TEMIC
Semiconductors

SPARC V7.0

Instruction Set

for Embedded Real time 32–bit Computer

(ERC32)

for SPACE Applications

Instruction Set

1. Assembly Language Syntax

The notations given in this section are taken from Sun's SPARC Assembler and are used to describe the suggested assembly language syntax for the instruction definitions given in Section 6.2.

Understanding the use of type fonts is crucial to understanding the assembly language syntax in the instruction definitions. Items in `typewriter font` are literals, to be entered exactly as they appear. Items in *italic font* are metasymbols that are to be replaced by numeric or symbolic values when actual assembly language code is written. For example, *asi* would be replaced by a number in the range of 0 to 255 (the value of the bits in the binary instruction), or by a symbol that has been bound to such a number.

Subscripts on metasymbols further identify the placement of the operand in the generated binary instruction. For example, *regrs2* is a *reg* (i.e., register name) whose binary value will end up in the *rs2* field of the resulting instruction.

1.1. Register Names

reg

A *reg* is an integer unit register. It can have a value of:

%0	through	%31	all integer registers
%g0	through	%g7	global registers—same as %0 through %7
%o0	through	%o7	out registers—same as %8 through %15
%l0	through	%l7	local registers—same as %16 through %23
%i0	through	%i7	in registers—same as %24 through %31

Subscripts further identify the placement of the operand in the binary instruction as one of the following:

reg_{rs1}	—*rs1* field
reg_{rs2}	—*rs2* field
reg_{rd}	—*rd* field

freg

A *freg* is a floating-point register. It can have a value from %f0 through %f31. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$freg_{rs1}$	—*rs*1 field
$freg_{rs2}$	—*rs*2 field
$freg_{rd}$	—*rd* field

creg

A *creg* is a coprocessor register. It can have a value from %c0 through %c31. Subscripts further identify the placement of the operand in the binary instruction as one of the following:

$creg_{rs1}$	—*rs*1 field
$creg_{rs2}$	—*rs*2 field
$creg_{rd}$	—*rd* field

1.2. Special Symbol Names

Certain special symbols need to be written exactly as they appear in the syntax table. These appear in `typewriter font,` and are preceded by a percent sign `(%)`. The percent sign is part of the symbol name; it must appear as part of the literal value.

The symbol names are:

`%psr`	Processor State Register
`%wim`	Window Invalid Mask register
`%tbr`	Trap Base Register
`%y`	Y register
`%fsr`	Floating-point State Register
`%csr`	Coprocessor State Register
`%fq`	Floating-point Queue
`%cq`	Coprocessor Queue
`%hi`	Unary operator that extracts high 22 bits of its operand
`%lo`	Unary operator that extracts low 10 bits of its operand

1.3. Values

Some instructions use operands comprising values as follows:

simm13—A signed immediate constant that fits in 13 bits
const22—A constant that fits in 22 bits
asi—An alternate address space identifier (0 to 255)

1.4. Label

A label is a sequence of characters comprised of alphabetic letters (a–z, A–Z (upper and lower case distinct)), underscore (_), dollar sign ($), period (.), and decimal digits (0–9), but which does not begin with a decimal digit.

Some instructions offer a choice of operands. These are grouped as follows:

regaddr:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$

address:

reg_{rs1}
$reg_{rs1} + reg_{rs2}$
$reg_{rs1} + simm13$
$reg_{rs1} - simm13$
$simm13$
$simm13 + reg_{rs1}$

reg_or_imm:

reg_{rs2}
$simm13$

1.5. Instruction Mnemonics

Figure 1.1 illustrates the mnemonics used to describe the SPARC instruction set. Note that some combinations possible in *Figure 1.1* do not correspond to valid instructions (such as store signed or floating-point convert extended to extended). Refer to the instruction summary on page 6–6 for a list of valid SPARC instructions.

Data Transfer

[**L**oa**D** / **ST**ore] [[**S**igned / **U**nsigned] [**B**yte / **H**alfword / word / **D**ouble word] [normal / **A**lternate] ; [single / **D**ouble] [**F**loating-point / **C**oprocessor] [register / **S**tatus **R**egister / **Q**ueue]]

atomic **SWAP** word

atomic **L**oad-**S**tore **U**nsigned **B**yte

Integer Operations

[**AND** / **OR** / **XOR**] [normal / **N**ot] [normal / set **CC**]

[**ADD** / **SUB**] [normal / e**X**tended] [normal / set **CC**]

[**R**ea**D** / **WR**ite] [**Y** / **PSR** / **WIM** / **TBR**]

Shift [**L**eft / **R**ight] [**L**ogical / **A**rithmetic]

Tagged [**ADD** / **SUB**] set **CC** [normal / **T**rap o**V**erflow]

MULtiply **S**tep set **CC**
SETHI
SAVE
RESTORE

Floating-Point Operations

Fp convert [**I**nteger / **S**ingle / **D**ouble / e**X**tended] **TO** [**I**nteger / **S**ingle / **D**ouble / e**X**tended]

Fp [**MOV**e / **NEG**ate / **ABS**olute] **S**ingle

Fp [**ADD** / **SUB**tract / **MUL**tiply / **DIV**ide / **SQ**uare **R**oo**T** / **C**o**MP**are / **C**o**MP**are and **E**xception] [**S**ingle / **D**ouble / e**X**tended]

Control Transfer

Branch [**I**nteger **CC** / **F**loating-point **CC** / **C**oprocessor **CC**] normal / **A**null delay instruction

Ju**MP** and **L**ink
RETurn from **T**rap

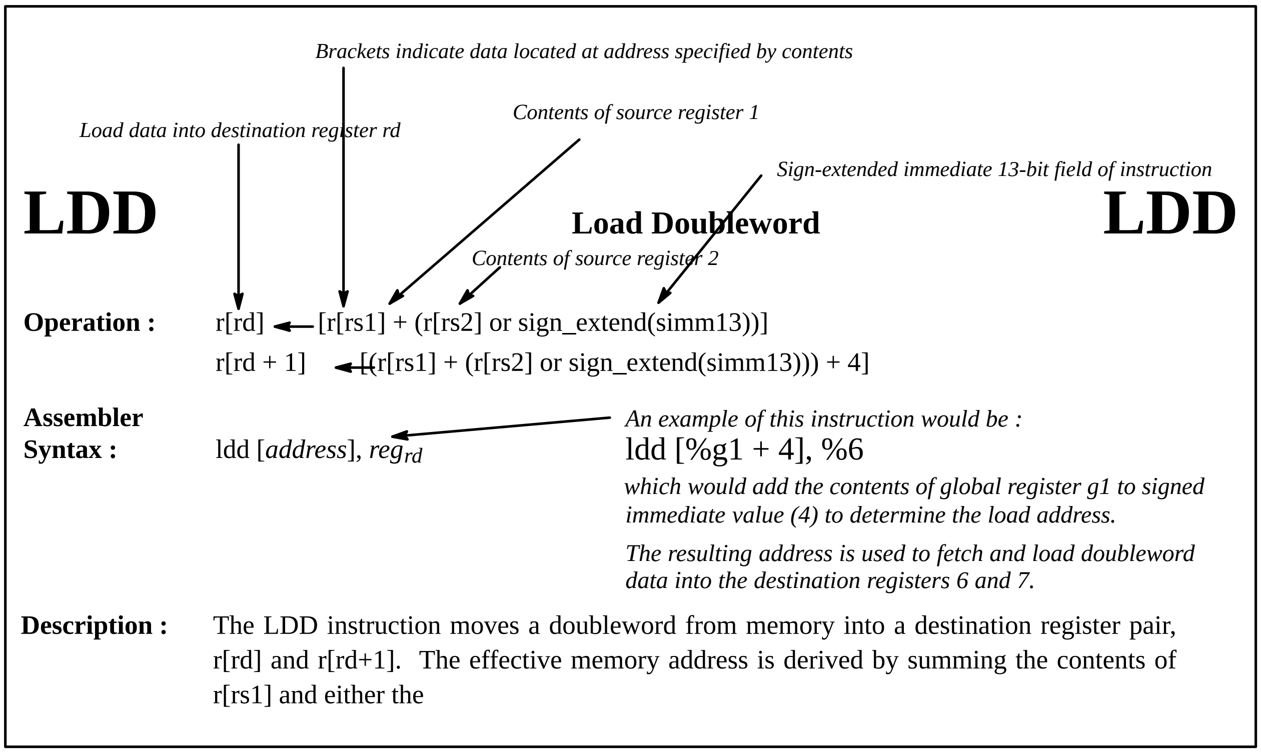
CALL
Trap on **I**nteger **CC**

Figure 1.1. SPARC Instruction Mnemonic Summary

2. Definitions

This section provides a detailed definition for each CY7C601 instruction. Each definition includes: the instruction operation; suggested assembly language syntax; a description of the salient features, restrictions and trap conditions; a list of synchronous or floating-point\coprocessor traps which can occur as a consequence of executing the instruction; and the instruction format and op codes. Instructions are defined in alphabetical order with the instruction mnemonic shown in large bold type at the top of the page for easy reference. The instruction set summary that precedes the definitions, (*Table 1.2*), groups the instructions by type.

Table 1.1 identifies the abbreviations and symbols used in the instruction definitions. An example of how some of the description notations are used is given below in *Figure 1.2*. Register names, labels and other aspects of the syntax used in these instructions are described in the previous section.


Brackets indicate data located at address specified by contents
Contents of source register 1
Load data into destination register rd
Sign-extended immediate 13-bit field of instruction
Contents of source register 2

LDD **Load Doubleword** **LDD**

Operation : r[rd] ← [r[rs1] + (r[rs2] or sign_extend(simm13))]
r[rd + 1] ← [(r[rs1] + (r[rs2] or sign_extend(simm13))) + 4]

Assembler Syntax : ldd [*address*], *reg*$_{rd}$

An example of this instruction would be :
ldd [%g1 + 4], %6
which would add the contents of global register g1 to signed immediate value (4) to determine the load address.

The resulting address is used to fetch and load doubleword data into the destination registers 6 and 7.

Description : The LDD instruction moves a doubleword from memory into a destination register pair, r[rd] and r[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the

Figure 1.2. Instruction Description

Table 1.1. Instruction Description Notations

Symbol	Description
a	Instruction field that controls instruction annulling during control transfers
AND, OR XOR, etc.	AND, OR, XOR, etc operators
asr_reg	Any implemented ASR (Ancillary State)
c	The icc carry bit
ccc	The coprocessor condition code field of the CCSR
CONCAT	Concatenate
cond	Instruction field that selects the condition code test for branches

Symbol	Description
creg	Communication Coprocessor Register : can be %ccsr, %ccfr, %ccpr, %cccrc
CWP	PSR‘s Current Window Pointer field
disp22	Instruction field that contains the 22-bit sign-extended displacement for branches
disp30	Instruction field that contains the 30-bit word displacement for calls
EC	PSR‘s Enable Coprocessor bit
EF	PSR‘s Enable FPU bit
ET	PSR‘s Enable Traps bit
i	Instruction field that selects rs2 or sign_extend(simm13) as the second operand
icc	The integer condition code field of the PSR
imm22	Instruction field that contains the 22-bit constant used by SETHI
n	The icc negative bit
not	Logical complement operator
nPC	next Program Counter
opc	Instruction field that specifies the count for Coprocessor-operate instructions
operand2	Either r[rs2] or sign_extend(simm13)
PC	Program Counter
pS	PSR‘s previous Supervisor bit
PSR	Processor State Register
r[15]	A directly addressed register (could be floating-point or coprocessor)
rd	Instruction field that specifies the destination register (except for store)
r[rd]	Depending on context, the integer register (or its contents) specified by the instruction field, e.g. , rd, rs1, rs2
r[rd]<31>	<> are used to specify bit fields of a particular register or I/O signal
[r[rs1] + r[rs2]]	The contents of the address specified by r[rs1] + r[rs2]
rs1	Instruction field that specifies the source 1 register
rs2	Instruction field that specifies the source 2 register
S	PSR‘s Supervisor bit
shcnt	Instruction field that specifies the count for shift instructions
sign_extend(simm13)	Instruction field that contains the 13-bit, sign-extended immediate value
Symbol	Description
TBR	Trap Base Register
tt	TBR‘s trap type field
uf	Floating-point exception : underflow

Symbol	Description
v	The icc overflow bit
WIM	Window Invalid Mask register
Y	Y Register
z	The icc zero bit
-	Subtract
x	Multiply
/	Divide
<--	Replaced by
7FFFFFF H	Hexadecimal number representation
+	Add

Table 1.2. Instruction Set Summary

Name		Operation		Cycles
Load and Store Instructions	LDSB(LDSBA*)	Load Signed Byte	(from Alternate Space)	2
	LDSH(LDSHA*)	Load Signed Halfword	(from Alternate Space)	2
	LDUB(LDUBA*)	Load Unsigned Byte	(from Alternate Space)	2
	LDUH(LDUHA*)	Load Unsigned Halfword	(from Alternate Space)	2
	LD(LDA*)	Load Word	(from Alternate Space)	2
	LDD(LDDA*)	Load Doubleword	(from Alternate Space)	3
	LDF	Load Floating Point		2
	LDDF	Load Double Floating Point		3
	LDFSR	Load Floating Point State Register		2
	LDC	Load Coprocessor		2
	LDDC	Load Double Coprocessor		3
	LDCSR	Load Coprocessor State Register		2
	STB(STBA*)	Store Byte	(into Alternate Space)	3
	STH(STHA*)	Store Halfword	(into Alternate Space)	3
	ST(STA*)	Store Word	(into Alternate Space)	3
	STD(STDA*)	Store Doubleword	(into Alternate Space)	4
	STF	Store Floating Point		3
	STDF	Store Double Floating Point		4
	STFSR	Store Floating Point State Register		3
	STDFQ*	Store Double Floating Point Queue		4
	STC	Store Coprocessor		3
	STDC	Store Double Coprocessor		4
	STCSR	Store Coprocessor State Register		3
	STDCQ*	Store Double Coprocessor Queue		4
	LDSTUB(LDSTUBA*)	Atomic Load/Store Unsigned Byte	(in Alternate Space)	4
	SWAP(SWAPA*)	Swap r Register with Memory	(in Alternate Space)	4
Arithmetic/Logical/Shift	ADD(ADDcc)	Add	(and modify icc)	1
	ADDX(ADDXcc)	Add with Carry	(and modify icc)	1
	TADDcc(TADDccTV)	Tagged Add and modify icc	(and Trap on overflow)	1
	SUB(SUBcc)	Subtract	(and modify icc)	1
	SUBX(SUBXcc)	Subtract with Carry	(and modify icc)	1
	TSUBcc(TSUBccTV)	Tagged Subtract and modify icc	(and Trap on overflow)	1
	MULScc	Multiply Step and modify icc		1
	AND(ANDcc)	And	(and modify icc)	1
	ANDN(ANDNcc)	And Not	(and modify icc)	1
	OR(ORcc)	Inclusive Or	(and modify icc)	1
	ORN(ORNcc)	Inclusive Or Not	(and modify icc)	1
	XOR(XORcc)	Exclusive Or	(and modify icc)	1
	XNOR(XNORcc)	Exclusive Nor	(and modify icc)	1
	SLL	Shift Left Logical		1
	SRL	Shift Right Logical		1
	SRA	Shift Right Arithmetic		1
	SETHI	Set High 22 Bits of r Register		1
	SAVE	Save caller's window		1
	RESTORE	Restore caller's window		1
Control Transfer	Bicc	Branch on Integer Condition Codes		1**
	FBicc	Branch on Floating Point Condition Codes		1**
	CBccc	Branch on Coprocessor Condition Codes		1**
	CALL	Call		1**
	JMPL	Jump and Link		2**
	RETT	Return from Trap		2**
	Ticc	Trap on Integer Condition Codes		1 (4 if Taken)
Read/Write Control Registers	RDY	Read Y Register		1
	RDPSR*	Read Processor State Register		1
	RDWIM*	Read Window Invalid Mask		1
	RDTBR*	Read Trap Base Register		1
	WRY	Write Y Register		1
	WRPSR*	Write Processor State Register		1
	WRWIM*	Write Window Invalid Mask		1
	WRTBR*	Write Trap Base Register		1
	UNIMP	Unimplemented Instruction		1
	IFLUSH	Instruction Cache Flush		1
FP (CP) Ops	FPop	Floating Point Unit Operations		1 to Launch
	CPop	Coprocessor Operations		1 to Launch

* privileged instruction

** assuming delay slot is filled with useful instruction

ADD — Add — ADD

Operation: r[rd] $\leftarrow$ r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: add *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: The ADD instruction adds the contents of the register named in the *rs1* field ,r[rs1], to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The result is placed in the register specified in the *rd* field.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 0 0 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 0 0 0 0 0	rs1	i=1	simm13

ADDcc — Add and modify icc — ADDcc

Operation:

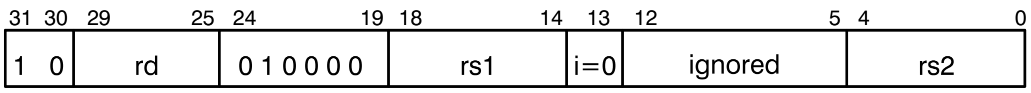
```
r[rd] ← r[rs1] + operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
c ← (r[rs1]<31> AND operand2<31>)
        OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
```

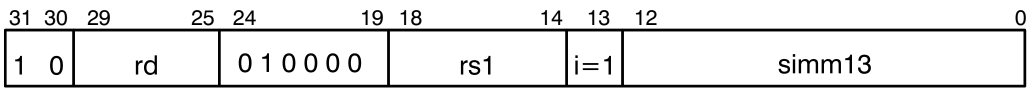
Assembler Syntax: addcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: ADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction’s *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition, ADDcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:





ADDX Add with Carry ADDX

Operation: r[rd] ← r[rs1] + (r[rs2] or sign extnd(simm13)) + c

Assembler Syntax: addx *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$*

Description: ADDX adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 0 1 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 0 1 0 0 0	rs1	i=1	simm13

ADDXcc Add with Carry and modify icc ADDXcc

Operation:

r[rd] $\leftarrow$ r[rs1] + operand2 + c, where operand2 = (r[rs2] or sign extnd(simm13))
n $\leftarrow$ r[rd]<31>
z $\leftarrow$ if r[rd] =0 then 1, else 0
v $\leftarrow$ (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
  OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
c $\leftarrow$ (r[rs1]<31> AND operand2<31>)
  OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))

Assembler Syntax: addxcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: ADDXcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. It then adds the PSR's carry bit (*c*) to that result. The final result is placed in the register specified in the *rd* field. ADDXcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 1 1 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 1 1 0 0 0	rs1	i=1	simm13

AND — And — AND

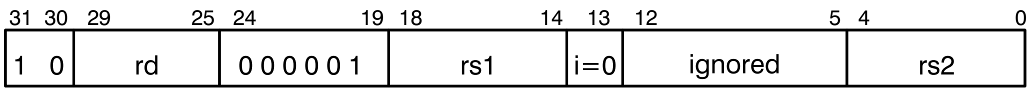
Operation: r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))

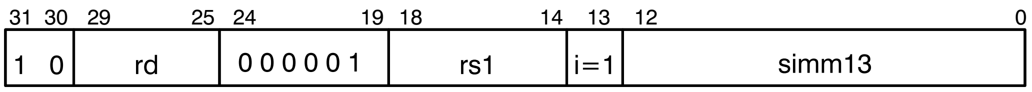
Assembler Syntax: and *reg$_{rs1}$*, *reg_or_imm*, *reg$_{rd}$*

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ANDcc — And and modify icc — ANDcc

Operation:

r[rd] ← r[rs1] AND (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: andcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical AND of the contents of register r[rs1] with either the contents of r[rs2] (if if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i=1). The result is stored in register r[rd]. ANDcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 1 0 0 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 1 0 0 0 1	rs1	i=1	simm13

ANDN And Not ANDN

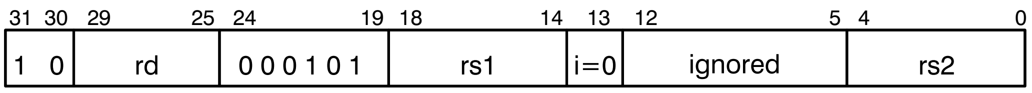
Operation: $r[rd] \leftarrow r[rs1]$ AND $\overline{(r[rs2] \text{ or sign extnd(simm13)})}$

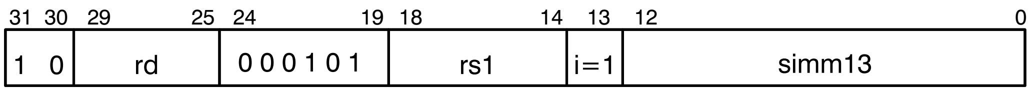
Assembler Syntax: andn *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: ANDN does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ANDNcc And Not and modify icc ANDNcc

Operation:

$r[rd] \leftarrow r[rs1]$ AND $\overline{(r[rs2] \text{ or sign extnd}(simm13))}$

$n \leftarrow r[rd]<31>$

$z \leftarrow$ if r[rd] =0 then 1, else 0

$v \leftarrow 0$

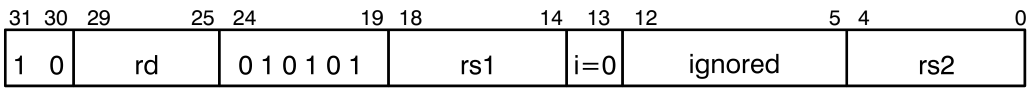
$c \leftarrow 0$

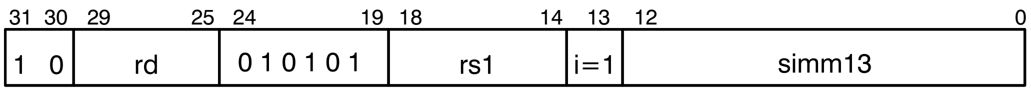
Assembler Syntax: andncc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: ANDNcc does a bitwise logical AND of the contents of register r[rs1] with the logical compliment (not) of either r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ANDNcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





Bicc Integer Conditional Branch Bicc

Operation:

```
PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
        else nPC ← nPC + 4
```

Assembler Syntax:

ba{,a}	*label*	
bn{,a}	*label*	
bne{,a}	*label*	synonym: bnz
be{,a}	*label*	synonym: bz
bg{,a}	*label*	
ble{,a}	*label*	
bge{,a}	*label*	
bl{,a}	*label*	
bgu{,a}	*label*	
bleu{,a}	*label*	
bcc{,a}	*label*	synonym: bgeu
bcs{,a}	*label*	synonym: blu
bpos{,a}	*label*	
bneg{,a}	*label*	
bvc{,a}	*label*	
bvs{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The Bicc instructions (except for BA and BN) evaluate specific integer condition code combinations (from the PSR's *icc* field) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. Refer to Section NO TAG for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section NO TAG regarding delay-branch instructions.

Branch Never (BN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

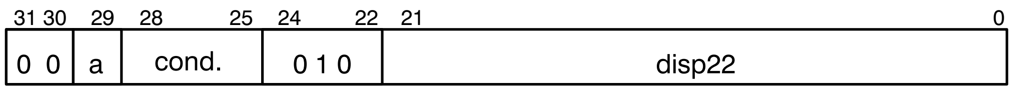
Branch Always (BA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a=1$, the delay instruction is annulled; if $a=0$, the delay instruction is executed.

The delay instruction following a Bicc (other than BA) should not be a delayed-control-transfer instruction. The results of following a Bicc with another delayed control transfer instruction are implementation-dependent and therefore unpredictable.

Traps: none

Mnemonic	Cond.	Operation	icc Test
BN	0000	Branch Never	No test
BE	0001	Branch on Equal	z
BLE	0010	Branch on Less or Equal	z OR (n XOR v)
BL	0011	Branch on Less	n XOR v
BLEU	0100	Branch on Less or Equal, Unsigned	c OR z
BCS	0101	Branch on Carry Set (Less than, Unsigned)	c
BNEG	0110	Branch on Negative	n
BVS	0111	Branch on oVerflow Set	v
BA	1000	Branch Always	No test
BNE	1001	Branch on Not Equal	not z
BG	1010	Branch on Greater	not(z OR (n XOR v))
BGE	1011	Branch on Greater or Equal	not(n XOR v)
BGU	1100	Branch on Greater, Unsigned	not(c OR z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not c
BPOS	1110	Branch on Positive	not n
BVC	1111	Branch on oVerflow Clear	not v

Format:



CALL — Call — CALL

Operation:

r[15] ← PC
PC ← nPC
nPC ← PC + (disp30 x 4)

Assembler Syntax: call *label*

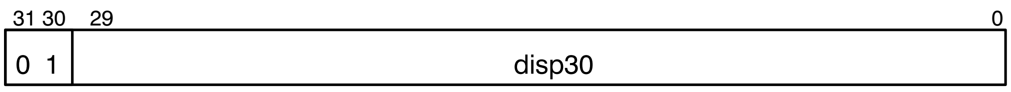
Description: The CALL instruction causes a delayed, unconditional, PC-relative control transfer to the address (PC + 4) + (disp30 x 4). The CALL instruction does not have an annul bit, therefore the delay slot instruction following the CALL instruction is always executed (See Section NO TAG). CALL first writes its return address (PC) into the *outs* register, r[15], and then adds 4 to the PC. The 32-bit displacement which is added to the new PC is formed by appending two low-order zeros to the 30-bit word displacement contained in the instruction. Consequently, the target address can be anywhere in the CY7C601's user or supervisor address space.

If the instruction following a CALL uses register r[15] as a source operand, hardware interlocks add a one cycle delay.

Programming note: a register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15.

Traps: none

Format:



CBccc Coprocessor Conditional Branch CBccc

Operation:

```
PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
          else nPC ← nPC + 4
```

Assembler Syntax:

cba{,a}	*label*
cbn{,a}	*label*
cb3{,a}	*label*
cb2{,a}	*label*
cb23{,a}	*label*
cb1{,a}	*label*
cb13{,a}	*label*
cb12{,a}	*label*
cb123{,a}	*label*
cb0{,a}	*label*
cb03{,a}	*label*
cb02{,a}	*label*
cb023{,a}	*label*
cb01{,a}	*label*
cb013{,a}	*label*
cb012{,a}	*label*

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The CBccc instructions (except for CBA and CBN) evaluate specific coprocessor condition code combinations (from the CCC<1:0> inputs) based on the branch type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See Section NO TAG regarding control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section NO TAG regarding delayed branching.

Branch Never (CBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (CBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a=1$, the delay instruction is annulled; if $a=0$, the delay instruction is executed.

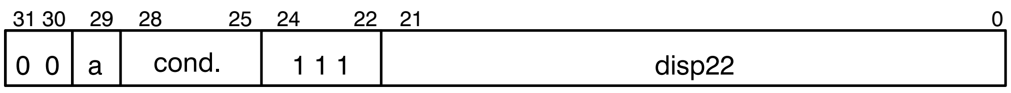
To prevent misapplication of the condition codes, a non-coprocessor instruction must immediately precede a CBccc instruction.

A CBccc instruction generates a cp_disabled trap (and does not branch or annul) if the PSR's EC bit is reset or if no coprocessor is present.

Traps: cp_disabled
cp_exception

Mnemonic	cond.	CCC<1:0> test
CBN	0000	Never
CB123	0001	1 or 2 or 3
CB12	0010	1 or 2
CB13	0011	1 or 3
CB1	0100	1
CB23	0101	2 or 3
CB2	0110	2
CB3	0111	3
CBA	1000	Always
CB0	1001	0
CB03	1010	0 or 3
CB02	1011	0 or 2
CB023	1100	0 or 2 or 3
CB01	1101	0 or 1
CB013	1110	0 or 1 or 3
CB012	1111	0 or 1 or 2

Format:



CPop Coprocessor Operate CPop

Operation: Dependent on Coprocessor implementation

Assembler Syntax: Unspecified

Description: CPop1 and CPop2 are the instruction formats for coprocessor operate instructions. The *op3* field for CPop1 is 110110; for CPop2 it's 110111. The coprocessor operations themselves are encoded in the *opc* field and are dependent on the coprocessor implementation. Note that this does not include load/ store coprocessor instructions, which fall into the integer unit's load/store instruction category.

All CPop instructions take all operands from, and return all results to, the coprocessor's registers. The data types supported, how the operands are aligned, and whether a CPop generates a cp_exception trap are Coprocessor dependent.

A CPop instruction causes a cp_disabled trap if the PSR's EC bit is reset or if no coprocessor is present.

Traps: cp_disabled
cp_exception

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 1 0	rs1	opc	rs2

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 1 1	rs1	opc	rs2

FABSs — Absolute Value Single — FABSs

(FPU Instruction Only)

Operation: f[rd]s ← f[rs2]s AND 7FFFFFFF H

Assembler Syntax: fabss *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FABSs instruction clears the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since rs2 can be either an even or odd register, FABSs can also operate on the high-order words of double and extended operands, which accomplishes sign bit clear for these data types.

Traps: fp_disabled
fp_exception*

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 1 0 0 1	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FADDd **Add Double** FADDd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs1]d + f[rs2]d

Assembler Syntax: faddd *$freg_{rs1}$*, *$freg_{rs2}$*, *$freg_{rd}$*

Description: The FADDd instruction adds the contents of f[rs1] CONCAT f[rs1+1] to the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 0	rs2

FADDs Add Single FADDs

(FPU Instruction Only)

Operation: f[rd]s $\leftarrow$ f[rs1]s + f[rs2]s

Assembler Syntax: fadds *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FADDs instruction adds the contents of f[rs1] to the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 0 1	rs2

FADDx Add Extended FADDx

(FPU Instruction Only)

Operation: f[rd]x ← f[rs1]x + f[rs2]x

Assembler Syntax: faddx *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FADDx instruction adds the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] to the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 0 1 1	rs2

FBfcc Floating-Point Conditional Branch FBfcc

Operation:

```
PC ← nPC
If condition true then nPC ← PC + (sign extnd(disp22) x 4)
          else nPC ← nPC + 4
```

Assembler Syntax:

fba{,a}	*label*	
fbn{,a}	*label*	
fbu{,a}	*label*	
fbg{,a}	*label*	
fbug{,a}	*label*	
fbl{,a}	*label*	
fbul{,a}	*label*	
fblg{,a}	*label*	
fbne{,a}	*label*	synonym: fbnz
fbe{,a}	*label*	synonym: fbz
fbue{,a}	*label*	
fbge{,a}	*label*	
fbuge{,a}	*label*	
fble{,a}	*label*	
fbule{,a}	*label*	
fbo{,a}	*label*	

Note: The instruction's annul bit field, *a*, is set by appending ",a" after the branch name. If it is not appended, the *a* field is automatically reset. ",a" is shown in braces because it is optional.

Description: The FBfcc instructions (except for FBA and FBN) evaluate specific floating-point condition code combinations (from the FCC<1:0> inputs) based on the branch type, as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, the branch is taken, causing a delayed, PC-relative control transfer to the address (PC + 4) + (sign extnd(disp22) x 4). If the condition codes evaluate as false, the branch is not taken. See Section NO TAG for additional information on control transfer instructions.

If the branch is not taken, the annul bit field (*a*) is checked. If *a* is set, the instruction immediately following the branch instruction (the delay instruction) *is not* executed (i.e., it is annulled). If the annul field is zero, the delay instruction *is* executed. If the branch is taken, the annul field is ignored, and the delay instruction is executed. See Section NO TAG regarding delayed branch instructions.

Branch Never (FBN) executes like a NOP, except it obeys the annul field with respect to its delay instruction.

Branch Always (FBA), because it always branches regardless of the condition codes, would normally ignore the annul field. Instead, it follows the same annul field rules: if $a=1$, the delay instruction is annulled; if $a=0$, the delay instruction is executed.

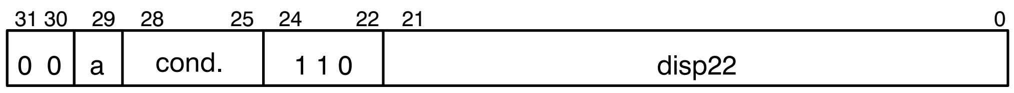
To prevent misapplication of the condition codes, a non−floating-point instruction must immediately precede an FBfcc instruction.

An FBfcc instruction generates an fp_disabled trap (and does not branch or annul) if the PSR's EF bit is reset or if no Floating-Point Unit is present.

Traps: fp_disabled
fp_exception*

Mnemonic	Cond.	Operation	fcc Test
FBN	0000	Branch Never	no test
FBNE	0001	Branch on Not Equal	U or L or G
FBLG	0010	Branch on Less or Greater	L or G
FBUL	0011	Branch on Unordered or Less	U or L
FBL	0100	Branch on Less	L
FBUG	0101	Branch on Unordered or Greater	U or G
FBG	0110	Branch on Greater	G
FBU	0111	Branch on Unordered	U
FBA	1000	Branch Always	no test
FBE	1001	Branch on Equal	E
FBUE	1010	Branch on Unordered or Equal	U or E
FBGE	1011	Branch on Greater or Equal	G or E
FBUGE	1100	Branch on Unordered or Greater or Equal	U or G or E
FBLE	1101	Branch on Less or Equal	L or E
FBULE	1110	Branch on Unordered or Less or Equal	U or L or E
FBO	1111	Branch on Ordered	L or G or E

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FCMPd Compare Double FCMPd

(FPU Instruction Only)

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmpd *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non−floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPd causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 0	rs2

FCMPEd Compare Double and Exception if Unordered FCMPEd

(FPU Instruction Only)

Operation: fcc ← f[rs1]d COMPARE f[rs2]d

Assembler Syntax: fcmped *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPEd subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1], f[rs1+1] and fs2 represents the contents of f[rs2], f[rs2+1].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non−floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEd causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 0	rs2

FCMPEs Compare Single and Exception if Unordered FCMPEs

(FPU Instruction Only)

Operation: fcc $\leftarrow$ f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmpes *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPEs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non−floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEs causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 0 1	rs2

FCMPEx Compare Extended and Exception if Unordered FCMPE x

(FPU Instruction Only)

Operation: fcc ← f[rs1]x COMPARE f[rs2]x

Assembler Syntax: fcmpex *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPEx subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPEx causes an invalid exception (nv) if either operand is a signaling or quiet NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 1 1 1	rs2

FCMPs — Compare Single — FCMPs

(FPU Instruction Only)

Operation: fcc $\leftarrow$ f[rs1]s COMPARE f[rs2]s

Assembler Syntax: fcmps *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPs subtracts the contents of f[rs2] from the contents of f[rs1] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1] and fs2 represents the contents of f[rs2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non-floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPs causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 0 1	rs2

FCMPx Compare Extended FCMPx

(FPU Instruction Only)

Operation: fcc ⟵ f[rs1]x COMPARE f[rs2]x

Assembler Syntax: fcmpx *freg*$_{rs1}$, *freg*$_{rs2}$

Description: FCMPx subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] following the ANSI/IEEE 754-1985 standard. The result is evaluated, the FSR's *fcc* bits are set accordingly, and then the result is discarded. The codes are set as follows:

fcc	Relation
0	fs1 = fs2
1	fs1 < fs2
2	fs1 > fs2
3	fs1 ? fs2 (unordered)

In this table, fs1 stands for the contents of f[rs1], f[rs1+1], f[rs1+2] and fs2 represents the contents of f[rs2], f[rs2+1], f[rs2+2].

Compare instructions are used to set up the floating-point condition codes for a subsequent FBfcc instruction. However, to prevent misapplication of the condition codes, at least one non−floating-point instruction must be executed between an FCMP and a subsequent FBfcc instruction.

FCMPx causes an invalid exception (nv) if either operand is a signaling NaN.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	ignored	1 1 0 1 0 1	rs1	0 0 1 0 1 0 0 1 1	rs2

FDIVd — Divide Double — FDIVd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs1]d / f[rs2]d

Assembler Syntax: fdivd *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FDIVd instruction divides the contents of f[rs1] CONCAT f[rs1+1] by the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 1 0	rs2

FDIVs — Divide Single — FDIVs

(FPU Instruction Only)

Operation: f[rd]s $\leftarrow$ f[rs1]s / f[rs2]s

Assembler Syntax: fdivs *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FDIVs instruction divides the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 1 0 1	rs2

FDIVx Divide Extended FDIVx

(FPU Instruction Only)

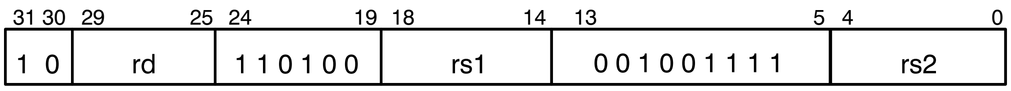
Operation: f[rd]x ◄— f[rs1]x / f[rs2]x

Assembler Syntax: fdivx *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FDIVx instruction divides the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception (of, uf, dz, nv, nx)

Format:



FdTOi Convert Double to Integer FdTOi

(FPU Instruction Only)

Operation: f[rd]i ← f[rs2]d

Assembler Syntax: fdtoi *freg*$_{rs2}$, *freg*$_{rd}$

Description: FdTOi converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding direction field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 0	rs2

FdTOs — Convert Double to Single — FdTOs

(FPU Instruction Only)

Operation: f[rd]s $\leftarrow$ f[rs2]d

Assembler Syntax: fdtos *freg*$_{rs2}$, *freg*$_{rd}$

Description: FdTOs converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 1 0	rs2

FdTOx Convert Double to Extended FdTOx

(FPU Instruction Only)

Operation: f[rd]x ← f[rs2]d

Assembler Syntax: fdtox *freg*$_{rs2}$, *freg*$_{rd}$

Description: FdTOx converts the floating-point double contents of f[rs2] CONCAT f[rs2+1] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], and f[rd+2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 1 0	rs2

FiTOd Convert Integer to Double FiTOd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs2]i

Assembler Syntax: fitod *freg*$_{rs2}$, *freg*$_{rd}$

Description: FiTOd converts the 32-bit, signed integer contents of f[rs2] to a floating-point, double-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception*

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 0	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FiTOs Convert Integer to Single FiTOs

(FPU Instruction Only)

Operation: f[rd]s ← f[rs2]i

Assembler Syntax: fitos *freg*$_{rs2}$, *freg*$_{rd}$

Description: FiTOs converts the 32-bit, signed integer contents of f[rs2] to a floating-point, single-precision format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field, *RD*.

Traps: fp_disabled
fp_exception (nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 0 0	rs2

FiTOx Convert Integer to Extended FiTOx

(FPU Instruction Only)

Operation: f[rd]x ← f[rs2]i

Assembler Syntax: fitox *freg*$_{rs2}$, *freg*$_{rd}$

Description: FiTOx converts the 32-bit, signed integer contents of f[rs2] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception*

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 0 0	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FMOVs — Move — FMOVs

(FPU Instruction Only)

Operation: f[rd]s ← f[rs2]s

Assembler Syntax: fmovs *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FMOVs instruction moves the word content of register f[rs2] to the register f[rd]. Multiple FMOVs's are required to transfer multiple-precision numbers between *f* registers.

Traps: fp_disabled
fp_exception*

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 0 0 0 1	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FMULd Multiply Double FMULd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs1]d x f[rs2]d

Assembler Syntax: fmuld *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FMULd instruction multiplies the contents of f[rs1] CONCAT f[rs1+1] by the contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 0	rs2

FMULs Multiply Single FMULs

(FPU Instruction Only)

Operation: f[rd]s $\leftarrow$ f[rs1]s x ([rs2]s

Assembler Syntax: fmuls *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FMULs instruction multiplies the contents of f[rs1] by the contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 0 1	rs2

FMULx Multiply Extended FMULx

(FPU Instruction Only)

Operation: f[rd]x $\leftarrow$ f[rs1]x x f[rs2]x

Assembler Syntax: fmulx *freg$_{rs1}$*, *freg$_{rs2}$*, *freg$_{rd}$*

Description: The FMULx instruction multiplies the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] by the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 1 0 1 1	rs2

FNEGs Negate FNEGs

(FPU Instruction Only)

Operation: f[rd]s $\leftarrow$ f[rs2]s XOR 80000000 H

Assembler Syntax: fnegs *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FNEGs instruction complements the sign bit of the word in f[rs2] and places the result in f[rd]. It does not round.

Since this FPop can address both even and odd *f* registers, FNEGs can also operate on the high-order words of double and extended operands, which accomplishes sign bit negation for these data types.

Traps: fp_disabled
fp_exception*

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 0 0 0 1 0 1	rs2

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

FSQRTd Square Root Double FSQRTd

(FPU Instruction Only)

Operation: f[rd]d $\leftarrow$ SQRT f[rs2]d

Assembler Syntax: fsqrtd *freg*$_{rs2}$, *freg*$_{rd}$

Description: FSQRTd generates the square root of the floating-point double contents of f[rs2] CONCAT f[rs2+1] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 1 0 1 0 1 0	rs2

FSQRTs — Square Root Single — FSQRTs

(FPU Instruction Only)

Operation: f[rd]s ← SQRT f[rs2]s

Assembler Syntax: fsqrts *freg*$_{rs2}$, *freg*$_{rd}$

Description: FSQRTs generates the square root of the floating-point single contents of f[rs2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 1 0 1 0 0 1	rs2

FSQRTx Square Root Extended FSQRTx

(FPU Instruction Only)

Operation: f[rd]x $\leftarrow$ SQRT f[rs2]x

Assembler Syntax: fsqrtx *freg*$_{rs2}$, *freg*$_{rd}$

Description: FSQRTx generates the square root of the floating-point extended contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], and f[rd+2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 0 0 1 0 1 0 1 1	rs2

FsTOd Convert Single to Double FsTOd

(FPU Instruction Only)

Operation: f[rd]d ◄— f[rs2]s

Assembler Syntax: fstod *freg*$_{rs2}$, *freg*$_{rd}$

Description: FsTOd converts the floating-point single contents of f[rs2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction field (*RD*) of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 0 1	rs2

FsTOi Convert Single to Integer FsTOi

(FPU Instruction Only)

Operation: f[rd]i ← f[rs2]s

Assembler Syntax: fstoi *freg*$_{rs2}$, *freg*$_{rd}$

Description: FsTOi converts the floating-point single contents of f[rs2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 0 1	rs2

FsTOx Convert Single to Extended FsTOx

(FPU Instruction Only)

Operation: f[rd]x ← f[rs2]s

Assembler Syntax: fstox *freg*$_{rs2}$, *freg*$_{rd}$

Description: FsTOx converts the floating-point single contents of f[rs2] to an extended-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd], f[rd+1], and f[rd+2]. Rounding is performed according to the rounding direction (*RD*) and rounding precision (*RP*) fields of the FSR.

Traps: fp_disabled
fp_exception (nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 1 0 1	rs2

FSUBd Subtract Double FSUBd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs1]d – f[rs2]d

Assembler Syntax: fsubd *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FSUBd instruction subtracts the contents of f[rs2] CONCAT f[rs2+1] from the contents of f[rs1] CONCAT f[rs1+1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd] and f[rd+1].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 0	rs2

FSUBs Subtract Single FSUBs

(FPU Instruction Only)

Operation: f[rd]s ← f[rs1]s – f[rs2]s

Assembler Syntax: fsubs *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FSUBs instruction subtracts the contents of f[rs2] from the contents of f[rs1] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd].

Traps: fp_disabled
fp_exception (of, uf, nx, nv)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 0 1	rs2

FSUBx — Subtract Extended — FSUBx

(FPU Instruction Only)

Operation: f[rd]x $\leftarrow$ f[rs1]x – f[rs2]x

Assembler Syntax: fsubx *freg*$_{rs1}$, *freg*$_{rs2}$, *freg*$_{rd}$

Description: The FSUBx instruction subtracts the contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] from the contents of f[rs1] CONCAT f[rs1+1] CONCAT f[rs1+2] as specified by the ANSI/IEEE 754-1985 standard and places the results in f[rd], f[rd+1], and f[rd+2].

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	rs1	0 0 1 0 0 0 1 1 1	rs2

FxTOd Convert Extended to Double FxTOd

(FPU Instruction Only)

Operation: f[rd]d ← f[rs2]x

Assembler Syntax: fxtod *freg*$_{rs2}$, *freg*$_{rd}$

Description: FxTOd converts the floating-point extended contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] to a double-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd] and f[rd+1]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 1 0 1 1	rs2

FxTOi Convert Extended to Integer FxTOi

(FPU Instruction Only)

Operation: f[rd]i ← f[rs2]x

Assembler Syntax: fxtoi *freg*$_{rs2}$, *freg*$_{rd}$

Description: FxTOi converts the floating-point extended contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] to a 32-bit, signed integer by rounding toward zero as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. The rounding field (*RD*) of the FSR is ignored.

Traps: fp_disabled
fp_exception (nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 1 0 0 1 1	rs2

FxTOs Convert Extended to Single FxTOs

(FPU Instruction Only)

Operation: f[rd]s ⟵ f[rs2]x

Assembler Syntax: fxtos *freg*$_{rs2}$, *freg*$_{rd}$

Description: FxTOs converts the floating-point extended contents of f[rs2] CONCAT f[rs2+1] CONCAT f[rs2+2] to a single-precision, floating-point format as specified by the ANSI/IEEE 754-1985 standard. The result is placed in f[rd]. Rounding is performed according to the rounding direction (*RD*) field of the FSR.

Traps: fp_disabled
fp_exception (of, uf, nv, nx)

Format:

31 30	29 25	24 19	18 14	13 5	4 0
1 0	rd	1 1 0 1 0 0	ignored	0 1 1 0 0 0 1 1 1	rs2

IFLUSH Instruction Cache Flush IFLUSH

Operation: FLUSH $\leftarrow$ [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: iflush *address*

Description: The IFLUSH instruction causes a word to be flushed from an instruction cache which may be internal to the processor. The word to be flushed is at the address specified by the contents of r[rs1] plus either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

Since there is no internal instruction cache in the current CY7C600 family, the result of executing an IFLUSH instruction is dependent on the state of the input signal, Instruction Cache Flush Trap ($\overline{\text{IFT}}$). If $\overline{\text{IFT}}$ = 1, IFLUSH executes as a NOP, with no side effects. If $\overline{\text{IFT}}$ = 0, execution of IFLUSH causes an illegal_instruction trap.

Traps: illegal_instruction

Format:

31	30	29 25	24 19	18 14	13	12 5	4 0
1	0	ignored	1 1 1 0 1 1	rs1	i=0	ignored	rs2

31	30	29 25	24 19	18 14	13	12 0
1	0	ignored	1 1 1 0 1 1	rs1	i=1	simm13

JMPL Jump and Link JMPL

Operation:
r[rd] ← PC
PC ← nPC
nPC ← r[rs1] + (r[rs2] or sign extnd(simm13))

Assembler Syntax: jmpl *address*, *reg*$_{rd}$

Description: JMPL first provides linkage by saving its return address into the register specified in the *rd* field. It then causes a register-indirect, delayed control transfer to an address specified by the sum of the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If either of the low-order two bits of the jump address is nonzero, a memory_address_not_aligned trap is generated.

Programming note: A register-indirect CALL can be constructed using a JMPL instruction with *rd* set to 15. JMPL can also be used to return from a CALL. In this case, *rd* is set to 0 and the return (jump) address would be equal to r[31] + 8.

Traps: memory_address_not_aligned

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 1 1 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	1 1 1 0 0 0	rs1	i=1	simm13

LD Load Word LD

Operation: r[rd] ⟵ [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *reg*$_{rd}$

Description: The LD instruction moves a word from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

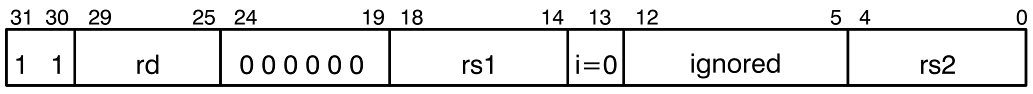
If LD takes a trap, the contents of the destination register remain unchanged.

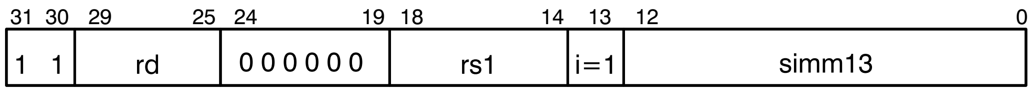
If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





LDA — Load Word from Alternate space — LDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]

Assembler Syntax: lda [*regaddr*] *asi*, *reg*$_{rd}$

Description: The LDA instruction moves a word from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If LDA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 0 0 0	rs1	i=0	asi	rs2

LDC Load Coprocessor register LDC

Operation: c[rd] ⟵ [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *creg*$_{rd}$

Description: The LDC instruction moves a word from memory into a coprocessor register, c[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

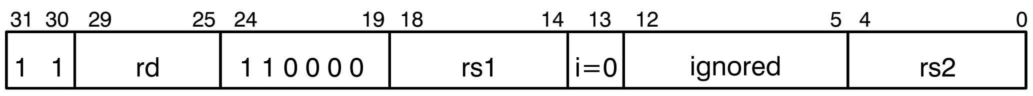
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDC takes a trap, the state of the coprocessor depends on the particular implementation.

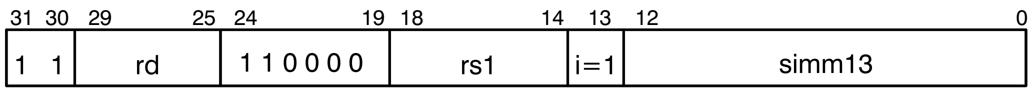
If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





LDCSR Load Coprocessor State Register LDCSR

Operation: CSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %csr

Description: The LDCSR instruction moves a word from memory into the Coprocessor State Register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

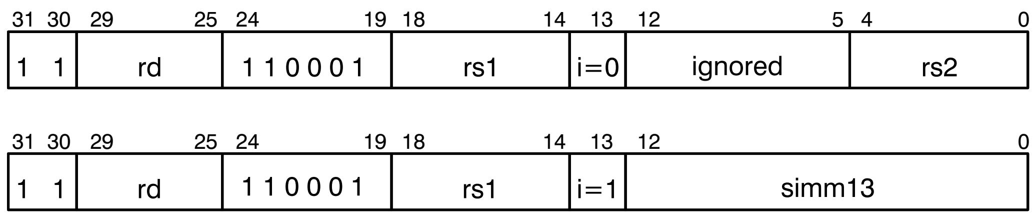
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDCSR takes a trap, the state of the coprocessor depends on the particular implementation.

If the instruction following a LDCSR uses the CSR as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon implementation of the coprocessor.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



LDD Load Doubleword LDD

Operation: r[rd] ← [r[rs1] + (r[rs2] or sign extnd(simm13))]
r[rd + 1] ← [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], *reg*$_{rd}$

Description: The LDD instruction moves a doubleword from memory into a destination register pair, r[rd] and r[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section NO TAG).

If a data_access_exception trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDD, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 0 0 0 1 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	0 0 0 0 1 1	rs1	i=1	simm13

LDDA Load Doubleword from Alternate space LDDA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← [r[rs1] + r[rs2]]
r[rd +1] ← [r[rs1] + r[rs2] + 4]

Assembler Syntax: ldda [*regaddr*] *asi, reg*$_{rd}$

Description: The LDDA instruction moves a doubleword from memory into the destination registers, r[rd] and r[rd+1]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section NO TAG).

If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDA, this applies to both destination registers.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 0 1 1	rs1	i=0	asi	rs2

LDDC Load Doubleword Coprocessor LDDC

Operation: c[rd] ⟵ [r[rs1] + (r[rs2] or sign extnd(simm13))]
c[rd + 1] ⟵ [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], $creg_{rd}$

Description: The LDDC instruction moves a doubleword from memory into the coprocessor registers, c[rd] and c[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section NO TAG).

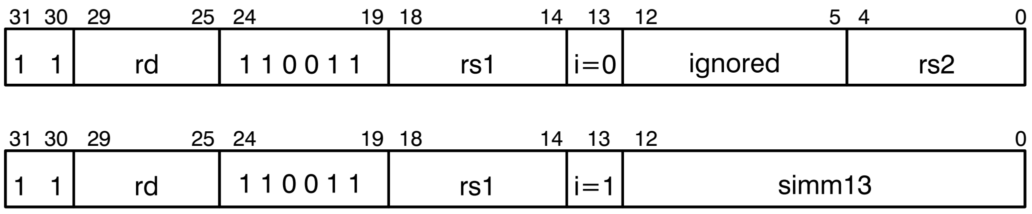
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If LDDC takes a trap, the state of the coprocessor depends on the particular implementation.

If the instruction following a coprocessor load uses the load's c[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem and coprocessor implementation. For an LDDC, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:



LDDF Load Doubleword Floating-Point LDDF

Operation: f[rd] ⟵ [r[rs1] + (r[rs2] or sign extnd(simm13))]
f[rd + 1] ⟵ [(r[rs1] + (r[rs2] or sign extnd(simm13))) + 4]

Assembler Syntax: ldd [*address*], *freg*$_{rd}$

Description: The LDDF instruction moves a doubleword from memory into the floating-point registers, f[rd] and f[rd+1]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant memory word is always moved into the even-numbered destination register and the least significant memory word is always moved into the next odd-numbered register (see discussion in Section NO TAG).

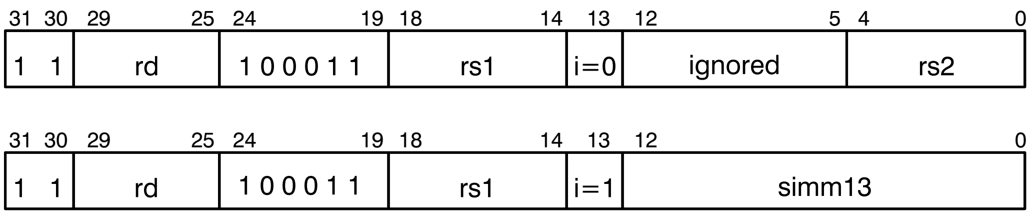
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place during the effective address memory access, the destination registers remain unchanged.

If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem. For an LDDF, this applies to both destination registers.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:



* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDF Load Floating-Point register LDF

Operation: f[rd] ⟵ [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], *freg*$_{rd}$

Description: The LDF instruction moves a word from memory into a floating-point register, f[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EF bit is set to zero or if no Floating-Point Unit is present, an fp_disabled trap will be generated. If LDF takes a trap, the contents of the destination register remain unchanged.

If the instruction following a floating-point load uses the load's f[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

31	30	29 25	24 19	18 14	13	12 5	4 0
1	1	rd	1 0 0 0 0 0	rs1	i=0	ignored	rs2

31	30	29 25	24 19	18 14	13	12 0
1	1	rd	1 0 0 0 0 0	rs1	i=1	simm13

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDFSR **Load Floating-Point State Register** LDFSR

Operation: FSR ← [r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ld [*address*], %fsr

Description: The LDFSR instruction moves a word from memory into the floating-point state register. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it loads the memory word into the FSR.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If LDFSR takes a trap, the contents of the FSR remain unchanged.

If the instruction following a LDFSR uses the FSR as a source operand, hardware interlocks add one or more cycle delay to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	1 0 0 0 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	1 0 0 0 0 1	rs1	i=1	simm13

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

LDSB Load Signed Byte LDSB

Operation: r[rd] ⟵ sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsb [*address*], *reg*$_{rd}$

Description: The LDSB instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and sign-extended in r[rd].

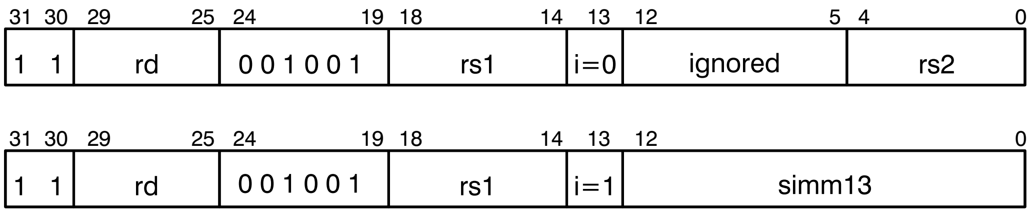
If LDSB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles to the following instruction depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:



LDSBA — Load Signed Byte from Alternate space — LDSBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsba [*regaddr*] *asi*, *reg*$_{rd}$

Description: The LDSBA instruction moves a signed byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and sign-extended in r[rd].

If LDSBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 1 0 0 1	rs1	i=0	asi	rs2

LDSH — Load Signed Halfword — LDSH

Operation: r[rd] ⟵ sign extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldsh [*address*], *reg*$_{rd}$

Description: The LDSH instruction moves a signed halfword from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched halfword is right-justified and sign-extended in r[rd].

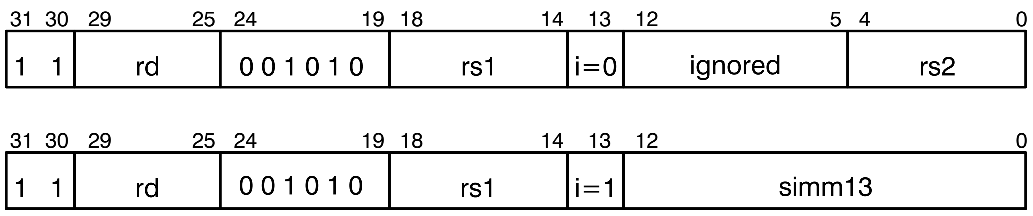
If LDSH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



LDSHA — Load Signed Halfword from Alternate space — LDSHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← sign extnd[r[rs1] + r[rs2]]

Assembler Syntax: ldsha [reg_{addr}] *asi*, reg_{rd}

Description: The LDSHA instruction moves a signed halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and sign-extended in r[rd].

If LDSHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 1 0 1 0	rs1	i=0	asi	rs2

LDSTUB Atomic Load/Store Unsigned Byte LDSTUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← FFFFFFFF H

Assembler Syntax: ldstub [*address*], *reg*$_{rd}$

Description: The LDSTUB instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them serially, in some order.

The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:

31	30	29 25	24 19	18 14	13	12 5	4 0
1	1	rd	0 0 1 1 0 1	rs1	i=0	ignored	rs2

31	30	29 25	24 19	18 14	13	12 0
1	1	rd	0 0 1 1 0 1	rs1	i=1	simm13

LDSTUBA Atomic Load/Store Unsigned Byte in Alternate space (Privileged Instruction) LDSTUBA

Operation:

address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]
[r[rs1] + r[rs2]] ← FFFFFFFF H

Assembler Syntax:

ldstuba [reg_{addr}] *asi*, reg_{rd}

Description:

The LDSTUBA instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them in some serial order.

The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

If LDSTUBA takes a trap, the contents of the memory address remain unchanged.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 1 1 0 1	rs1	i=0	asi	rs2

LDUB Load Unsigned Byte LDUB

Operation: r[rd] $\leftarrow$ zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: ldub [*address*], *reg*$_{rd}$

Description: The LDUB instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

If LDUB takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 0 0 0 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	0 0 0 0 0 1	rs1	i=1	simm13

LDUBA Load Unsigned Byte from Alternate space LDUBA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduba [*reg$_{addr}$*] *asi*, *reg$_{rd}$*

Description: The LDUBA instruction moves an unsigned byte from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched byte is right-justified and zero-extended in r[rd].

If LDUBA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 0 0 1	rs1	i=0	asi	rs2

LDUH Load Unsigned Halfword LDUH

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]

Assembler Syntax: lduh [*address*], *reg*$_{rd}$

Description: The LDUH instruction moves an unsigned halfword from memory into the destination register, r[rd]. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched halfword is right-justified and zero-extended in r[rd].

If LDUH takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 0 0 0 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	0 0 0 0 1 0	rs1	i=1	simm13

LDUHA Load Unsigned Halfword from Alternate space LDUHA

(Privileged Instruction)

Operation: address space ← asi
r[rd] ← zero extnd[r[rs1] + r[rs2]]

Assembler Syntax: lduha [*regaddr*] *asi*, reg_{rd}

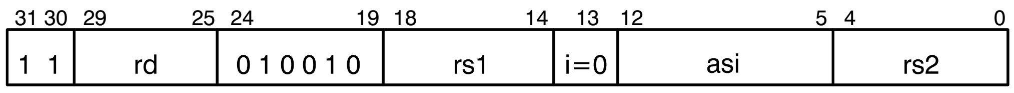
Description: The LDUHA instruction moves an unsigned halfword from memory into the destination register, r[rd]. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The fetched halfword is right-justified and zero-extended in r[rd].

If LDUHA takes a trap, the contents of the destination register remain unchanged.

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:



MULScc Multiply Step and modify icc MULScc

Operation:

```
op1 = (n XOR v) CONCAT r[rs1]<31:1>
if (Y<0> = 0) op2 = 0, else op2 = r[rs2] or sign extnd(simm13)
r[rd] ← op1 + op2
Y ← r[rs1]<0> CONCAT Y<31:1>
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← ((op1<31> AND op2<31> AND not r[rd]<31>)
        OR (not op1<31> AND not op2<31> AND r[rd]<31>))
c ← ((op1<31> AND op2<31>)
        OR (not r[rd] AND (op1<31> OR op2<31>))
```

Assembler Syntax: mulscc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: The multiply step instruction can be used to generate the 64-bit product of two signed or unsigned words. MULScc works as follows:

1. The “incoming partial product” in r[rs1] is shifted right by one bit and the high-order bit is replaced by the sign of the previous partial product (n XOR v). This is operand1.
2. If the least significant bit of the multiplier in the Y register equals zero, then operand2 is set to zero. If the LSB of the Y register equal one, then operand2 becomes the multiplicand, which is either the contents of r[rs2] if the instruction *i* field is zero, or sign extnd(simm13) if the *i* field is one. Operand2 is then added to operand1 and stored in r[rd] (the outgoing partial product).
3. The multiplier in the Y register is then shifted right by one bit and its high-order bit is replaced by the least significant bit of the incoming partial product in r[rs1].
4. The PSR’s integer condition codes are updated according to the addition performed in step 2.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	1 0 0 1 0 0	rs1	i=1	simm13

OR　　Inclusive-Or　　OR

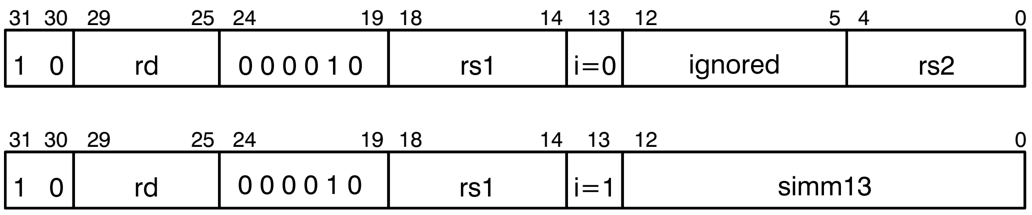
Operation: r[rd] ◄— r[rs1] OR (r[rs2] or sign extnd(simm13))

Assembler Syntax: or *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:



ORcc Inclusive-Or and modify icc ORcc

Operation:

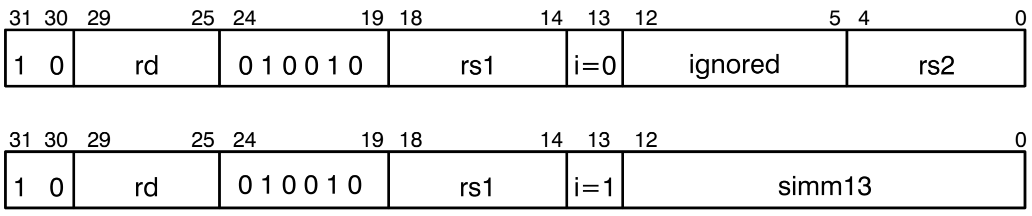
r[rd] ← r[rs1] OR (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: orcc *reg$_{rs1}$, reg_or_imm, reg$_{rd}$*

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:



ORN — Inclusive-Or Not — ORN

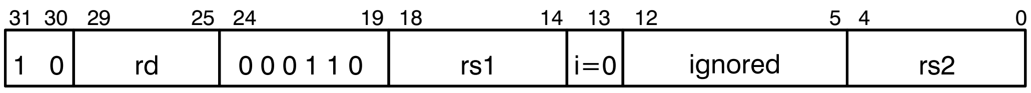
Operation: r[rd] ⟵ r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))

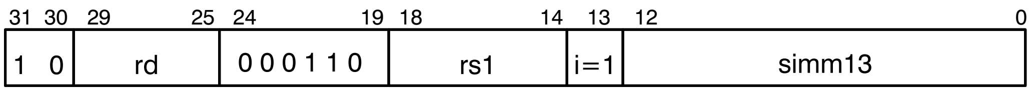
Assembler Syntax: orn *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





ORNcc Inclusive-Or Not and modify icc ORNcc

Operation:
r[rd] ← r[rs1] OR not(operand2), where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if [r[rd]]=0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: orncc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical OR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. ORNcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 1 0 1 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 1 0 1 1 0	rs1	i=1	simm13

RDPSR — Read Processor State Register — RDPSR

(Privileged Instruction)

Operation: r[rd] ⟵ PSR

Assembler Syntax: rd %psr, *reg*$_{rd}$

Description: RDPSR copies the contents of the PSR into the register specified by the *rd* field.

Traps: privileged-instruction (if S=0)

Format:

31 30	29 25	24 19	18 0
1 0	rd	1 0 1 0 0 1	ignored

RDTBR Read Trap Base Register RDTBR

(Privileged Instruction)

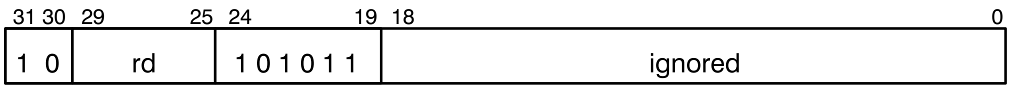
Operation: r[rd] $\leftarrow$ TBR

Assembler Syntax: rd %tbr, reg_{rd}

Description: RDTBR copies the contents of the TBR into the register specified by the *rd* field.

Traps: privileged_instruction (if S=0)

Format:



RDWIM Read Window Invalid Mask register RDWIM

(Privileged Instruction)

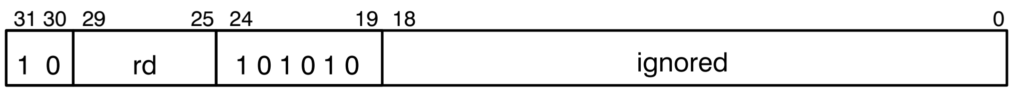
Operation: r[rd] $\leftarrow$ WIM

Assembler Syntax: rd %wim, *reg*$_{rd}$

Description: RDWIM copies the contents of the WIM register into the register specified by the *rd* field.

Traps: privileged_instruction (if S=0)

Format:



RDY Read Y register RDY

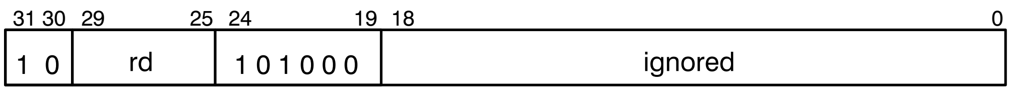
Operation: r[rd] $\leftarrow$ Y

Assembler Syntax: rd %y, *reg*$_{rd}$

Description: RDY copies the contents of the Y register into the register specified by the *rd* field.

Traps: none

Format:



RESTORE — Restore caller's window — RESTORE

Operation:

ncwp $\leftarrow$ CWP + 1
result $\leftarrow$ r[rs1] + (r[rs2] or sign extnd(simm13))
CWP $\leftarrow$ ncwp
r[rd] $\leftarrow$ result
RESTORE does not affect condition codes

Assembler Syntax: restore *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: RESTORE adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND 2^{ncwp} = 1), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit *i* = 1) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit *i* = 0). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the called window).

The new CWP value is written into the PSR, causing the previous window (the caller's window) to become the active window. The result of the addition is now written into the r[rd] register of the restored window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps: window_underflow

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 1 1 1 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	1 1 1 1 0 1	rs1	i=1	simm13

RETT Return from Trap RETT

(Privileged Instruction)

Operation:

$ncwp \leftarrow CWP + 1$
$ET \leftarrow 1$
$PC \leftarrow nPC$
$nPC \leftarrow r[rs1] + (r[rs2]$ or sign extnd$(simm13))$
$CWP \leftarrow ncwp$
$S \leftarrow pS$

Assembler Syntax: rett *address*

Description: RETT adds one to the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND $2^{ncwp} = 1$), a window_underflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then RETT causes a delayed control transfer to the address derived by adding the contents of r[rs1] to either the contents of r[rs2] (field bit *i* = 1) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit *i* = 0).

Before the control transfer takes place, the new CWP value is written into the PSR, causing the previous window (the one in which the trap was taken) to become the active window. In addition, the PSR's ET bit is set to one (traps enabled) and the previous Supervisor bit (pS) is restored to the S field.

Although in theory RETT is a delayed control transfer instruction, in practice, RETT must always be immediately preceded by a JMPL instruction, creating a delayed control transfer couple (see Section NO TAG). This has the effect of annulling the delay instruction.

If traps were already enabled before encountering the RETT instruction, an illegal_instruction trap is generated. If traps are not enabled (ET=0) when the RETT is encountered, but (1) the processor is not in supervisor mode (S=0), or (2) the window underflow condition described above occurs, or (3) if either of the two low-order bits of the target address are nonzero, then a reset trap occurs. If a reset trap does occur, the *tt* field of the TBR encodes the trap condition: privileged_instruction, window_underflow, or memory_address_not_aligned.

Programming note: To re-execute the trapping instruction when returning from a trap handler, use the following sequence:

```
jmpl    %17, %0          ! old PC
rett    %18              ! old nPC
```

Note that the CY7C601 saves the PC in r[17] (local 1) and the nPC in r[18] (local2) of the trap window upon entering a trap.

To return to the instruction after the trapping instruction (e.g., when the trapping instruction is emulated), use the sequence:

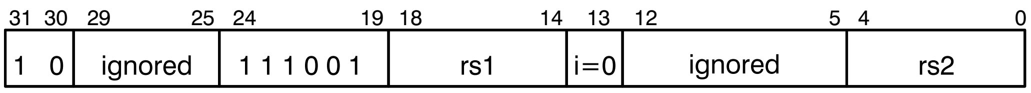
```
jmpl    %18,   %0        ! old nPC
rett    %18 + 4          ! old nPC + 4
```

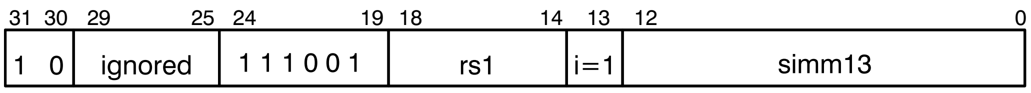
RETT Return from Trap RETT

(Privileged Instruction)

Traps:	illegal_instruction
	reset (privileged_instruction)
	reset (memory_address_not_aligned)
	reset (window_underflow)

Format:





SAVE — Save caller's window — SAVE

Operation:

ncwp ← CWP - 1

result ← r[rs1] + (r[rs2] or sign extnd(simm13))

CWP ← ncwp

r[rd] ← result

SAVE does not affect condition codes

Assembler Syntax: save *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

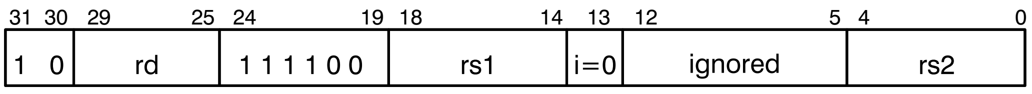
Description: SAVE subtracts one from the Current Window Pointer (modulo the number of implemented windows) and compares this value against the Window Invalid Mask register. If the new window number corresponds to an invalidated window (WIM AND 2^{ncwp} = 1), a window_overflow trap is generated. If the new window number is not invalid (i.e., its corresponding WIM bit is reset), then the contents of r[rs1] is added to either the contents of r[rs2] (field bit *i* = 1) or to the 13-bit, sign-extended immediate value contained in the instruction (field bit *i* = 0). Because the CWP has not been updated yet, r[rs1] and r[rs2] are read from the currently addressed window (the calling window).

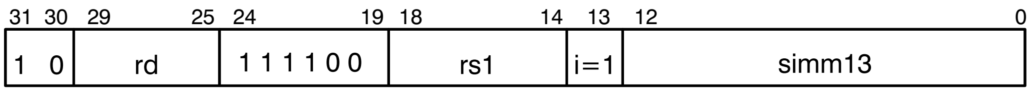
The new CWP value is written into the PSR, causing the active window to become the previous window, and the called window to become the active window. The result of the addition is now written into the r[rd] register of the new window.

Note that arithmetic operations involving the CWP are always done modulo the number of implemented windows (8 for the CY7C601).

Traps: window_overflow

Format:





SETHI **Set High 22 bits of *r register*** SETHI

Operation: r[rd]<31:10> $\leftarrow$ imm22
r[rd]<9:0> $\leftarrow$ 0

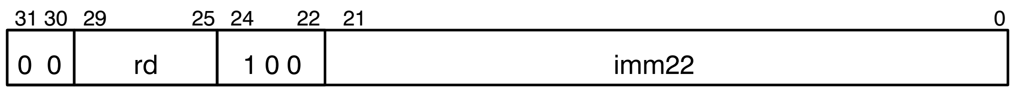
Assembler Syntax: sethi *const22*, reg_{rd}
sethi %hi *value*, reg_{rd}

Description: SETHI zeros the ten least significant bits of the contents of r[rd] and replaces its high-order 22 bits with *imm22*. The condition codes are not affected.

Programming note: SETHI 0, %0 is the preferred instruction to use as a NOP, because it will not increase execution time if it follows a load instruction.

Traps: none

Format:



SLL Shift Left Logical SLL

Operation: r[rd] ⟵ r[rs1] SLL by (r[rs2] or shcnt)

Assembler Syntax: sll *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SLL shifts the contents of r[rs1] left by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SLL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SLL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 0 1	rs1	i=1	ignored	shcnt

SRA Shift Right Arithmetic SRA

Operation: r[rd] ← r[rs1] SRA by (r[rs2] or shcnt)

Assembler Syntax: sra *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SRA shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with the MSB of r[rs1]. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SRA is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRA is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Programming note: A "Shift Left Arithmetic by 1 (and calculate overflow)" can be implemented with an ADDcc instruction.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 1 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 1 1	rs1	i=1	ignored	shcnt

SRL Shift Right Logical SRL

Operation: r[rd] ← r[rs1] SRL by (r[rs2] or shcnt)

Assembler Syntax: srl *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SRL shifts the contents of r[rs1] right by the number of bits specified by the shift count, filling the vacated positions with zeros. The shifted results are written into r[rd]. No shift occurs if the shift count is zero.

If the i bit field equals zero, the shift count for SRL is the least significant five bits of the contents of r[rs2]. If the i bit field equals one, the shift count for SRL is the 13-bit, sign extended immediate value, simm13. In the instruction format and the operation description above, the least significant five bits of simm13 is called *shcnt*.

This instruction does *not* modify the condition codes.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 1 1 0	rs1	i=1	ignored	shcnt

ST **Store Word** ST

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ⟵ r[rd]

Assembler Syntax: st *reg*$_{rd}$, [*address*]

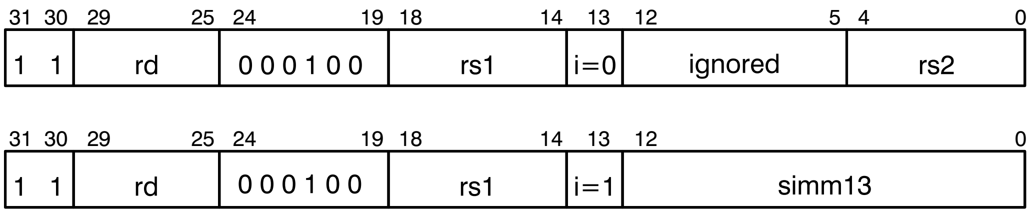
Description: The ST instruction moves a word from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If ST takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STA — Store Word into Alternate space — STA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

Assembler Syntax: sta *reg*$_{rd}$, [*reg addr*] *asi*

Description: The STA instruction moves a word from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 1 0 0	rs1	i=0	asi	rs2

STB Store Byte STB

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] $\leftarrow$ r[rd]

Assembler Syntax: stb *reg*$_{rd}$, [*address*]

synonyms: stub, stsb

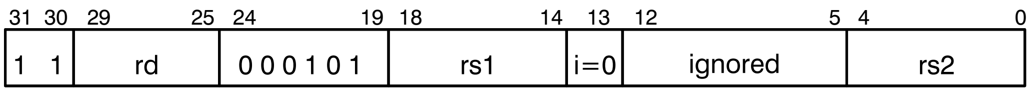
Description: The STB instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

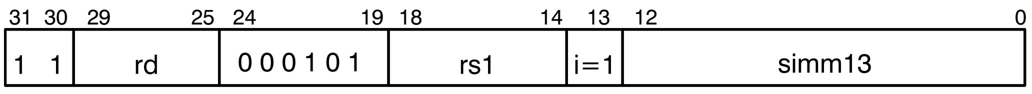
If STB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: data_access_exception

Format:





STBA — Store Byte into Alternate space — STBA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + r[rs2]] ← r[rd]

Assembler Syntax: stba *reg*$_{rd}$, [*reg addr*] *asi*
synonyms: stuba, stsba

Description: The STBA instruction moves the least significant byte from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STBA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 1 0 1	rs1	i=0	asi	rs2

STC Store Coprocessor register STC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] $\leftarrow$ c[rd]

Assembler Syntax: st *creg*$_{rd}$, [*address*]

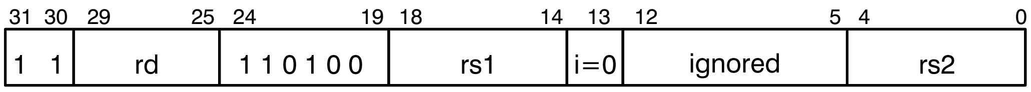
Description: The STC instruction moves a word from a coprocessor register, c[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

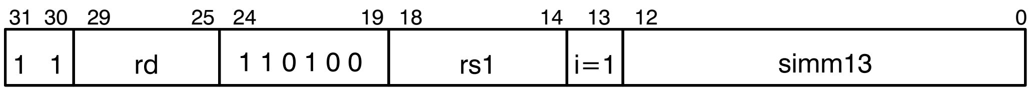
If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STC takes a trap, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:





STCSR **Store Coprocessor State Register** STCSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] $\leftarrow$ CSR

Assembler Syntax: st %csr, [*address*]

Description: The STCSR instruction moves the contents of the Coprocessor State Register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If STCSR takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:

31	30	29 25	24 19	18 14	13	12 5	4 0
1	1	rd	1 1 0 1 0 1	rs1	i=0	ignored	rs2

31	30	29 25	24 19	18 14	13	12 0
1	1	rd	1 1 0 1 0 1	rs1	i=1	simm13

STD Store Doubleword STD

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← r[rd + 1]

Assembler Syntax: std *reg*$_{rd}$, [*address*]

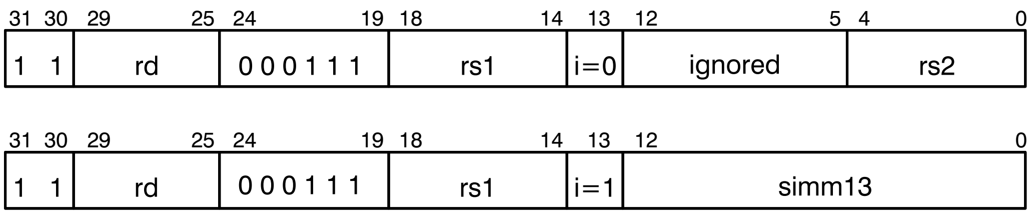
Description: The STD instruction moves a doubleword from the destination register pair, r[rd] and r[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:


31 30 29 25 24 19 18 14 13 12 5 4 0
1 1 | rd | 0 0 0 1 1 1 | rs1 | i=0 | ignored | rs2
31 30 29 25 24 19 18 14 13 12 0
1 1 | rd | 0 0 0 1 1 1 | rs1 | i=1 | simm13

STDA — Store Doubleword into Alternate space — STDA

(Privileged Instruction)

Operation: address space ← asi
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← r[rd + 1]

Assembler Syntax: stda *reg*$_{rd}$, [*reg addr*] *asi*

Description: The STDA instruction moves a doubleword from the destination register pair, r[rd] and r[rd+1], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2]. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 1 1 1	rs1	i=0	asi	rs2

STDC Store Doubleword Coprocessor STDC

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] $\leftarrow$ c[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] $\leftarrow$ c[rd + 1]

Assembler Syntax: std *creg*$_{rd}$, [*address*]

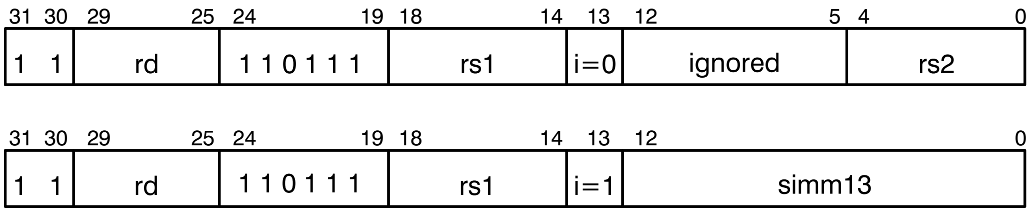
Description: The STDC instruction moves a doubleword from the coprocessor register pair, c[rd] and c[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: cp_disabled
cp_exception
memory_address_not_aligned
data_access_exception

Format:


31 30 29 25 24 19 18 14 13 12 5 4 0
1 1 | rd | 1 1 0 1 1 1 | rs1 | i=0 | ignored | rs2
31 30 29 25 24 19 18 14 13 12 0
1 1 | rd | 1 1 0 1 1 1 | rs1 | i=1 | simm13

STDCQ Store Doubleword Coprocessor Queue STDCQ

(Privileged Instruction)

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← CQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← CQ.INSTR

Assembler Syntax: std %cq, [*address*]

Description: The STDCQ instruction moves the front entry of the Coprocessor Queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4.

If the PSR's EC bit is set to zero or if no coprocessor is present, a cp_disabled trap will be generated. If a data_access_exception trap takes place during the effective address memory access, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: cp_disabled
cp_exception
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	1 1 0 1 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	1 1 0 1 1 0	rs1	i=1	simm13

STDF Store Doubleword Floating-Point STDF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← f[rd + 1]

Assembler Syntax: std *freg*$_{rd}$, [*address*]

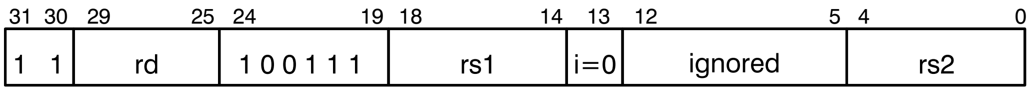
Description: The STDF instruction moves a doubleword from the floating-point register pair, f[rd] and f[rd+1], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The most significant word in the even-numbered destination register is written into memory at the effective address and the least significant memory word in the next odd-numbered register is written into memory at the effective address + 4.

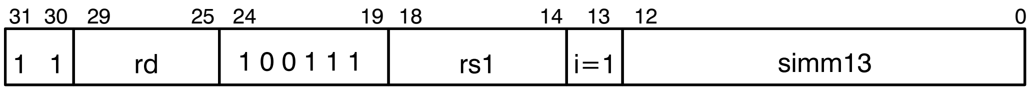
If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:





* NOTE: An attempt to execute <u>any</u> FP instruction will cause a pending FP exception to be recognized by the integer unit.

STDFQ — Store Doubleword Floating-Point Queue — STDFQ

(Privileged Instruction)

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FQ.ADDR
[r[rs1] + (r[rs2] or sign extnd(simm13)) + 4] ← FQ.INSTR

Assembler Syntax: std %fq, [*address*]

Description: The STDFQ instruction moves the front entry of the floating-point queue into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The address portion of the queue entry is written into memory at the effective address and the instruction portion of the entry is written into memory at the effective address + 4. If the FPU is in exception mode, the queue is then advanced to the next entry, or it becomes empty (as indicated by the *qne* bit in the FSR).

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If a trap takes place, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: fp_disabled
fp_exception*
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	1 0 0 1 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	1 0 0 1 1 0	rs1	i=1	simm13

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STF Store Floating-Point register STF

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← f[rd]

Assembler Syntax: st *freg*$_{rd}$, [*address*]

Description: The STF instruction moves a word from a floating-point register, f[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STF takes a trap, memory remains unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	1 0 0 1 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	1 0 0 1 0 0	rs1	i=1	simm13

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STFSR Store Floating-Point State Register STFSR

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ← FSR

Assembler Syntax: st %fsr, [*address*]

Description: The STFSR instruction moves the contents of the Floating-Point State Register into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. This instruction will wait for all pending FPops to complete execution before it writes the FSR into memory.

If the PSR's EF bit is set to zero or if no floating-point unit is present, an fp_disabled trap will be generated. If STFSR takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps:
fp_disabled
fp_exception*
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	1 0 0 1 0 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 1	rd	1 0 0 1 0 1	rs1	i=1	simm13

* NOTE: An attempt to execute any FP instruction will cause a pending FP exception to be recognized by the integer unit.

STH Store Halfword STH

Operation: [r[rs1] + (r[rs2] or sign extnd(simm13))] ⟵ r[rd]

Assembler Syntax: sth *reg$_{rd}$*, [*address*] synonyms: stuh, stsh

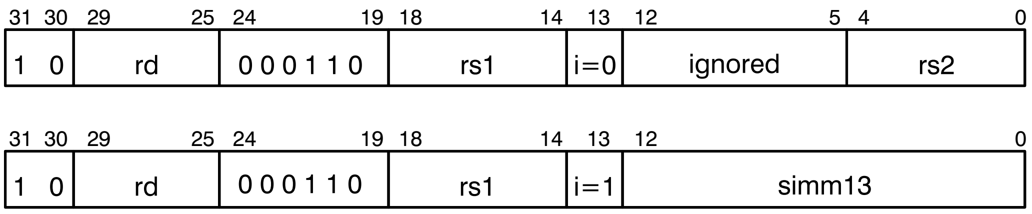
Description: The STH instruction moves the least significant halfword from the destination register, r[rd], into memory. The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

If STH takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be written to without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:



STHA — Store Halfword into Alternate space — STHA

(Privileged Instruction)

Operation: address space ← asi

[r[rs1] + (r[rs2] or sign extnd(simm13))] ← r[rd]

Assembler Syntax: stha *reg*$_{rd}$, [*address*]

synonyms: stuha, stsha

Description: The STHA instruction moves the least significant halfword from the destination register, r[rd], into memory. The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If STHA takes a trap, the contents of the memory address remain unchanged.

Traps: illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 0 1 1 0	rs1	i=0	asi	rs2

SUB Subtract SUB

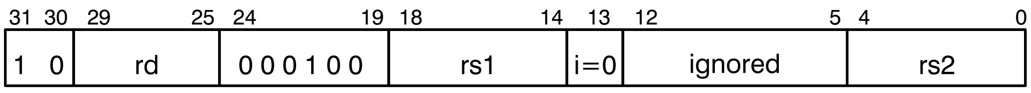
Operation: r[rd] $\leftarrow$ r[rs1] - (r[rs2] or sign extnd(simm13))

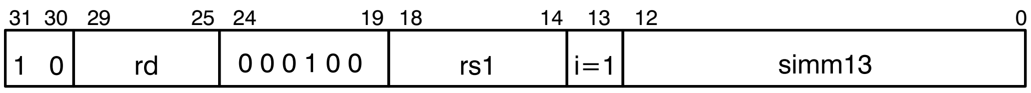
Assembler Syntax: sub *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: The SUB instruction subtracts either the contents of the register named in the *rs2* field, r[rs2], if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one, from register r[rs1]. The result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBcc Subtract and modify icc SUBcc

Operation:

```
r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)
c ← (not r[rs1]<31> AND operand2<31>)
        OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))
```

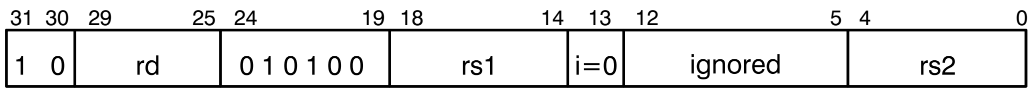
Assembler Syntax: subcc *regrs1*, *reg_or_imm*, *regrd*

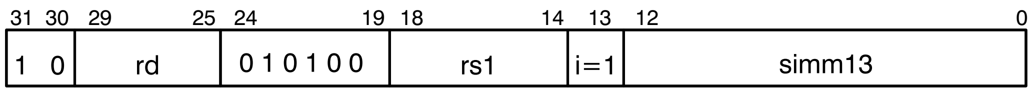
Description: The SUBcc instruction subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in register r[rd]. In addition, SUBcc modifies all the integer condition codes in the manner described above.

Programming note: A SUBcc instruction with *rd* = 0 can be used for signed and unsigned integer comparison.

Traps: none

Format:





SUBX Subtract with Carry SUBX

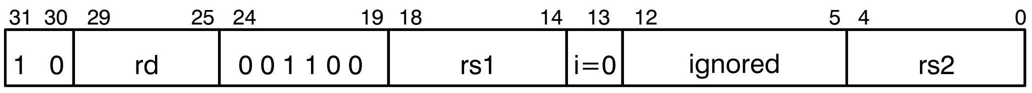
Operation: r[rd] ← r[rs1] - (r[rs2] or sign extnd(simm13)) - c

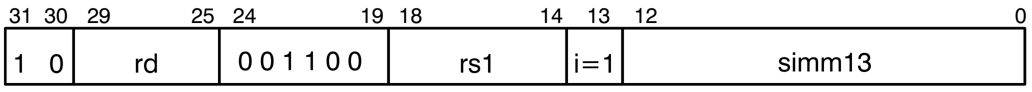
Assembler Syntax: subx *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SUBX subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field.

Traps: none

Format:





SUBXcc Subtract with Carry and modify icc SUBXcc

Operation:

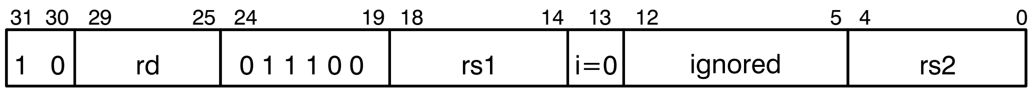
```
r[rd] ← r[rs1]  -  operand2  -  c, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND operand2<31> AND r[rd]<31>)
c ← (not r[rs1]<31> AND operand2<31>)
        OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))
```

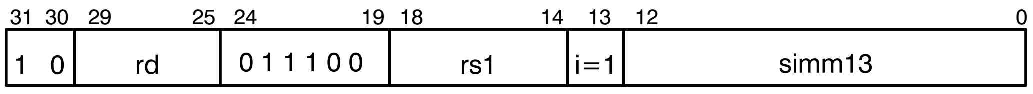
Assembler Syntax: subxcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: SUBXcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. It then subtracts the PSR's carry bit (*c*) from that result. The final result is placed in the register specified in the *rd* field. In addition, SUBXcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:





SWAP — Swap *r register* with memory — SWAP

Operation:

$\text{word} \leftarrow [\text{r[rs1]} + (\text{r[rs2] or sign extnd(simm13)})]$
$\text{temp} \leftarrow \text{r[rd]}$
$\text{r[rd]} \leftarrow \text{word}$
$\text{r[rs1]} + (\text{r[rs2] or sign extnd(simm13)}) \leftarrow \text{temp}$

Assembler Syntax: swap [*source*], reg_{rd}

Description: SWAP atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAP instructions simultaneously are guaranteed to execute them serially, in some order.

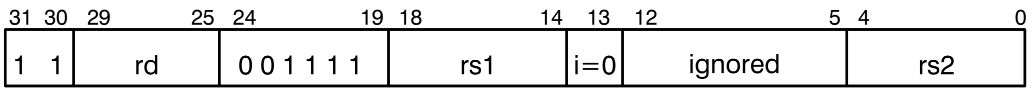
The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one.

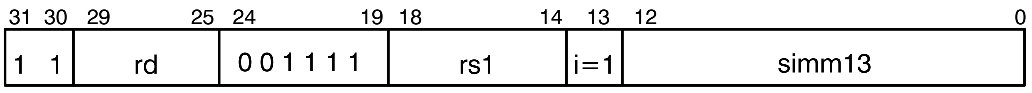
If SWAP takes a trap, the contents of the memory address and the destination register remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an address space can be accessed without setting up a register.

Traps: memory_address_not_aligned
data_access_exception

Format:





SWAPA Swap *r register* with memory in Alternate space SWAPA

(Privileged Instruction)

Operation:

address space ← asi
word ← [r[rs1] + r[rs2]]
temp ← r[rd]
r[rd] ← word
[r[rs1] + r[rs2]] ← temp

Assembler Syntax: swapa [*regsource*] *asi*, reg_{rd}

Description: **SWAPA atomically exchanges the contents of r[rd] with the contents of a memory location, i.e., without allowing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing SWAPA instructions simultaneously are guaranteed to execute them serially, in some order.**

The effective memory address is a combination of the address space value given in the *asi* field and the address derived by summing the contents of r[rs1] and r[rs2].

If SWAPA takes a trap, the contents of the memory address and the destination register remain unchanged.

Traps:

illegal_instruction (if i=1)
privileged_instruction (if S=0)
memory_address_not_aligned
data_access_exception

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 1	rd	0 1 1 1 1 1	rs1	i=0	asi	rs2

TADDcc — Tagged Add and modify icc — TADDcc

Operation:

```
r[rd] ← r[rs1]  +  operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
        OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
c ← (r[rs1]<31> AND operand2<31>
        OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
```

Assembler Syntax: taddcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: TADDcc adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TADDcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	1 0 0 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	1 0 0 0 0 0	rs1	i=1	simm13

TADDccTV Tagged Add (modify icc) Trap on Overflow TADDccTV

Operation:

```
result ← r[rs1] + operand2, where operand 2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND operand2<31> AND not r[rd]<31>)
        OR (not r[rs1]<31> AND not operand2<31> AND r[rd]<31>)
        OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (r[rs1]<31> AND operand2<31>
        OR (not r[rd]<31> AND (r[rs1]<31> OR operand2<31>))
r[rd] ← result
```

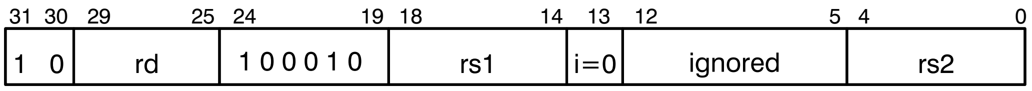
Assembler Syntax: taddcctv *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

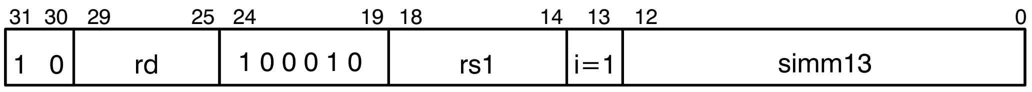
Description: TADDccTV adds the contents of r[rs1] to either the contents of r[rs2] if the instruction's *i* bit equals zero, or to a 13-bit, sign-extended immediate operand if *i* equals one. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TADDccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TADDccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps: tag_overflow

Format:





Ticc — Trap on integer condition codes — Ticc

Operation:

```
If condition true, then trap_instruction;
tt ← 128 + [r[rs1] + (r[rs2] or sign extnd(simm13))]<6:0>
else PC ← nPC
nPC ← nPC + 4
```

Assembler Syntax:

ta{,a}	*label*	
tn{,a}	*label*	
tne{,a}	*label*	synonym: tnz
te{,a}	*label*	synonym: tz
tg{,a}	*label*	
tle{,a}	*label*	
tge{,a}	*label*	
tl{,a}	*label*	
tgu{,a}	*label*	
tleu{,a}	*label*	
tcc{,a}	*label*	synonym: tgeu
tcs{,a}	*label*	synonym: tlu
tpos{,a}	*label*	
tneg{,a}	*label*	
tvc{,a}	*label*	
tvs{,a}	*label*	

Description: A Ticc instruction evaluates specific integer condition code combinations (from the PSR's *icc* field) based on the trap type as specified by the value in the instruction's *cond* field. If the specified combination of condition codes evaluates as true, and there are no higher-priority traps pending, then a trap_instruction trap is generated. If the condition codes evaluate as false, the trap is not generated.

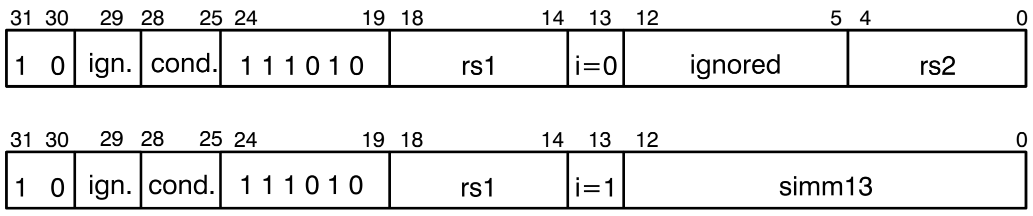
If a trap_instruction trap is generated, the *tt* field of the Trap Base Register (TBR) is written with 128 plus the least significant seven bits of r[rs1] plus either r[rs2] (bit field $i = 0$) or the 13-bit sign-extended immediate value contained in the instruction (bit field $i = 1$). See Section NO TAG for the complete definition of a trap.

Traps: trap_instruction

Ticc   Trap on integer condition codes   Ticc

Mnemonic	Cond.	Operation	icc Test
TN	0000	Trap Never	No test
TE	0001	Trap on Equal	z
TLE	0010	Trap on Less or Equal	z OR (n XOR v)
TL	0011	Trap on Less	n XOR v
TLEU	0100	Trap on Less or Equal, Unsigned	c OR z
TCS	0101	Trap on Carry Set (Less then, Unsigned)	c
TNEG	0110	Trap on Negative	n
TVS	0111	Trap on oVerflow Set	v
TA	1000	Trap Always	No test
TNE	1001	Trap on Not Equal	not z
TG	1010	Trap on Greater	not(z OR (n XOR v))
TGE	1011	Trap on Greater or Equal	not(n XOR v)
TGU	1100	Trap on Greater, Unsigned	not(c OR z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not c
TPOS	1110	Trap on Positive	not n
TVC	1111	Trap on oVerflow Clear	not v

Format:


31 30 29 28 25 24 19 18 14 13 12 5 4 0
1 0 | ign. | cond. | 1 1 1 0 1 0 | rs1 | i=0 | ignored | rs2
31 30 29 28 25 24 19 18 14 13 12 0
1 0 | ign. | cond. | 1 1 1 0 1 0 | rs1 | i=1 | simm13

ign. = ignored
cond. = condition

TSUBcc — Tagged Subtract and modify icc — TSUBcc

Operation:

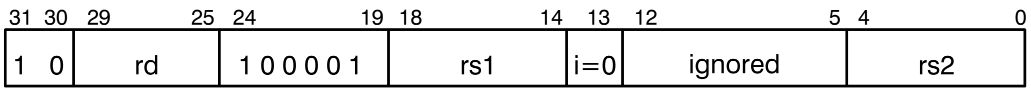
```
r[rd] ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ←     (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31>
        AND operand2<31> AND r[rd]<31>) OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
c ← (not r[rs1]<31> AND operand2<31>
        OR (r[rd]<31> AND (not r[rs1]<31> OR operand2<31>))
```

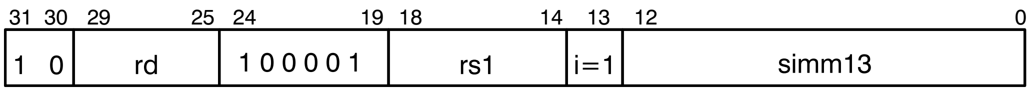
Assembler Syntax: tsubcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: TSUBcc subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. The result is placed in the register specified in the *rd* field. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero. TSUBcc modifies all the integer condition codes in the manner described above.

Traps: none

Format:





TSUBccTV Tagged Subtract (modify icc) TSUBccTV

Trap on Overflow

Operation:

```
result ← r[rs1] - operand2, where operand2 = (r[rs2] or sign extnd(simm13))
tv ← (r[rs1]<31> AND not operand2<31> AND not r[rd]<31>) OR (not r[rs1]<31>
          AND operand2<31> AND r[rd]<31>)
          OR (r[rs1]<1:0> ≠ 0 OR operand2<1:0> ≠ 0)
if tv = 1, then tag overflow trap; else
n ← r[rd]<31>
z ← if r[rd]=0 then 1, else 0
v ← tv
c ← (not(r[rs1]<31>) AND operand2<31> OR
(r[rd]<31> AND (not(r[rs1]<31>) OR operand2<31>))
r[rd] ← result
```

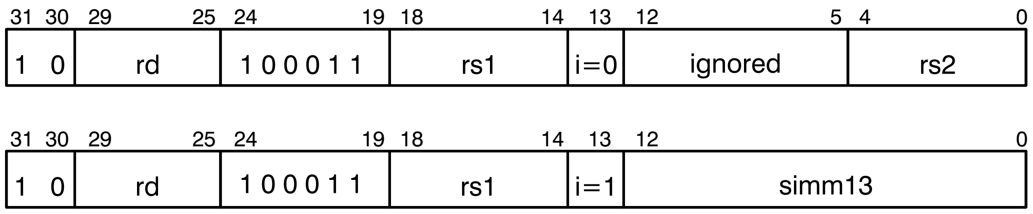
Assembler Syntax: tsubcctv *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: TSUBccTV subtracts either the contents of register r[rs2] (if the instruction's *i* bit equals zero) or the 13-bit, sign-extended immediate operand contained in the instruction (if *i* equals one) from register r[rs1]. In addition to the normal arithmetic overflow, an overflow condition also exists if bit 1 or bit 0 of either operand is not zero.

If TSUBccTV detects an overflow condition, a tag_overflow trap is generated and the destination register and condition codes remain unchanged. If no overflow is detected, TSUBccTV places the result in the register specified in the *rd* field and modifies all the integer condition codes in the manner described above (the overflow bit is, of course, set to zero).

Traps: tag_overflow

Format:



UNIMP Unimplemented instruction UNIMP

Operation: illegal instruction trap

Assembler Syntax: unimp *const22*

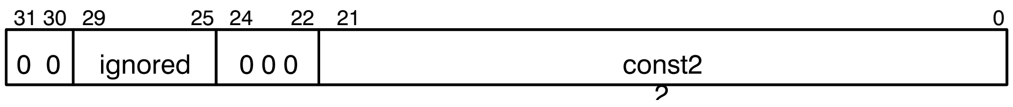
Description: Executing the UNIMP instruction causes an immediate illegal_instruction trap. The value in the const22 field is ignored.

Programming note: UNIMP can be used as part of the protocol for calling a function that is expected to return an aggregate value, such as a C-language structure.

1. An UNIMP instruction is placed after (not in) the delay slot after the CALL instruction in the calling function.
2. If the called function is expecting to return a structure, it will find the size of the structure that the caller expects to be returned as the const22 operand of the UNIMP instruction. The called function can check the opcode to make sure it is indeed UNIMP.
3. If the function is not going to return a structure, upon returning, it attempts to execute UNIMP rather than skipping over it as it should. This causes the program to terminate. The behavior adds some run-time checking to an interface that cannot be checked properly at compile time.

Traps: illegal_instruction

Format:



WRPSR Write Processor State Register WRPSR

(Privileged Instruction)

Operation: PSR ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg*$_{rs1}$, *reg_or_imm*, %psr

Description: WRPSR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the writable subfields of the PSR. However, if the result's CWP field would point to an unimplemented window, an illegal_instruction trap is generated and the PSR remains unchanged.

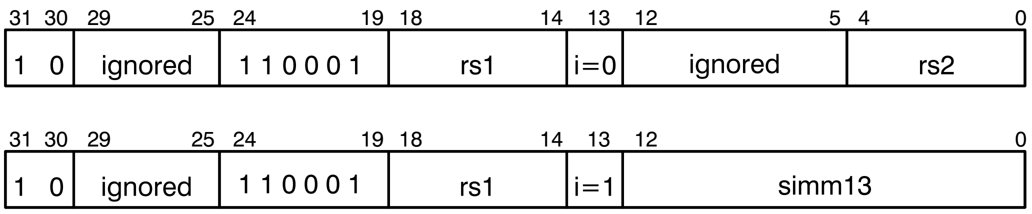
WRPSR is a delayed-write instruction:

1. If any of the three instructions following a WRPSR uses any PSR field that WRPSR modified, the value of that field is unpredictable. Note that any instruction which references a non−global register makes use of the CWP, so following WRPSR with three NOPs would be the safest course.
2. If a WRPSR instruction is updating the PSR's Processor Interrupt Level (PIL) to a new value and is simultaneously setting Enable Traps (ET) to one, this could result in an interrupt trap at a level equal to the old PIL value.
3. If any of the three instructions after a WRPSR instruction reads the modified PSR, the value read is unpredictable.
4. If any of the three instructions after a WRPSR is trapped, a subsequent RDPSR in the trap handler will get the register's new value.

Programming note: Two WRPSR instructions should be used when enabling traps and changing the PIL value. The first WRPSR should specify ET=0 with the new PIL value, and the second should specify ET=1 with the new PIL value.

Traps: illegal_instruction
privileged_instruction (if S=0)

Format:


31 30 29 25 24 19 18 14 13 12 5 4 0
1 0 ignored 1 1 0 0 0 1 rs1 i=0 ignored rs2
31 30 29 25 24 19 18 14 13 12 0
1 0 ignored 1 1 0 0 0 1 rs1 i=1 simm13

WRTBR Write Trap Base Register WRTBR

(Privileged Instruction)

Operation: TBR $\leftarrow$ r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg*$_{rs1}$, *reg_or_imm*, %tbr

Description: WRTBR does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the Trap Base Address field of the TBR.

WRTBR is a delayed-write instruction:

1. If any of the three instructions following a WRTBR causes a trap, the TBA used may be either the old or the new value.
2. If any of the three instructions after a WRTBR is trapped, a subsequent RDTBR in the trap handler will get the register's new TBA value.

Traps: privileged_instruction (if S=0)

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	ignored	1 1 0 0 1 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	ignored	1 1 0 0 1 1	rs1	i=1	simm13

WRWIM Write Window Invalid Mask register WRWIM

(Privileged Instruction)

Operation: WIM $\leftarrow$ r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg*$_{rs1}$, *reg_or_imm*, %wim

Description: WRWIM does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the writable bits of the WIM register.

WRWIM is a delayed-write instruction:

1. If any of the three instructions following a WRWIM is a SAVE, RESTORE, or RETT, the occurrence of window_overflow and window_underflow is unpredictable.
2. If any of the three instructions after a WRWIM instruction reads the modified WIM, the value read is unpredictable.
3. If any of the three instructions after a WRWIM is trapped, a subsequent RDWIM in the trap handler will get the register's new value.

Traps: privileged_instruction (if S=0)

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	ignored	1 1 0 0 1 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	ignored	1 1 0 0 1 0	rs1	i=1	simm13

WRY Write Y register WRY

Operation: Y ← r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: wr *reg$_{rs1}$*, *reg_or_imm*, %y

Description: WRY does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is written into the Y register.

WRY is a delayed-write instruction:

1. If any of the three instructions following a WRY is a MULScc or a RDY, the value of Y used is unpredictable.
2. If any of the three instructions after a WRY instruction reads the modified Y register, the value read is unpredictable.
3. If any of the three instructions after a WRY is trapped, a subsequent RDY in the trap handler will get the register's new value.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	ignored	1 1 0 0 0 0	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	ignored	1 1 0 0 0 0	rs1	i=1	simm13

XNOR Exclusive-Nor XNOR

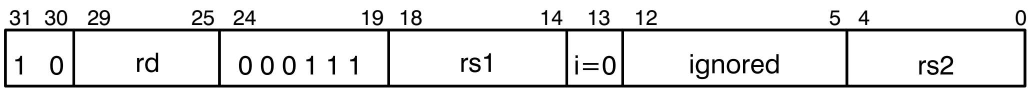
Operation: r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))

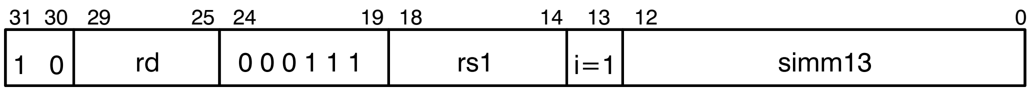
Assembler Syntax: xnor *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:





XNORcc — Exclusive-Nor and modify icc — XNORcc

Operation:

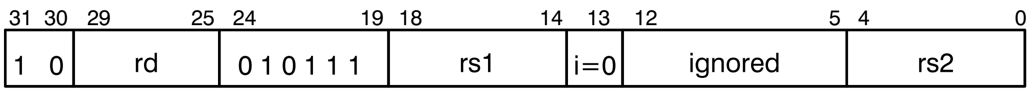
r[rd] ← r[rs1] XOR not(r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

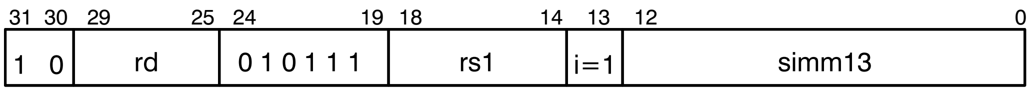
Assembler Syntax: xnorcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with the one's complement of either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. XNORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:





XOR Exclusive-Or XOR

Operation: r[rd] ◄— r[rs1] XOR (r[rs2] or sign extnd(simm13))

Assembler Syntax: xor *reg*$_{rs1}$*, reg_or_imm, reg*$_{rd}$

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd].

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 0 0 0 1 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 0 0 0 1 1	rs1	i=1	simm13

XORcc   Exclusive-Or and modify icc   XORcc

Operation:

r[rd] ← r[rs1] XOR (r[rs2] or sign extnd(simm13))
n ← r[rd]<31>
z ← if r[rd] =0 then 1, else 0
v ← 0
c ← 0

Assembler Syntax: xorcc *reg*$_{rs1}$, *reg_or_imm*, *reg*$_{rd}$

Description: This instruction does a bitwise logical XOR of the contents of register r[rs1] with either the contents of r[rs2] (if bit field i=0) or the 13-bit, sign-extended immediate value contained in the instruction (if bit field i=1). The result is stored in register r[rd]. XORcc also modifies all the integer condition codes in the manner described above.

Traps: none

Format:

31 30	29 25	24 19	18 14	13	12 5	4 0
1 0	rd	0 1 0 0 1 1	rs1	i=0	ignored	rs2

31 30	29 25	24 19	18 14	13	12 0
1 0	rd	0 1 0 0 1 1	rs1	i=1	simm13