEL register. Table 2-5 lists the instruction name, encoding and the corresponding operation of mfce*x* and mtce*x* instructions. It may transfer one or two registers at one time, rA is optional for mfce*x* and mtce*x* instructions.

Exceptions

None

mtcr **Move To Control Register**



Syntax

```
mtcr     rD, Cr
```

where:

<rD> Specifies the source general-purpose register.

<Cr> Specifies the destination control register.

Operation

$CR_n = GPR_{rD}$;

Usage

```
mtcr     r4, cr2
```

Description

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

Exceptions

None

mtcx **Move To Coprocessor data register**

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
```

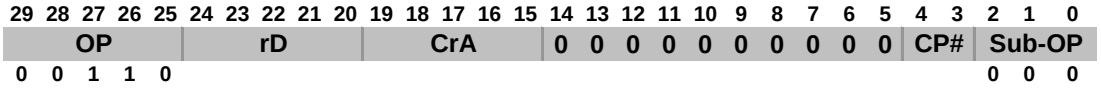
Description

This instruction moves general-purpose register rD into coprocessor (*CP#*) data register. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

Exceptions

Coprocessor usable exception

mtccx **Move To Coprocessor Control register**



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
```

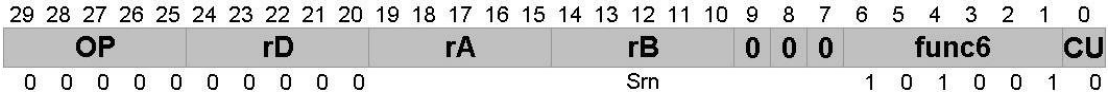
Description

This instruction moves general-purpose register rD into coprocessor (*CP#*) control register. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

Exceptions

Coprocessor usable exception

mtsr **Move To Special Register**



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
```

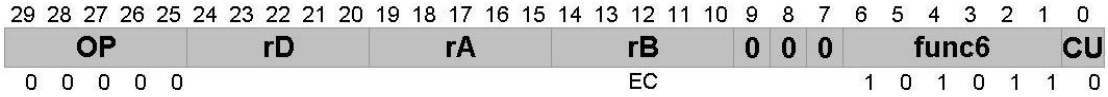
Description

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

Exceptions

None

mv{cond} **Move conditional**



Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the source general-purpose register.

{cond} Specifies the branch condition. Table 2-8 shows the 16 condition options that {*cond*} could be, with their corresponding execution condition (EC) 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	execute on (CNT>0)	CNT>0	CNZ
15	1	1	1	1	execute always	-	AL

Operation

```
if(cond)
    GPR_rD = GPR_rA;
else
    GPR_rD = GPR_rD;
```

Usage

```
mveq    r4, r2
```

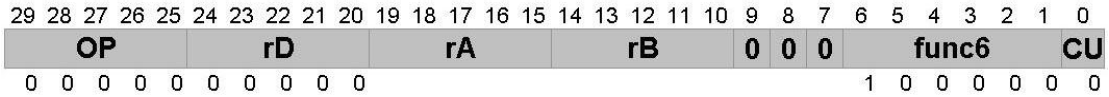
Description

This instruction is a move instruction, which support conditional data transfer between two general-purpose registers. If the condition true as described in Table 2-8, the source register rA moves to destination register rD. Otherwise, the destination register rD doesn't change.

Exceptions

None

mul **Multiply**



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_{rA} are treadted as signed .

Usage

```
mul      r2, r3
```

Description

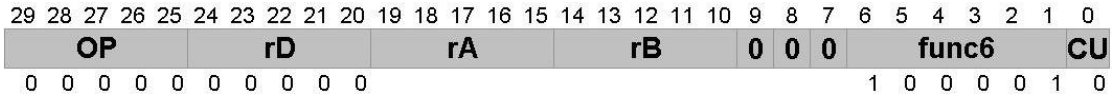
The multiplicand is the signed value in general-purpose register rA. The multiplier is the signed value in general-purpose register rB. Both operands are treated as 32-bit 2's-complement values. The low order word of the multiply result is placed in custom register CEL, and the high order word of the multiply result is placed in custom register CEH.

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

Exceptions

None

mulu **Multiply Unsigned**



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_{rA} are treadted as unsigned .

Usage

```
mulu     r2, r3
```

Description

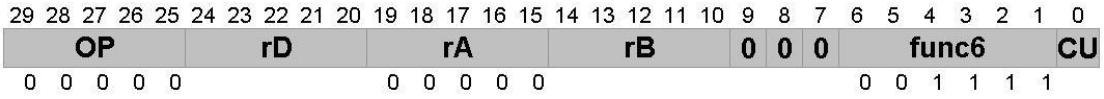
The multiplicand is the unsigned value in general-purpose register rA. The multiplier is the unsigned value in general-purpose register rB.The low order word of the multiply result is placed in custom register CEL, and the high order word of the multiply result is placed in custom register CEH.

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

Exceptions

None

negx **Negative**



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
```

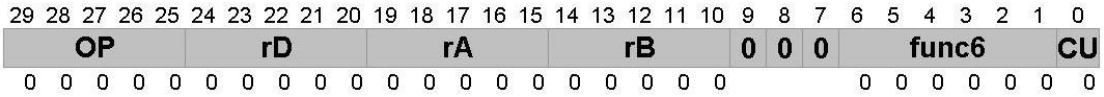
Description

This instruction subtract zero from the content of general-purpose register rB to get the negative value of the content of general-purpose register rB. On the other word, the negative operation performs the add of the one's complement of register rB and the value one and the resulting 2's complement is then placed to general-purpose register rD.

Exceptions

None

nop **No Operation**



Syntax

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

Operation

```
No operation;
```

Usage

```
nop
```

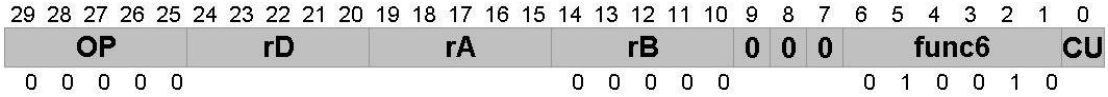
Description

This instruction performs no operation.

Exceptions

None

notx **Logical NOT**



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
```

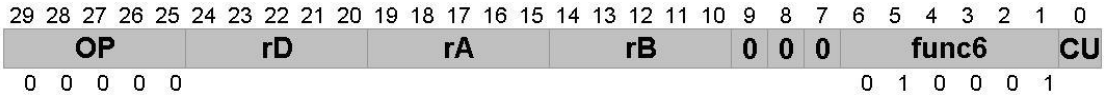
Description

The contents of general-purpose register rA performs a bit-wise NOT operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

orx **Logical OR**



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
```

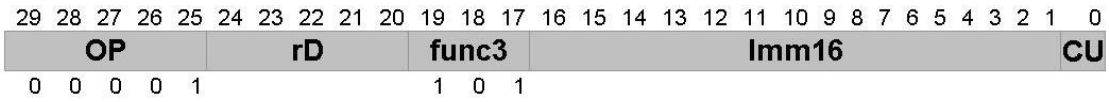
Description

The contents of general-purpose register rA and rB are combined in a bit-wise OR operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

orix **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 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

```
ori      r4, 0x1234
```

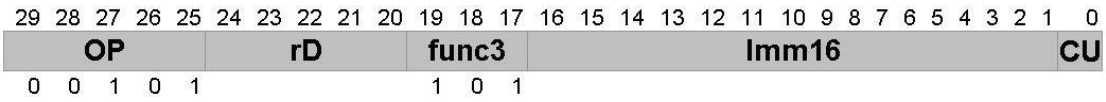
Description

The 16-bit immediate is zero extended to 32 bits and then is combined with the contents of general-purpose register rD in a bit-wise OR operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

orisx **Logical OR with Immediate Shifted**



Syntax

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

where:

<rD>	Specifies the destination general-purpose register.
<Imm16>	Specifies 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 16-bit immediate is shifted left 16 bits and then is combined with the contents of general-purpose register rD in a bit-wise OR operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

orrix **Logical OR Register with Immediate**

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 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

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

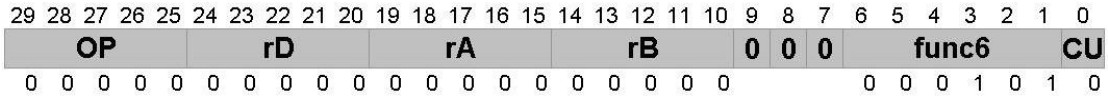
Description

The 14-bit immediate is zero extended to 32 bits and then is combined with the contents of general-purpose register rA in a bit-wise OR operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

pflush **Pipeline Flush**



Syntax

```
pflush
```

Operation

```
Pipeline flush;
Restart program execution at pflush + 4
```

Usage

```
pflush
```

Description

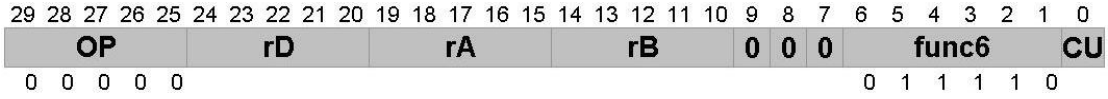
In pipelined processor, there have some instruction sequences need to insert some software bubbles to prevent data hazard. For example, execute load combine instruction to update load combine register (LCR) first and read it immediately (*mfsr*) will need to insert 3 software bubbles. Using one *pflush* instruction to instead inserting several software bubbles, it makes software programming more easily.

The *pflush* instruction behaves like a jump-next instruction, it flush the following pipeline stage and re-execute or re-fetch next instruction (*pflush + 4*) like a non-pipelined processor behavior.

Exceptions

None

rolx **Rotate Left**



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]];
        }
}
```

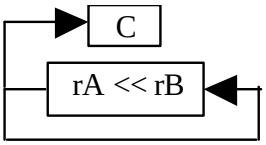
Usage

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

Description

This instruction is a rotate left instruction. The rotate operations only use the lower 5-bits of register rB to specify the rotate amount and then rotate the register rA left by 5-bit amount. This result is then placed to general-purpose register rD. If CU bit set, the rotate instruction also updates the carry flag to the last rotated bit. Fig 2-2 show the operation of rol*x* instructions.

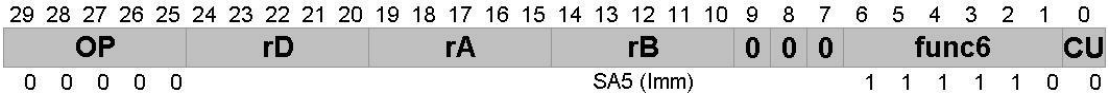
Fig 2-2 The operation of rol*x* instructions



Exceptions

None

rolix **Rotate Left with Immediate**



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
```

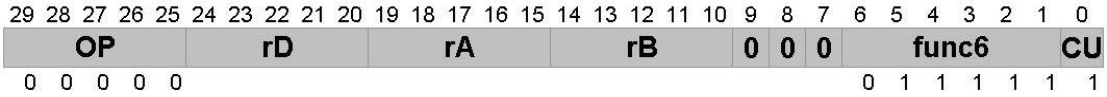
Description

This instruction is a rotate left instruction. The rotate operations use 5-bit immediate value to specify the rotate amount and then rotate the register rA left by 5-bit amount. This result is then placed to general-purpose register rD. If CU bit set, the rotate instruction also updates the carry flag to the last rotated bit. Fig 2-2 also show the operation of rol*x* instructions.

Exceptions

None

rolc.c **Rotate Left with Carry**



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'b1)
{
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];
}
```

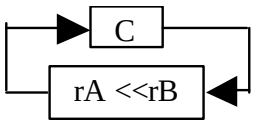
Usage

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

Description

This instruction is a rotate left with carry flag instruction. The rotate operations only use the lower 5-bits of register rB to specify the rotate amount and then rotate the register rA and carry flag left by 5-bit amount. This result is then placed to general-purpose register rD and carry flag. As Fig 2-2 shows, the carry flag and register rA are in rotate chain.

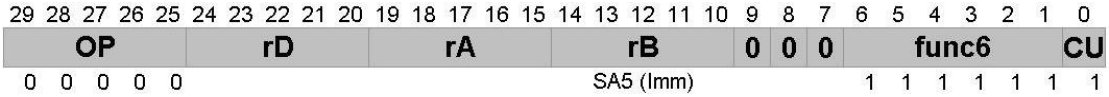
Fig 2-3 The operation of rolc.c instructions



Exceptions

None

rolic.c **Rotate Left Immediate with Carry**



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'b1)
{
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
```

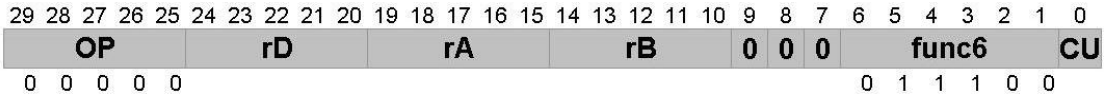
Description

This instruction is a rotate left immediate with carry flag instruction. The rotate operations use 5-bit immediate value to specify the rotate amount and then rotate the register rA and carry flag left by 5-bit amount. This result is then placed to general-purpose register rD and carry flag. As Fig 2-2 shows, the carry flag and register rA are in rotate chain.

Exceptions

None

rorx **Rotate Right**



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)];
        }
}
```

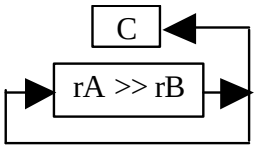
Usage

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

Description

This instruction is a rotate right instruction. The rotate operations only use the lower 5-bits of register rB to specify the rotate amount and then rotate the register rA right by 5-bit amount. This result is then placed to general-purpose register rD. If CU bit set, the rotate instruction also updates the carry flag to the last rotated bit. Fig 2-4 show the operation of ror*x* instructions.

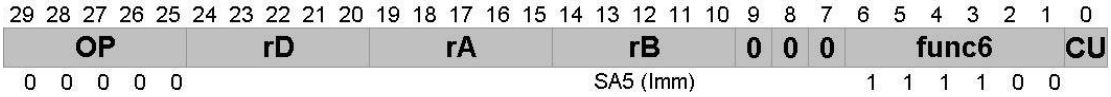
Fig 2-4 The operation of ror*x* instructions



Exceptions

None

rorix **Rotate Right Immediate**



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
```

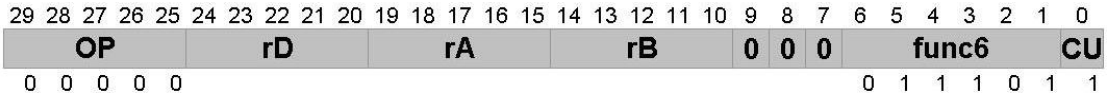
Description

This instruction is a rotate right instruction. The rotate operations 5-bit immediate to specify the rotate amount and then rotate the register rA right by 5-bit amount. This result is then placed to general-purpose register rD. If CU bit set, the rotate instruction also updates the carry flag to the last rotated bit. Fig 2-4 also show the operation of rori*x* instructions.

Exceptions

None

rorc.c **Rotate Right with Carry**



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'b1)
{
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];
}
```

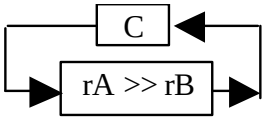
Usage

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

Description

This instruction is a rotate right with carry flag instruction. The rotate operations only use the lower 5-bits of register rB to specify the rotate amount and then rotate the register rA and carry flag right by 5-bit amount. This result is then placed to general-purpose register rD and carry flag. As Fig 2-5 shows, the carry flag and register rA are in rotate chain.

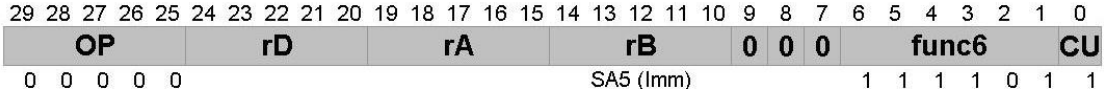
Fig 2-5 The operation of rorc.c instruction



Exceptions

None

roric.c **Rotate Right Immediate with Carry**



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'b1)
{
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

This instruction is a rotate right with carry flag instruction. The rotate operations 5-bit immediate to specify the rotate amount and then rotate the register rA and carry flag right by 5-bit amount. This result is then placed to general-purpose register rD and carry flag. As Fig 2-5 shows, the carry flag and register rA are in rotate chain.

Exceptions

None

rte **Return form Exception**

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;
ATbit = 0;
```

Usage

```
rte
```

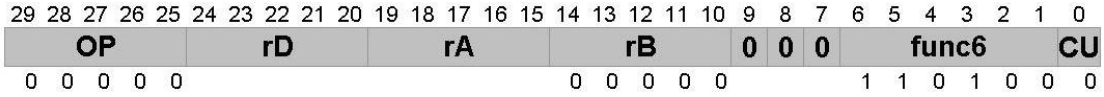
Description

This instruction is used to return from interrupt or exception service routines. The behavior of *rte* acts like jump register instruction, after executing the program counter change to target address pointed by exception program counter (EPC). And some fields of the processor status register (PSR) and condition register restored the previous saved value. It can only operate in Kernel mode or User mode with *cra*-bit set in PSR register.

Exceptions

None

scb **Store Combined word Begin**



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;
```

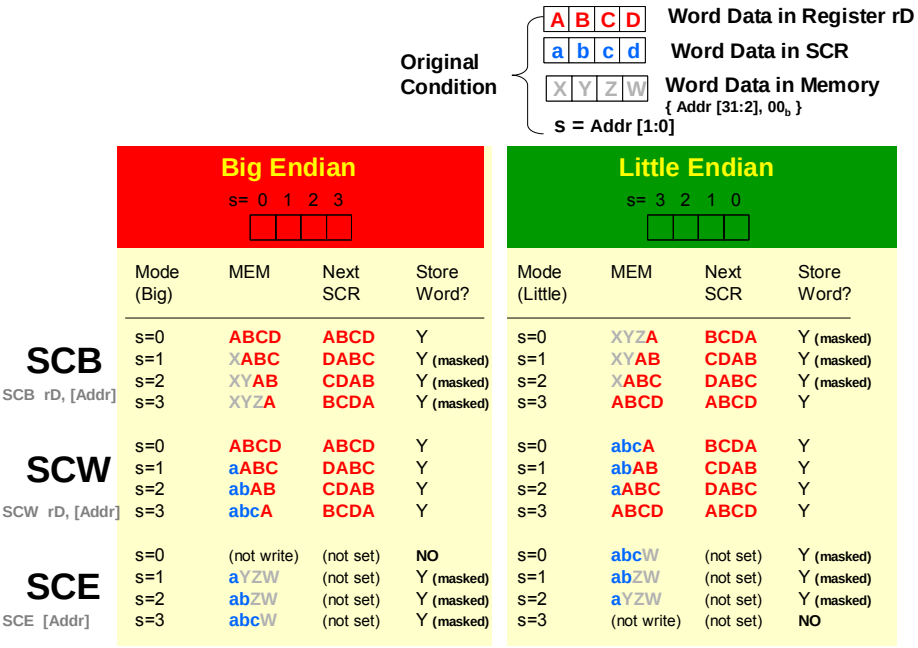
Usage

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

Description

This instruction stores combined word data to special register SCR (store combine register) and memory indexed by address rA, after store operation complete the indexed register rA is post-increment by 4. The store size and address is depended on processor endine and the least significant two bits of index address register rA, the address alignment error never occurs in this instruction. This instruction can be used in combined with the scw and sce instructions to achieve un-alignment memory access.

Given a word in a special register SCR and a word in memory, the operation of store combine instructions is as follows:



	Mode (Big)	MEM	Next SCR	Store Word?	Mode (Little)	MEM	Next SCR	Store Word?
SCB SCB rD, [Addr]	s=0	ABCD	ABCD	Y	s=0	XYZA	BCDA	Y (masked)
	s=1	XABC	DABC	Y (masked)	s=1	XYAB	CDAB	Y (masked)
	s=2	XYAB	CDAB	Y (masked)	s=2	XABC	DABC	Y (masked)
	s=3	XYZA	BCDA	Y (masked)	s=3	ABCD	ABCD	Y
SCW SCW rD, [Addr]	s=0	ABCD	ABCD	Y	s=0	abcA	BCDA	Y
	s=1	aABC	DABC	Y	s=1	abAB	CDAB	Y
	s=2	abAB	CDAB	Y	s=2	aABC	DABC	Y
	s=3	abcA	BCDA	Y	s=3	ABCD	ABCD	Y
SCE SCE [Addr]	s=0	(not write)	(not set)	NO	s=0	abcW	(not set)	Y (masked)
	s=1	aYZW	(not set)	Y (masked)	s=1	abZW	(not set)	Y (masked)
	s=2	abZW	(not set)	Y (masked)	s=2	aYZW	(not set)	Y (masked)
	s=3	abcW	(not set)	Y (masked)	s=3	(not write)	(not set)	NO

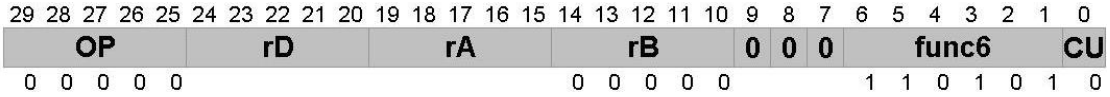
Fig 2-6 The operation table of store combine instruction

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

scw **Store Combined Word**



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

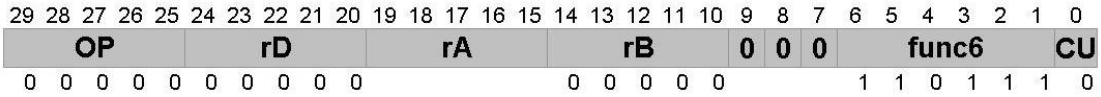
This instruction stores combined word data to special register SCR (store combine register) and memory indexed by address rA, after store operation complete the indexed register rA is post-increment by 4. After combining source register rD and SCR register according processor endian and least significant two bits of indexed register rA, the store operation is always a word access. So, the address alignment error never occurs in this instruction. This instruction can be used in combined with the scb and sce instructions to achieve un-alignment memory access.

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

sce **Store Combined word End**



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

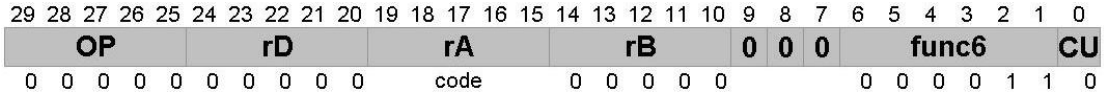
This instruction stores special register SCR (store combine register) to memory indexed by address rA, after store operation complete the indexed register rA is post-increment by 4. The store size and address is depended on processor endine and the least significant two bits of index address register rA, the address alignment error never occurs in this instruction. This instruction can be used in combined with the scb and scw instructions to achieve un-alignment memory access.

Exceptions

Bus error excetion

Address error exception: User mode accesses Kernel/Debug mode address range

sdbbp **Software Debug Breakpoint**



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; BrkSt = 1'b1; DBp = 1'b1;
}
else
NOP;
```

Usage

```
sdbbp    0x0E
```

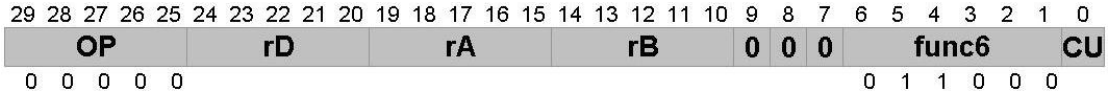
Description

This instruction will induce software debug breakpoint exception, passing control to an exception handler. The exception vector location depends the ice and probe circuit situation, also the corresponding status bits in debug register (DREG) will set and the debug exception program counter (DEPC) will point to the address of *sdbbp* instruction.

Exceptions

None

sllx **Shift Left Logical**



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
```

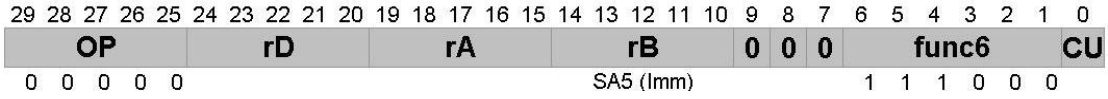
Description

This instruction is a shift left logical instruction. The shift operation only uses the low order 5 bits of register rB to specify the shift amount (*SA*), then shift left by *SA* bit and insert zeros into the low order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

sllix **Shift Left Logical with Immediate**



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
```

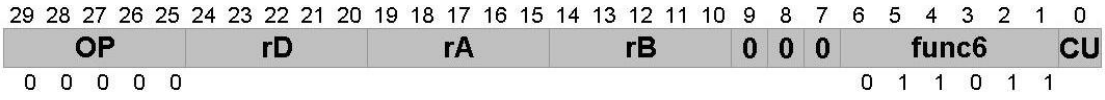
Description

This instruction is a shift left logical with immediate instruction. The shift operation uses 5-bit shift amount (*SA*) to shift left by *SA* bit and insert zeros into the low order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

srax **Shift Right Arithmetic**



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
```

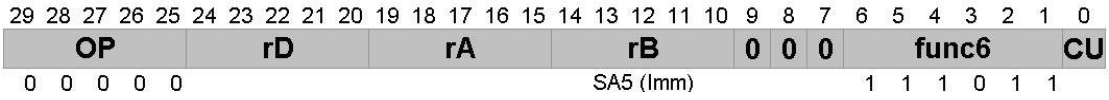
Description

This instruction is a shift right arithmetic instruction. The shift operation only uses the low order 5 bits of register rB to specify the shift amount (*SA*), then shift right by *SA* bit and sign-extend the high order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

sraix **Shift Right Arithmetic with Immediate**



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
```

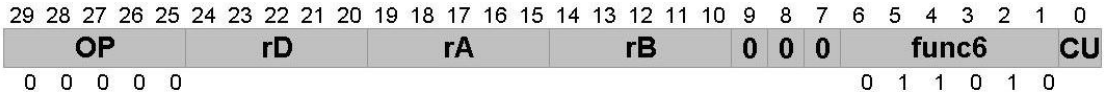
Description

This instruction is a shift right arithmetic instruction. The shift operation uses 5-bit immediate value to specify the shift amount (*SA*), then shift right by *SA* bit and sign-extend the high order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

srlx **Shift Right Logical**



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
```

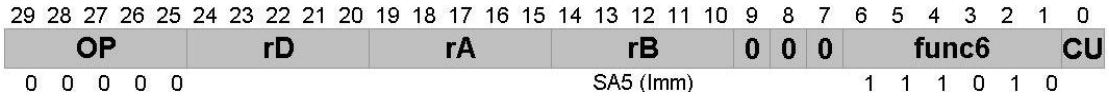
Description

This instruction is a shift right logical instruction. The shift operation only uses the low order 5 bits of register rB to specify the shift amount (*SA*), then shift right by *SA* bit and insert zeros into high order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

srlix **Shift Right Logical with Immediate**



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

This instruction is a shift right logic instruction. The shift operation uses 5-bit immediate value to specify the shift amount (*SA*), then shift right by *SA* bit and insert zeros into high order bits of general-purpose register rA. This result is placed to general-purpose register rD.

Exceptions

None

sb **Store Byte**

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

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

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The byte in memory addressed by this effective address is stored by the lowest 8-bits of general-purpose register rD.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

sb **Store Byte (post-index)**

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

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

The effective memory address is the general-purpose register rA. The byte in memory addressed by this effective address is stored by the lowest 8-bits of general-purpose register rD. After store operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

sb **Store Byte (pre-index)**

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

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The byte in memory addressed by this effective address is stored by the lowest 8-bits of general-purpose register rD. Also the base register rA is updated with the effective address.

Exceptions

Bus error excetion

Address error exception: User mode accesses kernel/debug mode address range

sh **Store Half-word**

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

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

Usage

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

Description

The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The half-word in memory addressed by this effective address is stored by the lowest 15-bits of general-purpose register rD. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

sh **Store Half-word (post-index)**

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

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

The effective memory address is the general-purpose register rA. The half-word in memory addressed by this effective address is stored by the lowest 16-bit of general-purpose register rD. After store operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

sh **Store Half-word (pre-index)**

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

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

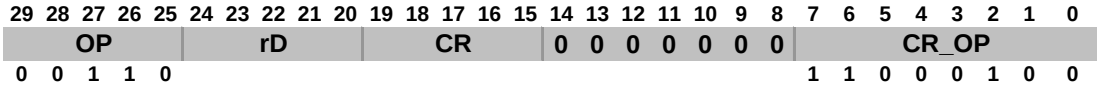
The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The half-word in memory addressed by this effective address is stored by the lowest 16-bit of general-purpose register rD. Also the base register rA is updated with the effective address. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

sleep **Sleep instruction**



Syntax

```
sleep
```

Operation

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

Usage

```
sleep
```

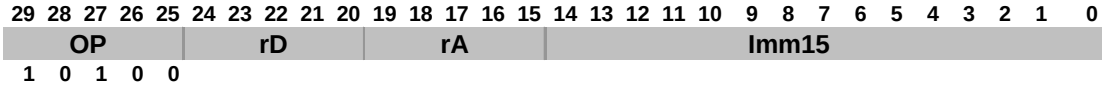
Description

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

Exceptions

Coprocessor usable exception

sw **Store Word**



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

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

Usage

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

Description

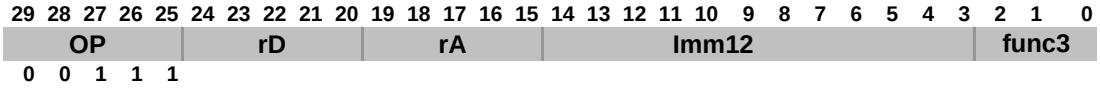
The effective memory address is the sum of general-purpose register rA and 15-bit signed immediate value. The word in memory addressed by this effective address is stored by 32-bit general-purpose register rD. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

sw **Store Word (post-index)**



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

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

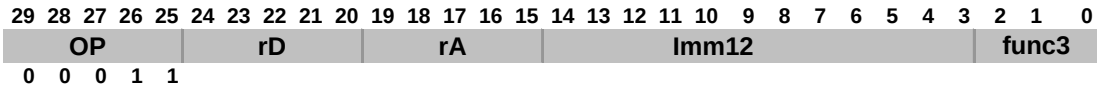
The effective memory address is the general-purpose register rA. The word in memory addressed by this effective address is stored by 32-bit general-purpose register rD. After store operation complete, the base register rA is updated with the address summed by rA and 12-bit signed immediate value. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

sw **Store Word (pre-index)**



Syntax

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

Operation

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

$GPR_{rA} = GPR_{rA} + SignExtend(Imm12);$

Usage

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

Description

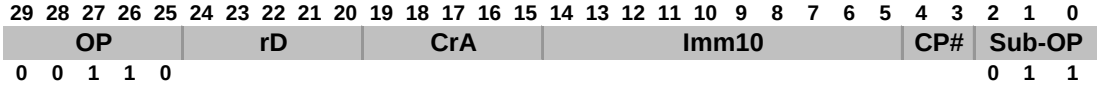
The effective memory address is the sum of general-purpose register rA and 12-bit signed immediate value. The word in memory addressed by this effective address is stored by 32-bit general-purpose register rD. Also the base register rA is updated with the effective address. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error excetion

Address error exception

stcx **Store Coprocessor data register to memory**



Syntax

```
stc1    CrA, [rD, SImm10]     (CP# = 2'b01)
stc2    CrA, [rD, SImm10]     (CP# = 2'b10)
stc3    CrA, [rD, SImm10]     (CP# = 2'b11)
                              (CP# = 2'b00 is reserved)
```

where:

`<rD>` Specifies the base address register.

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

`<Simm10>` Specifies the 10-bit signed immediate value.

Operation

```
Mem32[GPR_rD+SignExtend({Imm10, 2{0}})] = CrA;
```

Usage

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

Description

The effective memory address is the sum of general-purpose register rD and 10-bit signed immediate value with shifting left 2 bits. The word in memory addressed by this effective address stores by coprocessor (*CP#*) data register. If the effective address is not word alignment, address alignment error exception will occur. If the corresponding bit of *cu* field in processor status register (PSR) don't enable, execute coprocessor transfer instruction will occur coprocessor usable (*CpU*) exception.

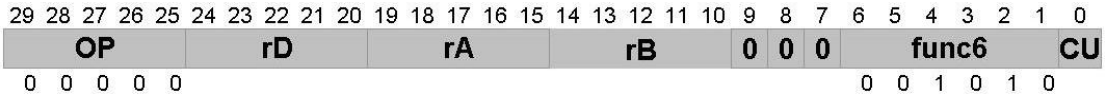
Exceptions

Coprocessor usable exception

Bus error excetion

Address error exception

subx **Subtract**



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
```

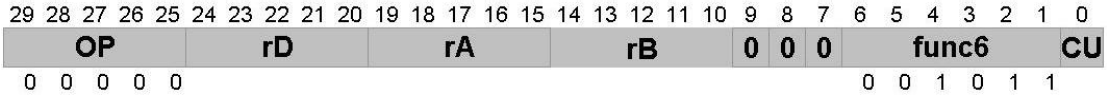
Description

The contents of general-purpose register rA are subtracted from register rB to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

subcx **Subtract with Carry**



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
```

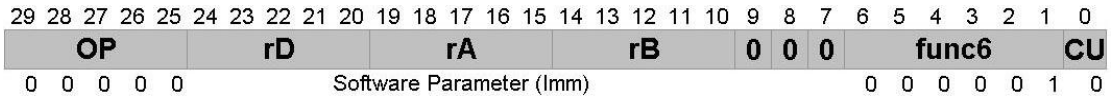
Description

The contents of general-purpose register rA and rB are subtracted with the *Carry flag* to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

syscall **System Call trap**



Syntax

```
syscall  Software_Parameter(Imm)        (CU = 0)
```

where:

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

Operation

```
System call trap
```

Usage

```
syscall  0x008E
```

Description

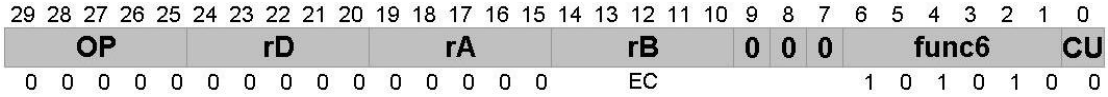
The syscall exception occurs immediately and unconditionally transferring control to the exception handler. The program registers in the processor core are unchanged when this exception occurs except the following:

- The bit field IEc, KUc in processor status register (PSR) is saved
- The bit field Tc, Nc, Zc, Cc, Vc in Condition register is saved
- The Exc_code in Cause register is 7
- The EPC register points at the syscall instruction

Exceptions

None

t{cond} **test and set T flag condition**



Syntax

t*{cond}* (CU = 0)

where:

t Specifies the condition flag T. T flag is used in parallel condition execution, it determines 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 branch condition (EC) 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
```

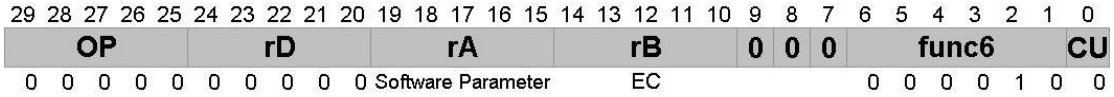
Description

The t<*cond*> instructions are used to set/clear T flag according to the result of condition flag test. The EC field specifies the corresponding condition flag test.

Exceptions

None

trap{cond} **Trap conditional**



Syntax

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

where:

<software_parameter>	Specifies the 5-bits software parameter.
{cond}	Specifies the trap exception condition. Table 2-10 shows the 15 condition options that {*cond*} could be, with their corresponding trap condition (EC) 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 User mode with cra-bit set in PSR register.

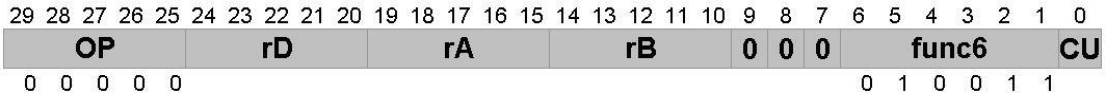
The trap exception occurs conditionally transferring control to the exception handler. The program registers in the processor core are unchanged when this exception occurs except the following:

- The bit field IEc, KUc in processor status register (PSR) is saved
- The bit field Tc, Nc, Zc, Cc, Vc in Condition register is saved
- The Exc_code in Cause register is 10
- The EPC register points at the trap<*cond*> instruction

Exceptions

None

xorx **Logical XOR**



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 = GPRrA ^ GPRrB
    Z = (R==0)? 1 : 0;
}
```

Usage

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

Description

The contents of general-purpose register rA and rB are combined in a bit-wise XOR operation to generate a 32 bit result. This result is then placed to general-purpose register rD.

Exceptions

None

2.2 16-bit Instruction Sets

The 16-bit instruction set is used in hybrid instruction mode and parallel condition execution to reduce the code size. Due to the restriction of instruction width in 16-bit instruction set, 2-operand operation and small immediate value is allowed. If the last 16-bit instruction is not word alignment, the best way is to change this 16-bit instruction into the corresponding 32-bit instruction. The bad way is to insert one 16-bit "*nop!*" instruction to align word boundary, it increases the execution instruction count.

add! **ADD**

Syntax

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

where:

<rDg0> Specifies the destination and first 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 = carry (GPR_rDg0 + GPR_rAg0);
V = overflow (GPR_rDg0 + GPR_rAg0);
GPR_rDg0 = GPR_rDg0 + GPR_rAg0;
```

Usage

```
add!     r4, r2
```

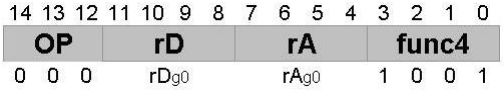
Description

The contents of 32-bit general-purpose register rDg0 and rAg0 are added to generate a 32-bit result. This result is then placed to general-purpose register rDg0. Also the condition flags (N, Z, C, V) always update by this instruction.

Exceptions

None

addc! **ADD with Carry**



Syntax

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

where:

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

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

Operation

```
N = R[31];   // R = GPRrDg0 + GPRrAg0 + C
Z = (R==0)? 1 : 0;
C = carry (GPRrDg0 + GPRrAg0 + C);
V = overflow (GPRrDg0 + GPRrAg0 + C);
GPRrDg0 = GPRrDg0 + GPRrAg0 + C;
```

Usage

```
addc!    r15, r7
```

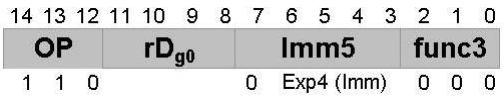
Description

The contents of 32-bit general-purpose register rDg0 and rAg0 are added with the *Carry flag* to generate a 32-bit result. This result is then placed to general-purpose register rDg0. Also the condition flags (N, Z, C, V) always update by this instruction.

Exceptions

None

addei! **ADD with Exponent Immediate**



Syntax

```
addei!   rDg0, Exp4        (CU = 1)
```

where:

<Dg0> Specifies the destination and first source general-purpose register (r0~r15).

<Exp4> Specifies the 4-bit exponent immediate value.

Operation

```
N = R[31];   // R = GPRrDg0 + 2Exp4
Z = (R==0)? 1 : 0;
C = carry (GPRrDg0 + 2Exp4);
V = overflow (GPRrDg0 + 2Exp4);
GPRrDg0 = GPRrDg0 + 2Exp4;
```

Usage

```
addei!   r9, r11
```

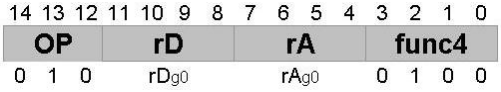
Description

The contents of 32-bit general-purpose register rDg0 and immediate, 2^{Exp4} are added to generate a 32-bit result. This result is then placed to general-purpose register rDg0. Also the condition flags (N, Z, C, V) always update by this instruction.

Exceptions

None

and! **Logical AND**



Syntax

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

where:

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

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

Operation

```
N = R[31];   // R = GPRrDg0 & GPRrAg0
Z = (R==0)? 1 : 0;
GPRrDg0 = GPRrDg0 & GPRrAg0;
```

Usage

```
and!     r14, r2
```

Description

The contents of 32-bit general-purpose register rDg0 and rAg0 are combined in a bit-wise AND operation to generate a 32-bit result. This result is then placed to general-purpose register rDg0. Also the condition flags (N, Z) always update by this instruction.

Exceptions

None

b{cond}! **Branch condition**

Syntax

```
b{cond}! Disp8
```

where:

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

<Disp8> Specifies the branch target address.

Operation

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

Usage

```
beq!    Label
```

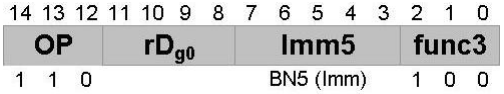
Description

A branch target address is computed from the sum of address of the branch instruction and the 8-bit branch displacement, shifted left one bit and sign-extended to 32 bit. If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address.

Exceptions

None

bitclr! **Bit Clear in register**



Syntax

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

where:

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

<BN> Specifies the clear bit location.

Operation

```
N = R[31];   // R = { GPRrDg0 [31:BN+1], 1'b0, GPRrDg0 [BN-1:0]}
Z = (GPRrDg0 [BN]==0)? 1 : 0;
GPRrDg0 = {GPRrDg0 [31:BN+1], 1'b0, GPRrDg0 [BN-1:0]};
```

Usage

```
bltclr!  r5, 0x0E
```

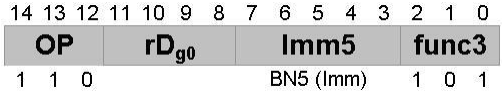
Description

This instruction clears the BN-bit of general-purpose register rDg0 to 0 and updates the value of N and Z flags.

Exceptions

None

bitset! **Bit Set in register**



Syntax

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

where:

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

<BN> Specifies the set bit location.

Operation

```
N = R[31];   // R = { GPRrDg0 [31:BN+1], 1'b1, GPRrDg0 [BN-1:0]}
Z = (GPRrDg0 [BN]==0)? 1 : 0;
GPRrDg0 = {GPRrDg0 [31:BN+1], 1'b1, GPRrDg0 [BN-1:0]};
```

Usage

```
bitset!  R8, 0x09
```

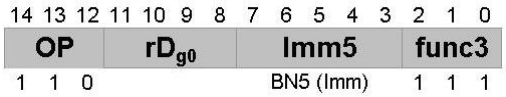
Description

This instruction sets the BN-bit of general-purpose register rDg0 to 1 and updates the value of N and Z flags.

Exceptions

None

bittgl! **Bit Toggle in register**



Syntax

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

where:

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

<BN> Specifies the toggle bit location.

Operation

```
N = R[31];   // R = {GPRrDg0 [31:BN+1], ~ GPRrDg0 [BN], GPRrDg0 [BN-1:0]}
Z = (GPRrDg0[BN]==0)? 1 : 0;
GPRrDg0 = {GPRrDg0 [31:BN+1], ~ GPRrDg0 [BN], GPRrDg0 [BN-1:0]};
```

Usage

```
bittgl!  r0, 0x0E
```

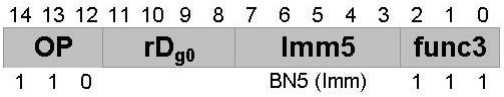
Description

This instruction toggles the BN-bit of general-purpose register rDg0 and updates the value of N and Z flags.

Exceptions

None

bittst! **Bit Test in register**



Syntax

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

where:

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

<BN> Specifies the test bit location.

Operation

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

Usage

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

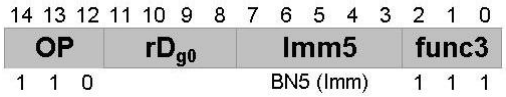
Description

This instruction tests the BN-bit of general-purpose register rDg0 and updates the value of N and Z flags.

Exceptions

None

br{cond}! **Branch Register Conditional**



Syntax

`br{cond}!  rDg0`

where:

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

`<rDg0>` Specifies the branch target address register.

Operation

```
if (cond)
    PC = GPRrDg0;
else
    NOP;
```

Usage

```
brcc!   r12
```

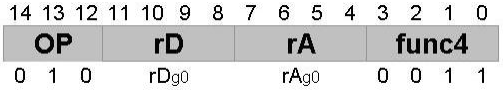
Description

A branch target address is stored in general-purpose register (rDg0). If the condition code (cc) test returns a true result according to the BC field, then the program counter branch to the target address.

Exceptions

None

cmp! **Compare**



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 = GPRrDg0 - GPRrAg0
Z = (R==0)? 1 : 0;
C = ~borrow (GPRrDg0 - GPRrAg0);
V = overflow (GPRrDg0 - GPRrAg0);
```

Usage

```
cmp!    r6, r2
```

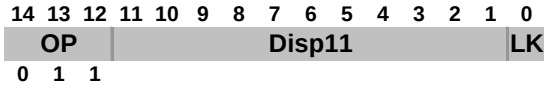
Description

This instruction subtracts general-purpose register rDg0 from register rAg0 and then update the condition flags (N, Z, C, V) according the result.

Exceptions

None

jx! **Jump (and Link)**



Syntax

```
j!      Disp11          (LK = 0)
jl!     Disp11          (LK = 1)
```

where:

<Disp11> Specifies the 11-bit jump 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
```

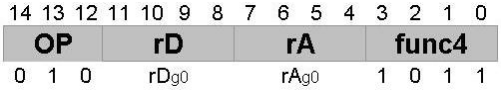
Description

The 11-bit bits displacement address Disp11 is shifted left one bit and combined with the high order seven bits of the address of jump instruction. The program unconditionally execute from the calculated target address. If LK bit sets, the address of the instruction after the jump instruction is placed into the link register GPR_{r3}.

Exceptions

None

lbu! **Load Byte Unsigned**



Syntax

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

where:

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

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

Operation

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

Usage

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

Description

The effective memory address is the general-purpose register rAg0. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: User mode accesses kernel/debug mode address range

lbup! **Load Byte Unsigned with base pointer**

Syntax

```
lbup!    rDg0, Imm5
```

where:

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

<Imm5> Specifies the 5-bit immediate value.

Operation

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

Usage

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

Description

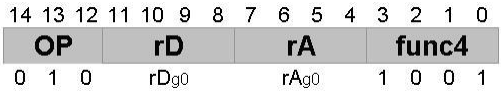
The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate value. The byte in memory addressed by this effective address is zero-extended to 32-bits and loads into general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: User mode accesses kernel/debug mode address range

lh! **Load Half-word signed**



Syntax

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

where:

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

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

Operation

$GPR_{rD}g0 = SignExtend(Mem16[GPR_{rA}g0]);$

Usage

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

Description

The effective memory address is the general-purpose register rAg0. The half-word in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rDg0. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

lhp! **Load Half-word signed with base pointer**

Syntax

```
lhp!     rDg0, Imm5
```

where:

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

<Imm5> Specifies the 5-bit immediate value.

Operation

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

Usage

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

Description

The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate with shifting left one bit. The half-word in memory addressed by this effective address is signed-extended to 32-bits and loads into general-purpose register rDg0. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: User mode accesses kernel/debug mode address range

ldiu! **Load Immediate Unsigned**

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
```

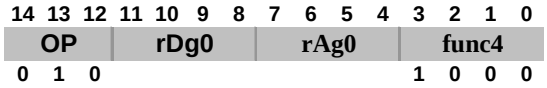
Description

This instruction loads an unsigned 8-bit immediate value (*Imm8)* and places the result in general-purpose register rDg0.

Exceptions

None

lw! **Load Word**



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

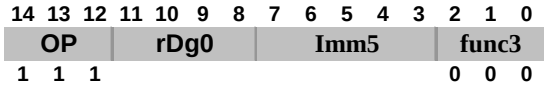
The effective memory address is the general-purpose register rAg0. The word in memory addressed by this effective address loads into general-purpose register rDg0. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

lwp! **Load Word with base pointer**



Syntax

```
lwp!rDg0, Imm5
```

where:

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

<Imm5> Specifies the 5-bit immediate value.

Operation

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

Usage

```
lwp!     R7, 0x0E
```

Description

The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate with shifting left 2-bits. The word in memory addressed by this effective address loads into general-purpose register rDg0. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: User mode accesses kernel/debug mode address range

mlfh! **Move Lower register From Higher register**

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
```

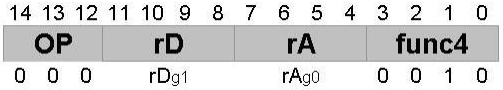
Description

This instruction is a move instruction, which support data transfer between two general-purpose registers. It moves the higher source register (r16~r31) to lower destination register (r0~r15).

Exceptions

None

mhfl! **Move Higher register From Lower register**



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
```

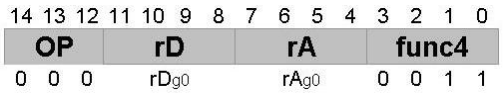
Description

This instruction is a move instruction, which support data transfer between two general-purpose registers. It moves the lower source register (r0~r15) to higher destination register (r16~r31).

Exceptions

None

mv! **Move**



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

This instruction is a move instruction, which support data transfer between two general-purpose registers.

It moves the lower source register (r0~r15) to lower destination register (r0~r15).

Exceptions

None

neg! **Negative**

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 = R[31];   // R = 0 - GPRrAg0
Z = (R==0)? 1 : 0;
C = ~borrow (0 - GPRrAg0);
V = overflow (0 - GPRrAg0);
GPRrDg0 = 0 - GPRrAg0;
```

Usage

```
neg!     r9, r1
```

Description

This instruction subtract zero from the content of general-purpose register rAg0 to get the negative value of the content of general-purpose register rAg0. On the other word, the negative operation performs the add of the one's complement of register rAg0 and the value one and the resulting 2's complement is then placed to general-purpose register rDg0.

Exceptions

None

nop! **No Operation**

Syntax

```
nop!
```

Operation

```
No operation;
```

Usage

```
nop!
```

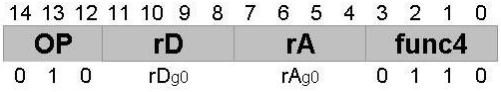
Description

This instruction represents to 16-bit no operation instruction.

Exceptions

None

not! **Logical NOT**



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 = R[31];   // R = ~GPRrAg0
Z = (R==0)? 1 : 0;
GPRrDg0 = ~GPRrAg0;
```

Usage

```
not!     r4g0, r2g0
```

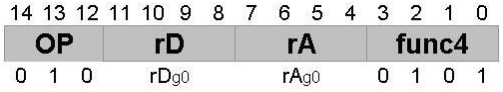
Description

The contents of general-purpose register rAg0 performs a bit-wise NOT operation to generate a 32 bit result. This result is then placed to general-purpose register rDg0.

Exceptions

None

or! **Logical OR**



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 = R[31];   // R = GPRrDg0 | GPRrAg0
Z = (R==0)? 1 : 0;
GPRrDg0 = GPRrDg0 | GPRrAg0;
```

Usage

```
or!      r11, r2
```

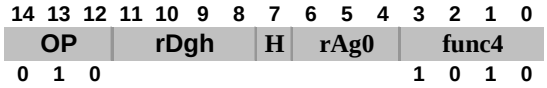
Description

The contents of general-purpose register rDg0 and rAg0 are computed in a bit-wise OR operation to generate a 32 bit result. This result is then placed to general-purpose register rDg0.

Exceptions

None

pop! **Load Post-increment**



Syntax

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

where:

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

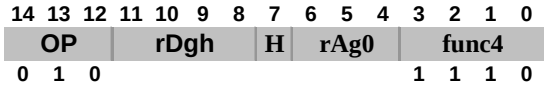
The effective memory address is one of the lower eight general-purpose register (r0~r7). The word in memory addressed by this effective address loads into general-purpose register rDgh. After load operation complete, the base register rAg0 is incremented by four. If H bit field is high, destination register is from higher register (r16~r31), otherwise is from lower register (r0~r15). If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

push! **Store pre-decrement**



Syntax

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

where:

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

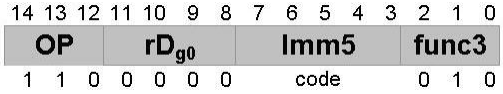
The effective memory address is one of the lower eight general-purpose register (r0~r7) minus four. The word in memory addressed by this effective address stores from general-purpose register rDgh. After store operation complete, the base register is updated with this effective address. If H bit field is high, source register is from higher register (r16~r31), otherwise is from lower register (r0~r15). If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

sdbbp! **Software Debug Breakpoint**



Syntax

```
sdbbp!   code
```

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
```

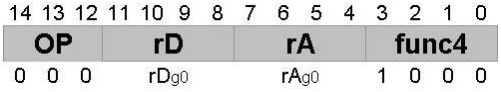
Description

This instruction will induce software debug breakpoint exception, passing control to an exception handler. The exception vector location depends the ice and probe circuit situation, also the corresponding status bits in debug register (DREG) will set and the debug exception program counter (DEPC) will point to the address of *sdbbp* instruction.

Exceptions

None

sll! **Shift Left Logical**



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 = R[31];
// R = { GPRrDg0[(31 - GPRrAg0[4:0]):0], GPRrAg0[4:0]{0}}
Z = (R==0)? 1 : 0;
C = GPRrDg0 [32 - GPRrAg0 [4:0]];
GPRrDg0 = {GPRrDg0[(31 - GPRrAg0[4:0]):0], GPRrAg0[4:0]{0}};
```

Usage

```
sll!     r4, r2
```

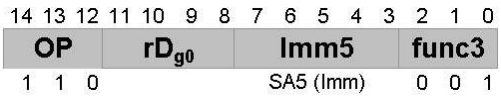
Description

This instruction is a shift left logical instruction. The shift operation only uses the low order 5 bits of register rAg0 (r0~r15) to specify the shift amount (*SA*), then shift left by *SA* bit and insert zeros into the low order bits of general-purpose register rDg0 (r0~r15). This result is placed to general-purpose register rDg0 (r0~r15).

Exceptions

None

slli! **Shift Left Logical with Immediate**



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 = R[31];   // R = { GPRrDg0[(31-SA):0], SA{0}}
Z = (R==0)? 1 : 0;
C = GPRrDg0[32 - SA];
GPRrDg0 = {GPRrDg0[(31-SA):0], SA{0}};
```

Usage

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

Description

This instruction is a shift left logical with immediate instruction. The shift operation uses 5-bit shift amount (*SA*) to shift left by *SA* bit and insert zeros into the low order bits of general-purpose register rDg0 (r0~r15). This result is placed to general-purpose register rDg0.

Exceptions

None

srli! **Shift Right Logical with Immediate**

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 = R[31];   // R = {SA{0}, GPRrDg0[31:SA]}
Z = (R==0)? 1 : 0;
C = GPRrDg0[SA - 1];
GPRrDg0 = {SA{0}, GPRrDg0[31:SA]};
```

Usage

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

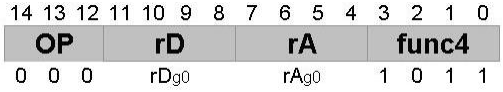
Description

This instruction is a shift right logic instruction. The shift operation uses 5-bit immediate value to specify the shift amount (*SA*), then shift right by *SA* bit and insert zeros into high order bits of general-purpose register rDg0 (r0~r15). This result is placed to general-purpose register rDg0.

Exceptions

None

sra! **Shift Right Arithmetic**



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 = R[31];   // R = {GPR[rA]g0 [4:0]{GPR[rD]g0 [31]}, GPR[rD]g0[31:GPR[rA]g0 [4:0]]}
Z = (R==0)? 1 : 0;
C = GPR[rD]g0[GPR[rA]g0[4:0]-1];
GPR[rD]g0 = {GPR[rA]g0 [4:0]{GPR[rD]g0 [31]}, GPR[rD]g0[31:GPR[rA]g0 [4:0]]};
```

Usage

```
sra!     r11, r12
```

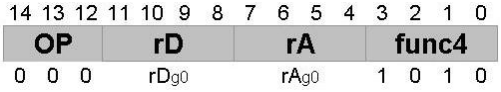
Description

This instruction is a shift right arithmetic instruction. The shift operation only uses the low order 5 bits of register rAg0 to specify the shift amount (*SA*), then shift right by *SA* bit and sign-extend the high order bits of general-purpose register rDg0 (r0~r15). This result is placed to general-purpose register rDg0.

Exceptions

None

srl! **Shift Right Logical**



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 = R[31];   // R = {GPRrAg0[4:0]{0}, GPRrDg0[31: GPRrAg0[4:0]]}
Z = (R==0)? 1 : 0;
C = GPRrDg0[GPRrAg0[4:0] - 1];
GPRrDg0 = {GPRrAg0[4:0]{0}, GPRrDg0[31: GPRrAg0[4:0]]};
```

Usage

```
srl!     r6, r0
```

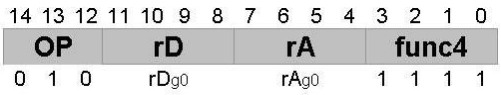
Description

This instruction is a shift right logical instruction. The shift operation only uses the low order 5 bits of register rAg0 (r0~r15) to specify the shift amount (*SA*), then shift right by *SA* bit and insert zeros into high order bits of general-purpose register rDg0 (r0~r15). This result is placed to general-purpose register rDg0.

Exceptions

None

sb! **Store Byte**



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_rAg0] = GPR_rDg0[7:0];
```

Usage

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

Description

The effective memory address is the general-purpose register rAg0. The byte in memory addressed by this effective address stores by the lower 8-bits of general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: User mode accesses kernal/debug address range

sbp! **Store Byte with base point**

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 effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate value. The byte in memory addressed by this effective address stores by the lower 8-bits of general-purpose register rDg0.

Exceptions

Bus error exception

Address error exception: User mode accesses kernal/debug address range

sh! **Store Half-word**

Syntax

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

where:

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

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

Operation

```
Mem16[GPRrAg0] = GPRrDg0[15:0];
```

Usage

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

Description

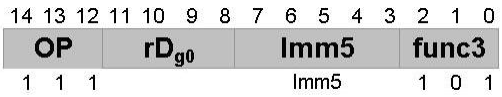
The effective memory address is the general-purpose register rAg0. The half-word in memory addressed by this effective address stores by the lower 16-bits of general-purpose register rDg0. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

shp! **Store Half-word with base pointer**



Syntax

```
shp!     rDg0, Imm5
```

where:

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

<Imm5> Specifies the 5-bit immediate value.

Operation

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

Usage

```
shp!     r6, 0x07
```

Description

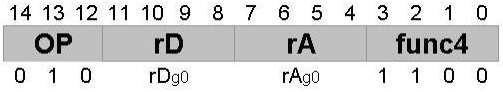
The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate with shifting left one bit. The half-word in memory addressed by this effective address stores by the lower 16-bits of general-purpose register rDg0. If the effective address is not half-word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: User mode accesses kernal/debug address range

sw! **Store Word**



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_rAg0] = GPR_rDg0[31:0];
```

Usage

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

Description

The effective memory address is the general-purpose register rAg0. The word in memory addressed by this effective address stores by the general-purpose register rDg0. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception

swp! **Store Word with base pointer**

Syntax

```
swp!    rDg0, Imm5
```

where:

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

<Imm5> Specifies the 5-bit immediate value.

Operation

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

Usage

```
swp!    r8, 0x0E
```

Description

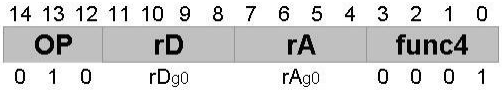
The effective memory address is the sum of base pointer register (r2) and 5-bit unsigned immediate with shifting left 2-bits. The word in memory addressed by this effective address stores by the general-purpose register rDg0. If the effective address is not word aligned, address alignment error exception will occur.

Exceptions

Bus error exception

Address error exception: User mode accesses kernal/debug address range

sub! **Subtract**



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 = R[31];   // R = GPR[rD]g0 - GPR[rA]g0
Z = (R==0)? 1 : 0;
C = ~borrow (GPR[rD]g0 - GPR[rA]g0);
V = overflow (GPR[rD]g0 - GPR[rA]g0);
GPR[rD]g0 = GPR[rD]g0 - GPR[rA]g0;
```

Usage

```
sub!     r14, r1
```

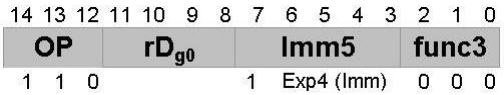
Description

The contents of general-purpose register rDg0 are subtracted from register rAg0 to generate a 32 bit result. This result is then placed to general-purpose register rDg0.

Exceptions

None

subei! **Subtract with Exponent Immediate**



Syntax

```
subei!   rDg0, Exp4            (CU = 1)
```

where:

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

<Exp4> Specifies the 4-bit exponent immediate value.

Operation

```
N = R[31];   // R = GPRrDg0 - 2^Exp4
Z = (R==0)? 1 : 0;
C = carry (GPRrDg0 - 2^Exp4);
V = overflow (GPRrDg0 - 2^Exp4);
GPRrDg0 = GPRrDg0 - 2^Exp4;
```

Usage

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

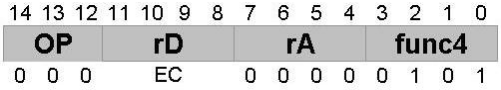
Description

The contents of 32-bit general-purpose register rDg0 are subtracted from immediate, 2^{Exp4} to generate a 32-bit result. This result is then placed to general-purpose register rDg0. Also the condition flags (N, Z, C, V) always update by this instruction.

Exceptions

None

t{cond}! **test and set T flag on condition**



Syntax

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

where:

`t` Specifies the condition flag T. T flag is used in parallel condition execution, it determines 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 branch condition (EC) 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!
```

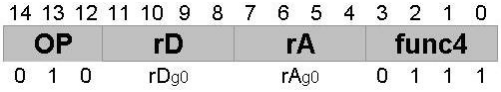
Description

The t<*cond*>! instructions are used to set/clear T flag according to the result of condition flag test. The EC field specifies the corresponding condition flag test.

Exceptions

None

xor! **Logical XOR**



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 = R[31];   // R = GPRrDg0 ^ GPRrAg0
Z = (R==0)? 1 : 0;
GPRrDg0 = GPRrDg0 ^ GPRrAg0;
```

Usage

```
xor!     r9, r13
```

Description

The contents of 32-bit general purpose register rAg0 and rDg0 are combined in a bit-wise XOR operation to generate a 32-bit result. This result is then placed to general-purpose register rDg0.

Exceptions

None

2.3 Synthetic Instruction Sets

subrix **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 equivant to
addri    rD, rA, -SImm14       (CU = 0)
addri.c  rD, rA, -Simm14       (CU = 1)
```

Usage

```
subri    r4, r3, 0x0123
```

subix **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 equivant to
addi     rD, -SImm16       (CU = 0)
addi.c   rD, -Simm16       (CU = 1)
```

Usage

```
subi     r4, 0x1234
```

subisx **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 equivant to
addis    rD, -Imm16          (CU = 0)
addis.c  rD, -Imm16          (CU = 1)
```

Usage

```
subis    r4, 0x1234
```

li **Load immediate to register**

Syntax

```
li      rD, Imm32
li      rD, LABEL
```

where:

<rD> Specifies the destination general-purpose register.

<Imm32> Specifies 32-bit immediate value.

<LABEL > Specifies the label name

Operation

1.

If ((Imm32 value> -32768) & (Imm32 <32767))

Then li rD, Imm32 is equivalent to

ldi rD, Imm32

else li rD, Imm32 is equivalent to

ldis rD, Hi_{16}(Imm32)

ori rD, Lo_{16}(Imm32)

2.

li rD, LABEL is equivalent to

ldis rD, Hi_{16}(LABEL)

ori rD, Lo_{16}(LABEL)

Usage

```
li      r4, 0x7234
li      r5,0x80001234
li      r6,label
label:
```

la **Load immediate to register**

Syntax

```
la      rD, Imm32
la      rD, LABEL
```

where:

<rD> Specifies the destination general-purpose register.

<Imm32> Specifies 32-bit immediate value.

<LABEL > Specifies the label name

Operation

1.

If ((Imm32 value> -32768) & (Imm32 <32767))

Then `la rD, Imm32` is equivalent to

```
ldi     rD, Imm32
```

else `la rD, Imm32` is equivalent to

```
ldis    rD, Hi16(Imm32)
ori     rD, Lo16(Imm32)
```

2.

`la rD, LABEL` is equivalent to

```
ldis    rD, Hi16(LABEL)
ori     rD, Lo16(LABEL)
```

Usage

```
la      r4, 0x7234
la      r5,0x80001234
la      r6,label
label:
```

lb **Load Byte signed**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
lb      rD, [rA,0]
```

Usage

```
lb      r6,[r5]
```

lbu **Load Byte unsigned**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
lbu     rD, [rA,0]
```

Usage

```
lbu     r6,[r5]
```

lh **Load half-word signed**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
lb      rD, [rA,0]
```

Usage

```
lb      r6,[r5]
```

lhu **Load half-word unsigned**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
lhu     rD, [rA,0]
```

Usage

```
lhu     r6,[r5]
```

lw **Load word**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
lw      rD, [rA,0]
```

Usage

```
lw      r6,[r5]
```

sb **Store byte**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
sb      rD, [rA,0]
```

Usage

```
sb      r6,[r5]
```

sh **Store half-word**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
sh      rD, [rA,0]
```

Usage

```
sh      r6,[r5]
```

sw **Store word**

Syntax

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

where:

<rD> Specifies the destination general-purpose register.

<rA> Specifies the base address register.

Operation

This instruction is equivant to

```
sw      rD, [rA,0]
```

Usage

```
sw      r6,[r5]
```

mul **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 **Unsigned multiply**

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 **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 **Unsigned divide**

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 **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 **Unsigned divide**

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 Compiler Options

To run the C compiler, type: gcc

Usage:gcc [option|file]

-mSCORE5U	S+core5U compiler option
-mSCORE5	S+core5 compiler option
-mSCORE7	S+core7 compiler option
-meb/-mel	Big/Little endian option
-S	compile to assembly language
-E	run only the preprocessor on the named C programs
-o *file*	place output in file `*file*'
--help	print a description of the command line options recognized by gcc
-ansi	support all ANSI standard C programs. 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 strict ANSI C and ISO C++
-w	inhibit all warning messages
-Wall	enable all the warnings about constructions that some 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 each function name as it is compiled, and print some statistics about each pass when it finishes.
-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*	undefine 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 turns on all optimizations -O2 does. In addition, turn function in-lining on.
-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 …….)

Usage:

Example 1:

In DOS command line, type:

gcc –mSCORE5U -S test.c -o test.s

The test.c is a C code file. It generates an asm file with name of test.s.

Example2:

gcc –mSCORE5U -S –gstabs+ test.c -o test.s

The test.c is a C code file. It generates an asm file with name of test.s, which includes the debug information.

Example3:

gcc –mSCORE5U -S -O2 test.c -o test.s

The test.c is a c code file. It generates an optimized asm file with name of test.s.

Example4:

gcc –mSCORE5U -S -O2 –gstabs+ test.c -o test.s

The test.c is a c code file. It 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 S+core7 C Compiler basic data types

Data type	Size bits
char / unsigned char	8
short / unsigned short	16
int / unsigned int	32
long / unsigned long	32
long long / unsigned long long	64
float	32 bits IEEE floating point format
double	64 bits IEEE floating point format

3.3 S+core7 C Compiler Calling Convention

S+core 7 have 32 general registers, we divide them into several register class(list in following table),

reg#	Symbolic	description
0	sp	Stack pointer
1	at	Assembler temp
2	bp	Base pointer
3	lr	Link register
4-5	a0-a1	Argument 0-1/Return value
6-7	a2-a3	Argument 2-3
8-11	t0-t3	Caller save registers
12-15	s0-s3	Callee save registers
16-21	s4-s9	Callee save registers
22-27	t4-t9	Caller save registers
28		Pic register
29	jp	Jump register
30-31	k0-k1	Kernel scratch

The C Compiler pass 1st ~ 4th parameter (if it small than 32-bit) using register a0-a3, and put the return value in register v0. If the function has more than 4 parameters, the 5th parameter will be place in memory [sp+16], 6th will be place in [sp+20] and so on

Usage:

Examples

Here we list an example of arguments passing. Left side is the C source and assembly code generated bye the compiler listed in right size. We can see that the 5th parameter (5) will be place in [sp+16] and 6th parameter will be place 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 Reference

- http://gcc.gnu.org
- http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc

4 GNU Assembler for S+core

4.1 Command Line Options

4.1.1 S+core Operation

Argument List	Comments
-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
--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*

4.1.2 Command line 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 characterics A to Z and a to z ,the digit 0 to 9 , as well as "_", "-" and "$".Note you cannot start a symbol with a digit,

A comment is anything that is followed by "@" Everything that is followed by "@" is ignored to the end of the line.C-style comments (using /* and */)are also allowed .You can also use "#" instead of "@".

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 are discussed in later below.

4.3 Assembler Directive

All assembler directives have names that begin with a full-stop ".".The list of 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]]
```

Usage :

```
.align  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
```

.ascii

Syntax:

```
.ascii strings
```

Usage :

```
.ascii "JNZ"  @inserts the byte 0x4a 0x4e 0x5a
```

.asciz

Syntax:

```
.asciz strings
```

Usage :

```
.asciz "JNZ"  @inserts the byte 0x4a 0x4e 0x5a 0x00
```

.balign

Syntax :

```
.balign <power_2>[, [fill][, max]]
```

Usage :

```
.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 exptessions
```

Usage :

```
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
```

Usage : .

.data

Syntax:

```
.data [subsection]
```

Usage :

```
.data   @ switch to data section
```

.equ

Syntax:

```
.equ symbol,expression
```

Usage :

```
.equ zzz,(5*8)+2 @  zzz is absolute symbol whose address is 42
This is equivalent to zzz = 42
```

.extern

Syntax:

```
.extern symbol
```

Usage :

```
.extern label_zzz     @ specify label_zzz is defined in other source file
```

.global

Syntax:

```
.global symbol
```

Usage :

```
.global _start   @Specify _start is globally visiable to all other module ,
including linker .
```

.hword

Syntax:

```
.hword expression
```

Usage :

```
.hword   0xaa55,12345 @inserts the bytes 0x55,0xaa,0x39,30
```

.include

Syntax:

```
.include filename
```

Usage :

```
.include "aaa"   @ same as #include file aaa
```

.int

Syntax:

```
.int exptessions
```

Usage :

```
.int aaa     @ inserts 4 bytes in uninitialized section for symbol aaa
```

.org

Syntax:

```
.org new-lc[,fill]
```

Usage :

```
.org 0x1234  @advance the location counter of the current section to 0x1234.
```

.section

Syntax:

```
.section <section-name> {,"<flags>"}
```

Usage :

```
.section .text1,wa   @start a text1 section which are writable and
allowable.
```

.set

Syntax:

```
.set symbol,expression
```

Usage :

```
.set nor1    @  from current position , if using r1(temp register) ,
assembler will pop warn messages
.set r1      @  from current position , if using r1(temp register) ,
assembler will not pop warn messages
lw r1,[r2]   @ assembler will not pop warn message ,though using r1
.set nor1    @ from current position , if using r1(temp register),assembler
will pop warn messages
```

.space

Syntax:

```
.space size[,fill]
```

Usage :

```
.space 100   @  will insert 100 bytes of 0x00 from current position.
.space  100,0x11 @ will insert 100 bytes of 0x11 from current position.
```

.text

Syntax:

```
.text
```

Usage :

```
.text   @ switch to .text section
```

.word

Syntax:

```
.word expression
```

Usage :

```
.word 0xdeadbeaf @ inserts the bytes 0xaf 0xbe 0xad 0xde
```

4.4 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 purpose. 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 useing 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 Option

Argument List	Comments
-l*archive*	Add archive file *archive* to the list of files to link
-L*searchdir*	Add path *searchdir* to the list of paths that ld will search for archive libraries and ld control scripts.
-M	Print a link map to the standard output
-o *output*	Use *output* as the name for the program produced by ld
-EB	Link big-endian objects (default)
-EL	Link little-endian objects
T*scriptfile*	Read link commands from the file *commandfile*. These commands replace ld's default link script

Usage :

Score-linux-elf-ld –Texec.ld test.o –lc –L .–o test.exec

6 Appendix

6.1 Appendix A : instructions 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 16bit 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
Locical	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!
	32 bit NOT :notx 16 bit NOT : not!

Contents	instructions
	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 clear : bitclr.c bit set:: bitset.c bit toggle : bittgl.c bit test : bittst.c 16 bit clear : bitclr! 16 bit set : bitset! 16 bit toggle : bittgl! 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 ritht 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 arithematic: srax Shift right arithematic with immediate : sraix 16 bit shift right arithematic: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

Contents	instructions
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}!
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 combine begin : lcb Load combine word :lcw Load combine end : lce Store combine begin : scb Store combine word : scw Store combine 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 curstom 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 Appendix B : Instruction Alphabet table

A	addx addcx addix addisx addrix andx andx andisx andrix add! addc! addei! and!
B	b{cond} b{cond}l bitclr.c bitset.c bittgl.c bittst.c br{cond} br{cond}l b{cond}! bitclr! bitset! bittgl! 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!
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 mfcr mfcx mfccx mfsr mtcex mtcr mtcx mtccx mtsr mv{cond} mul mulu mlfh! mhfl! 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 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 Appendix C:Assembler Directive index

GNU Assembly Directive	Description
.align	Pad the location counter (in the current subsection) to a particular storage boundary
.ascii	Inserts the string into the assembly , with no null character.
.asciz	Like ".ascii" , but follows the string with a zero byte.
.balign	Aligns the address to <power_2>bytes
.byte	Insert the byte value of the expression into the object file.Expressions are separated by comments.
.comm	declares a common symbol named *symbol*
.data	Switch the destinition 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 is to made globally visiable to all other module that are part of executable and visiable to linker.
.hword	Inserts the (16-bit) half-word value of the expression into the object file.
.include	Inserts 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
.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	Inserts the (32-bit) word value of expression into the object file.

6.4 Reference

S+core Binutils is ported from gnu Binutils 2.13.2.1 version .

Using as : http://www.gnu.org/software/binutils

Using ld The GNU linker : http://www.gnu.org/software/binutils