Chapter 1

Instruction Set Manual

1.1 Compute Processor Instruction Set

Refer to Section 1.2 for concrete definitions of semantic helper functions such as *sign-extend-16-to-32* and *rotate-left*.

1.1.1 Register Conventions

The compute processor uses register conventions similar to those used in MIPS microprocessors. Procedures cannot rely on *caller-saved* registers retaining their values upon a procedure call. Procedures must restore the initial values of *callee-saved* registers before returning to their caller. The conventions are shown below.

Register Number	Assembly Alias	Saved by	Description
$0		n/a	Always has value zero.
$1	$at	caller	Assembler temporary clobbered by some assembler operations.
$2..$3		caller	First and second words of return value, respectively.
$4..$7		caller	First 4 arguments of function.
$8..$15		caller	General registers.
$16..$23		callee	General registers.
$24	$cst[i/o]	n/a	Static Network input/output port.
$25	$cgn[i/o]	n/a	General Dynamic Network input/output port.
$26	$csti2	n/a	Static Network input port #2.
$27	$cmn[i/o]	n/a	Memory Dynamic Network input/output port.
$28	$gp	callee	Global pointer. Points to start of tile's code and static data.
$29	$sp	callee	Stack pointer. Stack grows towards lower addresses.
$30		callee	General register.
$31		caller	Link register. Saves return address for function call.

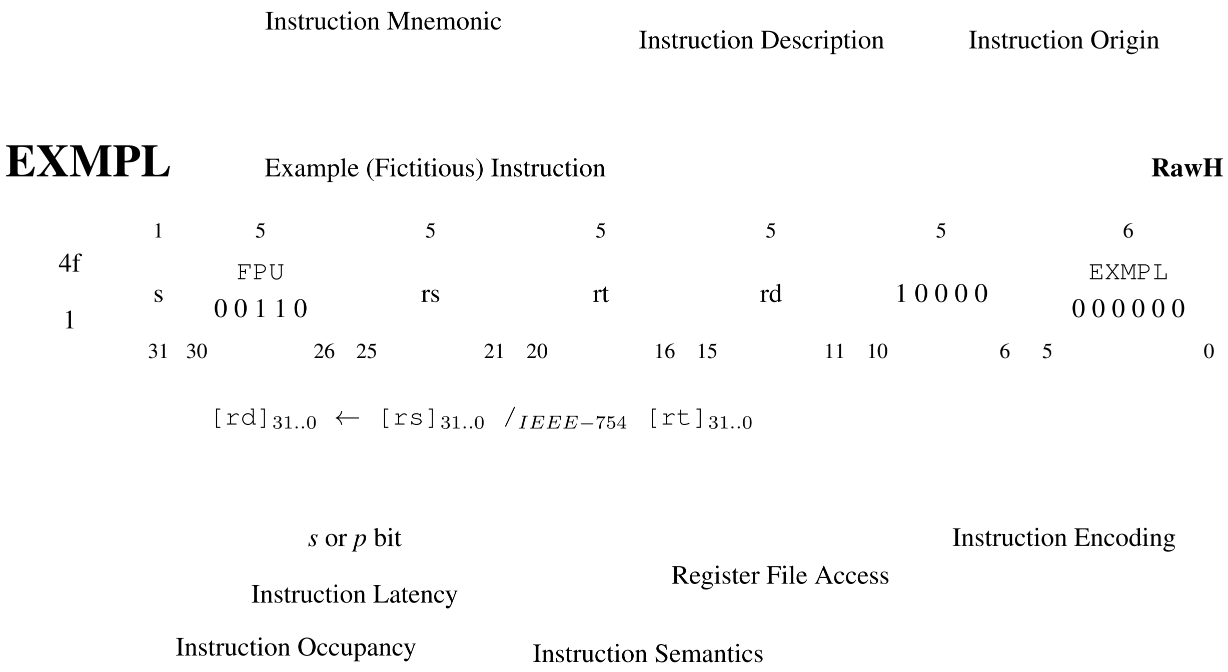
[0] Based on a Template provided by Prof. Michael Taylor of UC San Diego. Free for general use as long as this notice remains here.

1.1.2 Compute Processor Instruction Template

Shown below is an example instruction listing. The *instruction occupancy* is the number of compute processor issue cycles that are occupied by the instruction. Subsequent instructions must wait for this number of cycles before issuing. The *instruction latency* is the total number of cycles that must pass before a subsequent dependent instruction can issue. Suffixes of *d* and *f* indicate that an instruction uses the integer and floating-point divide units, respectively, for that number of cycles. Subsequent instructions that require a particular unit will stall until that unit is free. A suffix of *b* means that the instruction has an additional 3 cycles of occupancy on a branch misprediction. An occupancy of *c* means that the instruction takes at least 13 cycles if a cache line is evicted, 5 cycles if only an invalidation occurs, otherwise 1 cycle.

The *s* or *p* bits in the instruction encoding specify respectively whether 1) an instruction's output will be copied to csto in addition to the destination register, or 2) whether a branch is predicted taken or not.

Generally, the Raw compute processor attempts to inherit the MIPS instruction set mnemonics to the extent that it reduces the learning curve for new users of the system. However, the underlying instruction semantics have been "cleaned up"; for instance, interlocks have been added for load, branch, multiply and divide instructions (reducing the need to insert nops), and the FPU uses the same register set as the ALU. To this end, the *instruction origin* specifies whether the instruction semantics are very similar the MIPS instruction of the same name ("MIPS"), whether they are specific to the Raw architecture ("" or Raw), or to the Raw architecture extended with hardware instruction caching ("RawH"). Of course, the instruction encodings (including the presence of *s* and *p* bits) are completely different from MIPS.



ADDIU

Add Immediate

MIPS

	1	5	5	5	16
1	s	ADDIU 10111	rs	rt	simm16
	31	30 26	25 21	20 16	15 0

$\texttt{simm32}_{31..0} \leftarrow (\textit{sign-extend-16-to-32}\ \texttt{simm16})$

$\texttt{[rt]}_{31..0} \leftarrow \{\ \texttt{[rs]}_{31..0} + \texttt{simm32}\ \}_{31..0}$

ADDU

Add

MIPS

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	ADDU 100001
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..00} \leftarrow \{\ \texttt{[rs]}_{31..00} + \texttt{[rt]}_{31..00}\ \}_{31..0}$

AND

And Bitwise

MIPS

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	AND 100100
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..00} \leftarrow \texttt{[rs]}_{31..00}\ \&\ \texttt{[rt]}_{31..00}$

ANDI

And Bitwise Immediate

MIPS

	1	5	5	5	16
1	s	ANDI 00100	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

$\texttt{[rt]}_{31..16} \leftarrow \texttt{[rs]}_{31..16}$

$\texttt{[rt]}_{15..00} \leftarrow \texttt{[rs]}_{15..00}\ \&\ \texttt{uimm16}$

AUI

Add Upper Immediate

	1	5	5	5	16
1	s	AUI 0 0 1 1 1	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

```
[rt]$_{31..16}$ ← { [rs]$_{31..16}$ + uimm16 }$_{15..0}$
[rt]$_{15..00}$ ← [rs]$_{15..00}$
```

B

Branch Unconditional (Assembly Macro)

1 *b <label>*

```
PC$_{31..00}$ ← <label>
```

BEQ

Branch if equal

	1	5	5	5	16
1b	p	BEQ 1 1 0 1 1	rs	rt	boffs16
	31	30 26	25 21	20 16	15 0

```
if ([rs] == [rt])
    PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-16-to-30* boffs16) }$_{29..00}$
    PC$_{01..00}$ ← 0
```

BGEZ

Branch if greater than or equal to zero (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BGEZ 0 0 0 1 0	boffs16
	31	30 26	25 21	20 16	15 0

```
if (![rs]$_{31}$)
    PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-16-to-30* boffs16) }$_{29..00}$
    PC$_{01..00}$ ← 0
```

BGEZAL Branch if greater than or equal to zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BGEZAL 1 0 0 1 0	boffs16
	31	30 26	25 21	20 16	15 0

```
if (![rs]$_{31}$)
   [31]    ← { PC + 4 }$_{31..00}$
   PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-16-to-30* boffs16) }$_{29..00}$
   PC$_{01..00}$ ← 0
else
   [31]    ← { PC + 4 }$_{31..00}$
```

BGTZ Branch if greater than zero (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BGTZ 0 0 0 1 1	boffs16
	31	30 26	25 21	20 16	15 0

```
if (![rs]$_{31}$ && ([rs] != 0))
   PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-16-to-30* boffs16) }$_{29..00}$
   PC$_{01..00}$ ← 0
```

BL Branch Long **RawH**

	1	5	26
1	1	BL 1 1 1 0 0	boffs26
	31	30 26	25 0

```
PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-26-to-30* boffs26) }$_{29..00}$
PC$_{01..00}$ ← 0
```

BLAL Branch Long and Link **RawH**

	1	5	26
1	1	BLAL 1 1 1 0 1	boffs26
	31	30 26	25 0

```
[31]    ← { PC + 4 }$_{31..00}$
PC$_{31..02}$ ← { PC$_{31..02}$ + (*sign-extend-26-to-30* boffs26) }$_{29..00}$
PC$_{01..00}$ ← 0
```

BLEZ

Branch if less than or equal to zero (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BLEZ 0 0 0 0 1	boffs16
	31	30 26	25 21	20 16	15 0

```
if ($[rs]_{31}$ || ([rs] == 0))
   $PC_{31..02}$ $\leftarrow$ { $PC_{31..02}$ + (*sign-extend-16-to-30* boffs16) $\}_{29..00}$
   $PC_{01..00}$ $\leftarrow$ 0
```

BLTZ

Branch if less than zero (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BLTZ 0 0 0 0 0	boffs16
	31	30 26	25 21	20 16	15 0

```
if ($[rs]_{31}$)
   $PC_{31..02}$ $\leftarrow$ { $PC_{31..02}$ + (*sign-extend-16-to-30* boffs16) $\}_{29..00}$
   $PC_{01..00}$ $\leftarrow$ 0
```

BLTZAL

Branch if less than zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	BLTZAL 0 0 0 0 0	boffs16
	31	30 26	25 21	20 16	15 0

```
if ($[rs]_{31}$)
   [31]    $\leftarrow$ { PC + 4 $\}_{31..00}$
   $PC_{31..02}$ $\leftarrow$ { $PC_{31..02}$ + (*sign-extend-16-to-30* boffs16) $\}_{29..00}$
   $PC_{01..00}$ $\leftarrow$ 0
else
   [31]    $\leftarrow$ { PC + 4 $\}_{31..00}$
```

BNE

Branch if not equal

	1	5	5	5	16
1b	p	BNE 1 1 0 1 0	rs	rt	boffs16
	31	30 26	25 21	20 16	15 0

```
if ([rs] != [rt])
   $PC_{31..02}$ $\leftarrow$ { $PC_{31..02}$ + (*sign-extend-16-to-30* boffs16) $\}_{29..00}$
   $PC_{01..00}$ $\leftarrow$ 0
```

BNEA

Branch if not equal and add

	1	5	5	5	16
1b	p	BNEA 1 1 0 0 1	rs	rt	boffs16
	31	30 26	25 21	20 16	15 0

```
if ([rs] != [rt])
   $PC_{31..02} \leftarrow \{ PC_{31..02} + (\textit{sign-extend-16-to-30}\ boffs16) \}_{29..00}$
   $PC_{01..00} \leftarrow 0$
[rs] = [rs] + SR[BR_INC]
```

CLZ

Count Leading Zero

	1	5	5	5	5	5	6
1	s	SPECIAL 0 0 0 0 0	rs	0 0 0 0 0	rd	0 0 0 0 0	CLZ 1 1 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$[rd]_{05..00} \leftarrow \sum_{i=0}^{31} ([rs]_{31..i}\ ?\ 0 : 1)$
$[rd]_{31..06} \leftarrow 0$
```

DIV

Divide Signed

	1	5	5	5	5	5	6
42d 1	0	SPECIAL 0 0 0 0 0	rs	rt	0 0 0 0 0	0 0 0 0 0	DIV 0 1 1 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$LO \leftarrow \{ \{ [rs]\ /_{signed}\ [rt] \}_{31..0} \}_{31..0}$
$HI \leftarrow \{ \{ [rs]\ \%_{signed}\ [rt] \}_{31..0} \}_{63..32}$

if ([rt] == 0)
   $HI \leftarrow [rs]$
   if ($[rs]_{31}$)
      $LO \leftarrow 1$
   else
      $LO \leftarrow -1$
```

DIVU Divide Unsigned

	1	5	5	5	5	5	6
42d 1	0	SPECIAL 00000	rs	rt	00000	00000	DIVU 011011
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
LO ← { [rs] $/_{unsigned}$ [rt] $\}_{31..0}$ $\}_{31..0}$
HI ← { [rs] $\%_{unsigned}$ [rt] $\}_{31..0}$ $\}_{63..32}$

if ([rt] == 0)
   HI ← [rs]
   LO ← -1
```

J Jump

	1	5	5	5	16
1	p	REGIMM 11000	00000	J 01100	targ16
	31	30 26	25 21	20 16	15 0

```
PC$_{31..02}$ ← (zero-extend-16-to-30 targ16)
PC$_{01..00}$ ← 0
```

JAL Jump and link

	1	5	5	5	16
1	p	REGIMM 11000	rs	JAL 11100	targ16
	31	30 26	25 21	20 16	15 0

```
[31]      ← { PC + 4 $\}_{31..00}$
PC$_{31..02}$ ← (zero-extend-16-to-30 targ16)
PC$_{01..00}$ ← 0
```

JALR Jump and link through Register

	1	5	5	5	5	5	6
4	0	SPECIAL 00000	rs	00000	00000	00000	JALR 001001
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
[31]      ← { PC + 4 $\}_{31..00}$
PC$_{31..02}$ ← [rs]$_{31..02}$
PC$_{01..00}$ ← 0
```

JEQL

Jump if not equal and link

	1	5	5	5	16
1b	p	JEQL 1 1 1 1 1	rs	rt	targ16
	31	30 26	25 21	20 16	15 0

```
if ([rs] == [rt])
    [31]    ← { PC + 4 }$_{31..00}$
    PC$_{31..02}$ ← (*zero-extend-16-to-30* targ16)
    PC$_{01..00}$ ← 0
else
    [31]    ← { PC + 4 }$_{31..00}$
```

JGEZL

Jump if greater than or equal to zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	JGEZL 1 0 1 1 0	targ16
	31	30 26	25 21	20 16	15 0

```
if (![rs]$_{31}$)
    [31]    ← { PC + 4 }$_{31..00}$
    PC$_{31..02}$ ← (*zero-extend-16-to-30* targ16)
    PC$_{01..00}$ ← 0
else
    [31]    ← { PC + 4 }$_{31..00}$
```

JGTZL

Jump if greater than zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	JGTZL 1 0 1 1 1	targ16
	31	30 26	25 21	20 16	15 0

```
if (![rs]$_{31}$ && ([rs] != 0))
    [31]    ← { PC + 4 }$_{31..00}$
    PC$_{31..02}$ ← (*zero-extend-16-to-30* targ16)
    PC$_{01..00}$ ← 0
else
    [31]    ← { PC + 4 }$_{31..00}$
```

JLEZL

Jump if less than or equal to zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	JLEZL 1 0 1 0 1	targ16
	31	30 26	25 21	20 16	15 0

```
if ([rs]$_{31}$ || ([rs] == 0))
   [31]      ← { PC + 4 }$_{31..00}$
   PC$_{31..02}$ ← (zero-extend-16-to-30 targ16)
   PC$_{01..00}$ ← 0
else
   [31]      ← { PC + 4 }$_{31..00}$
```

JLTZL

Jump if less than zero and link (signed)

	1	5	5	5	16
1b	p	REGIMM 1 1 0 0 0	rs	JLTZL 1 0 1 0 0	targ16
	31	30 26	25 21	20 16	15 0

```
if ([rs]$_{31}$)
   [31]      ← { PC + 4 }$_{31..00}$
   PC$_{31..02}$ ← (zero-extend-16-to-30 targ16)
   PC$_{01..00}$ ← 0
else
   [31]      ← { PC + 4 }$_{31..00}$
```

JNEL

Jump if not equal and link

	1	5	5	5	16
1b	p	JNEL 1 1 1 1 0	rs	rt	targ16
	31	30 26	25 21	20 16	15 0

```
if ([rs] != [rt])
   [31]      ← { PC + 4 }$_{31..00}$
   PC$_{31..02}$ ← (zero-extend-16-to-30 targ16)
   PC$_{01..00}$ ← 0
else
   [31]      ← { PC + 4 }$_{31..00}$
```

JR

Jump through Register

	1	5	5	5	5	5	6
4	0	SPECIAL 00000	rs	00000	00000	00000	JR 001000
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
PC$_{31..02}$ ← [rs]$_{31..02}$
PC$_{01..00}$ ← 0
```

JRHOFF

Jump through Register and Disable Hardware ICaching **RawH**

	1	5	5	5	5	5	6
4	0	SPECIAL 00000	rs	00000	00000	00000	JRHOFF 001011
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
PC$_{31..02}$ ← [rs]$_{31..02}$
PC$_{01..00}$ ← 0
```

JRHON

Jump through Register and Enable Hardware ICaching **RawH**

	1	5	5	5	5	5	6
4	0	SPECIAL 00000	rs	00000	00000	00000	JRHON 001010
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
PC$_{31..02}$ ← [rs]$_{31..02}$
PC$_{01..00}$ ← 0
```

LB

Load Byte

	1	5	5	5	16
3 1	s	LB 10000	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea    ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
[rt]  ← (sign-extend-8-to-32 (cache-read-byte ea))
```

LBU Load Byte Unsigned

	1	5	5	5	16
3 1	s	LBU 1 0 0 0 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea              ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
[rt]$_{31..8}$  ← 0
[rt]$_{7..0}$   ← (cache-read-byte ea)
```

LH Load Halfword

	1	5	5	5	16
3 1	s	LH 1 0 0 1 0	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea    ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
[rt]  ← (sign-extend-16-to-32 (cache-read-half-word ea))
```

LHU Load Halfword Unsigned

	1	5	5	5	16
3 1	s	LHU 1 0 0 1 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea               ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
[rt]$_{31..16}$  ← 0
[rt]$_{15..0}$   ← (cache-read-half-word ea)
```

LI Load Immediate (Assembly Macro) **MIPS**

1-2 *li rd, uimm32*
li! rd, uimm32

```
[rd]$_{31..0}$  ← uimm32$_{31..0}$
```

LW Load Word

	1	5	5	5	16
3 1	s	LW 1 0 1 0 0	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

ea ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..0}$
[rt] ← (*cache-read-word* ea)

MAGIC User-specified simulator function

	1	5	5	5	10	6
1	s	SPECIAL 0 0 0 0 0	rs	rt	code	MAGIC 0 0 0 0 1
	31	30 26	25 21	20 16	15 6	5 0

[rt]$_{31..00}$ ← (*user˙function* code [rs]) – On BTL simulator

[rt]$_{31..00}$ ← *unspecified value* – On RTL and hardware

MFFD Move from FD

	1	5	5	5	5	5	6
1f	s	SPECIAL 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	rd	0 0 0 0 0	MFFD 0 1 0 1 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

[rd]$_{31..00}$ ← FD$_{31..00}$

MFHI Move from HI

	1	5	5	5	5	5	6
1d	s	SPECIAL 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	rd	0 0 0 0 0	MFHI 0 1 0 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

[rd]$_{31..00}$ ← HI$_{31..00}$

MFLO

Move from LO

	1	5	5	5	5	5	6
1d	s	SPECIAL 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	rd	0 0 0 0 0	MFLO 0 1 0 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..00} \leftarrow \texttt{LO}_{31..00}$

MOVE

MOVE (Assembly Macro) **MIPS**

1 *move rd, rt*
move! rd, rt

$\texttt{[rd]}_{31..0} \leftarrow \texttt{[rt]}_{31..0}$

MTFD

Move to FD

	1	5	5	5	5	5	6
1f	0	SPECIAL 0 0 0 0 0	rs	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	MTFD 0 1 0 1 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{FD}_{31..00} \leftarrow \texttt{[rs]}_{31..00}$

MTHI

Move to HI

	1	5	5	5	5	5	6
1d	0	SPECIAL 0 0 0 0 0	rs	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	MTHI 0 1 0 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{HI}_{31..00} \leftarrow \texttt{[rs]}_{31..00}$

MTLO

Move to LO

	1	5	5	5	5	5	6
1d	0	SPECIAL 0 0 0 0 0	rs	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	MTLO 0 1 0 0 1 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{LO}_{31..00} \leftarrow \texttt{[rs]}_{31..00}$

MULLO Multiply Low Signed

	1	5	5	5	5	5	6
2 1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	MULLO 0 1 1 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$[\texttt{rd}]_{31..00} \leftarrow \{ \; [\texttt{rs}]_{31..00} \; *_{signed} \; [\texttt{rt}]_{31..00}) \; \}_{31..0}$$

MULLU Multiply Low Unsigned

	1	5	5	5	5	5	6
2 1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	MULLU 0 1 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$[\texttt{rd}]_{31..00} \leftarrow \{ \; [\texttt{rs}]_{31..00} \; *_{unsigned} \; [\texttt{rt}]_{31..00}) \; \}_{31..0}$$

MULHI Multiply High Signed

	1	5	5	5	5	5	6
2 1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	MULHI 1 0 1 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$[\texttt{rd}]_{31..00} \leftarrow \{ \; [\texttt{rs}]_{31..00} \; *_{signed} \; [\texttt{rt}]_{31..00}) \; \}_{63..32}$$

MULHU Multiply High Unsigned

	1	5	5	5	5	5	6
2 1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	MULHU 1 0 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$[\texttt{rd}]_{31..00} \leftarrow \{ \; [\texttt{rs}]_{31..00} \; *_{unsigned} \; [\texttt{rt}]_{31..00}) \; \}_{63..32}$$

NOR Nor Bitwise **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	NOR 1 0 0 1 1 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$[\texttt{rd}]_{31..00} \leftarrow {\sim}([\texttt{rs}]_{31..00} \; | \; [\texttt{rt}]_{31..00})$$

OR

Or Bitwise **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	OR 1 0 0 1 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$\texttt{[rd]}_{31..00} \leftarrow \texttt{[rs]}_{31..00} \mid \texttt{[rt]}_{31..00}$$

ORI

Or Bitwise Immediate **MIPS**

	1	5	5	5	16
1	s	ORI 0 0 0 1 0	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

$$\texttt{[rt]}_{31..16} \leftarrow \texttt{[rs]}_{31..16}$$
$$\texttt{[rt]}_{15..00} \leftarrow \texttt{[rs]}_{15..00} \mid \texttt{uimm16}$$

POPC

Population Count

	1	5	5	5	5	5	6
1	s	SPECIAL 0 0 0 0 0	rs	0 0 0 0 0	rd	0 0 0 0 0	POPC 1 1 1 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$\texttt{[rd]}_{04..00} \leftarrow \sum_{i=0}^{31} \texttt{[rs]}_{i}$$
$$\texttt{[rd]}_{31..05} \leftarrow 0$$

RLM

Rotate Left and Mask

	6	5	5	5	5	5	1
1	RLM 1 0 1 S 0 0	rs	rt	ra	mb	me	z
	31 26	25 21	20 16	15 11	10 6	5 1	00

$$\texttt{mask} \leftarrow (\textit{create-mask}\ \texttt{mb me z})$$
$$\texttt{[rt]}_{31..0} \leftarrow (\textit{left-rotate}\ \texttt{[rs]}_{31..0}\ \texttt{ra})\ \texttt{\&}\ \texttt{mask}$$

RLMI

Rotate Left and Masked Insert

	6	5	5	5	5	5	1
1	RLMI 1 0 1 S 0 1	rs	rt	ra	mb	me	z
	31 26	25 21	20 16	15 11	10 6	5 1	00

```
mask            ← (create-mask mb me z)
[rt]$_{31..0}$  ← ((left-rotate [rs]$_{31..0}$ ra) & mask) | ([rt]$_{31..0}$ & ˜mask)
```

RLVM

Rotate Left Variable and Mask

	6	5	5	5	5	5	1
1	RLVM 1 0 1 S 1 0	rs	rt	rd	mb	me	z
	31 26	25 21	20 16	15 11	10 6	5 1	00

```
mask            ← (create-mask mb me z)
ra              ← [rt]$_{31..0}$
[rd]$_{31..0}$  ← (left-rotate [rs]$_{31..0}$ ra) & mask
```

RRM

Rotate Right and Mask (Assembly Macro)

1 *rrm rt, rs, ra, mask*
rrm! rt, rs, ra, mask

```
[rt]$_{31..0}$  ← (right-rotate [rs]$_{31..0}$ ra) & mask
```

(instruction is implemented using RLM; same set of masks are valid)

RRMI

Rotate Right and Mask (Assembly Macro)

1 *rrmi rt, rs, ra, mask*
rrmi! rt, rs, ra, mask

```
[rt]$_{31..0}$  ← ((right-rotate [rs]$_{31..0}$ ra) & mask) | ([rt]$_{31..0}$ & ˜mask)
```

(instruction is implemented using RLMI; same set of masks are valid)

SLL Shift Left Logical MIPS

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	sa	00000	SLL 000000
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$\texttt{[rt]}_{31..sa} \leftarrow \texttt{[rs]}_{(31-sa)..0}$
$\texttt{[rt]}_{(sa-1)..0} \leftarrow \texttt{0}$
```

SLLV Shift Left Logical Variable MIPS

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	SLLV 000100
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$\texttt{sa} \leftarrow \texttt{[rt]}_{4..0}$
$\texttt{[rd]}_{31..sa} \leftarrow \texttt{[rs]}_{(31-sa)..0}$
$\texttt{[rd]}_{(sa-1)..0} \leftarrow \texttt{0}$
```

SLT Set Less Than Signed MIPS

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	SLT 101010
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$\texttt{[rd]}_{00} \leftarrow \texttt{([rs]}_{31..00} <_{signed} \texttt{[rt]}_{31..00}\texttt{) ? 1 : 0}$
$\texttt{[rd]}_{31..01} \leftarrow \texttt{0}$
```

SLTI Set Less Than Immediate Signed MIPS

	1	5	5	5	16
1	s	SLTI 10110	rs	rt	simm16
	31	30 26	25 21	20 16	15 0

```
$\texttt{simm32} \leftarrow \texttt{(}\textit{sign-extend-16-to-32}\ \texttt{simm16)}$
$\texttt{[rt]}_{00} \leftarrow \texttt{([rs]}_{31..00} <_{signed} \texttt{simm32}_{31..00}\texttt{) ? 1 : 0}$
$\texttt{[rt]}_{31..01} \leftarrow \texttt{0}$
```

SLTIU Set Less Than Immediate Unsigned **MIPS**

	1	5	5	5	16
1	s	SLTIU 10101	rs	rt	simm16
	31	30 26	25 21	20 16	15 0

```
uimm32          ← (sign-extend-16-to-32 simm16)
$[rt]_{00}$      ← $([rs]_{31..00} <_{unsigned}$ $uimm32_{31..00})$ ? 1 : 0
$[rt]_{31..01}$  ← 0
```

SLTU Set Less Than Unsigned **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	SLTU 101011
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$[rd]_{00}$      ← $([rs]_{31..00} <_{unsigned} [rt]_{31..00})$ ? 1 : 0
$[rd]_{31..01}$  ← 0
```

SRA Shift Right Arithmetic **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	sa	00000	SRA 000011
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
$[rt]_{(31-sa)..00}$ ← $[rs]_{31..sa}$
$[rt]_{31..(31-sa)}$ ← $[rs]_{31}$
```

SRAV Shift Right Arithmetic Variable **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	SRAV 000111
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
sa                    ← $[rt]_{4..0}$
$[rd]_{(31-sa)..00}$ ← $[rs]_{31..sa}$
$[rd]_{31..(31-sa)}$ ← $[rs]_{31}$
```

SRL Shift Right Logical **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	sa	00000	SRL 000010
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rt]}_{(31-sa)..0} \leftarrow \texttt{[rs]}_{31..sa}$

$\texttt{[rt]}_{(31..(32-sa)} \leftarrow \texttt{0}$

SRLV Shift Right Logical Variable **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	sa	00000	SRLV 000110
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{sa} \leftarrow \texttt{[rt]}_{4..0}$

$\texttt{[rd]}_{(31-sa)..00} \leftarrow \texttt{[rs]}_{31..sa}$

$\texttt{[rd]}_{31..(32-sa)} \leftarrow \texttt{0}$

SUBU Subtract **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 00000	rs	rt	rd	00000	SUBU 100011
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..00} \leftarrow \{ \texttt{[rs]}_{31..00} \texttt{ - } \texttt{[rt]}_{31..00} \}_{31..0}$

SB Store Byte **MIPS**

	1	5	5	5	16
1	0	SB 01000	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

$\texttt{ea} \leftarrow \{ \texttt{[base] + (}\textit{sign-extend-16-to-32}\texttt{ soffs16)} \}_{31..0}$

$\texttt{(}\textit{cache-write-byte}\texttt{ ea [rt])}$

SH

Store Halfword **MIPS**

	1	5	5	5	16
1	0	SH 0 1 0 1 0	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea      ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
(cache-write-half-word ea [rt])
```

SW

Store Word **MIPS**

	1	5	5	5	16
1	0	SW 0 1 1 0 0	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea      ← { [base] + (sign-extend-16-to-32 soffs16) }$_{31..0}$
(cache-write-word ea [rt])
```

XOR

Exclusive-Or Bitwise **MIPS**

	1	5	5	5	5	5	6
1	s	SPECIAL 0 0 0 0 0	rs	rt	rd	0 0 0 0 0	XOR 1 0 0 1 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
[rd]$_{31..0}$ ← [rs]$_{31..00}$ ^ [rt]$_{31..00}$
```

XORI

Exclusive-Or Bitwise Immediate **MIPS**

	1	5	5	5	16
1	s	XORI 0 0 0 1 1	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

```
[rt]$_{31..16}$ ← [rs]$_{31..16}$
[rt]$_{15..00}$ ← [rs]$_{15..00}$ ^ uimm16
```

ADD.s Add (single precision floating point)

	1	5	5	5	5	5	6
4 1	s	FPU 0 0 1 1 0	rs	rt	rd	1 0 0 0 0	ADD.s 0 0 0 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$\texttt{[rd]}_{31..0} \leftarrow \texttt{[rs]}_{31..0} +_{IEEE-754} \texttt{[rt]}_{31..0}$$

C.⟨XXX⟩.s Compare (single precision floating point)

	1	5	5	5	5	5	6
4 1	s	FPU 0 0 1 1 0	rs	rt	rd	1 0 0 0 0	C.⟨XXX⟩.s 0 0 code$_{3..0}$
	31	30 26	25 21	20 16	15 11	10 6	5 0

$$\begin{aligned}
\{\texttt{invalid}_0\ \texttt{result}_0\} &\leftarrow (\textit{floating-point-compare}\ \langle\texttt{XXX}\rangle\ \texttt{[rs]}_{31..0}\ \texttt{[rt]}_{31..0}) \\
\texttt{[rd]}_0 &\leftarrow \texttt{result}_0 \\
\texttt{[rd]}_{31..1} &\leftarrow \texttt{0} \\
\texttt{SR[FPSR]}_4 &\leftarrow \texttt{SR[FPSR]}_4 \ | \ \texttt{invalid}_0
\end{aligned}$$

e.g.,

```
    c.ult.s $4, $5, $7
```

The code values of 8..15 correspond to instructions that set the `invalid` bit of the floating point status register (`FPSR`) when an unordered comparison occurs. The behavior of the helper function *floating-point-compare* matches the MIPS ISA [**?**] and is shown in the following table:

Predicate			*floating-point-compare* outputs for each comparison outcome					
code	Mnemonic ⟨XXX⟩	Description	result$_0$				invalid$_0$	
			>	<	==	unordered	>, <, ==	unordered
0	F	False	0	0	0	0	0	0
1	UN	Unordered		0	0	1		
2	EQ	Equal		0	1	0		
3	UEQ	Unordered ==		0	1	1		
4	OLT	Ordered <		1	0	0		
5	ULT	Unordered or <		1	0	1		
6	OLE	Ordered ≤		1	1	0		
7	ULE	Unordered or ≤		1	1	1		
8	SF	Signaling False		0	0	0		1
9	NGLE	Not (> or ≤)		0	0	1		
10	SEQ	Signaling ==		0	1	0		
11	NGL	Not (< or >)		0	1	1		
12	LT	<		1	0	0		
13	NGE	Not ≥		1	0	1		
14	LE	≤		1	1	0		
15	NGT	Not >		1	1	1		

CVT.s

Convert from integer to float

	1	5	5	5	5	5	6
4 1	s	FPU 00110	rs	00000	rd	10100	CVT.s 100000
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow (\textit{convert-from-integer-to-float}\ \ \texttt{[rs]}_{31..0})$

CVT.w

Convert from float to integer, with round to nearest even

	1	5	5	5	5	5	6
4 1	s	FPU 00110	rs	00000	rd	10000	CVT.w 100100
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow (\textit{convert-from-float-to-integer-round-nearest-even}\ \ \texttt{[rs]}_{31..0})$

DIV.s

Divide (single precision floating point)

	1	5	5	5	5	5	6
12f 1	s	FPU 00110	rs	rt	00000	10000	DIV.s 000011
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{FD}_{31..0} \leftarrow \texttt{[rs]}_{31..0}\ /_{IEEE-754}\ \texttt{[rt]}_{31..0}$

MUL.s

Multiply (single precision floating point)

	1	5	5	5	5	5	6
4 1	s	FPU 00110	rs	rt	rd	10000	MUL.s 000010
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow \texttt{[rs]}_{31..0}\ *_{IEEE-754}\ \texttt{[rt]}_{31..0}$

NEG.s

Negate (single precision floating point)

	1	5	5	5	5	5	6
4 1	s	FPU 00110	rs	00000	rd	10000	NEG.s 000111
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow -_{IEEE-754}\ \texttt{[rs]}_{31..0}$

SUB.s

Subtract (single precision floating point)

	1	5	5	5	5	5	6
4 1	s	FPU 0 0 1 1 0	rs	rt	rd	1 0 0 0 0	SUB.s 0 0 0 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow \texttt{[rs]}_{31..0} -_{IEEE-754} \texttt{[rt]}_{31..0}$

TRUNC.w

Convert from float to integer, with truncation

	1	5	5	5	5	5	6
4 1	s	FPU 0 0 1 1 0	rs	0 0 0 0 0	rd	1 0 0 0 0	TRUNC.w 0 0 1 1 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{[rd]}_{31..0} \leftarrow (\textit{convert-from-float-to-integer-truncate}\ \texttt{[rs]}_{31..0})$

DRET

Return from User Interrupt

	1	5	5	5	5	5	6
4	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	DRET 0 0 0 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{PC}_{31..02} \leftarrow \texttt{SR[EX\_UPC]}_{31..02}$
$\texttt{PC}_{01..00} \leftarrow \texttt{0}$
$\texttt{SR[EX\_BITS]}_{31} \leftarrow \texttt{1'b1}$

ERET

Return from System Interrupt

	1	5	5	5	5	5	6
4	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	ERET 0 0 0 0 1 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

$\texttt{PC}_{31..02} \leftarrow \texttt{SR[EX\_PC]}_{31..02}$
$\texttt{PC}_{01..00} \leftarrow \texttt{0}$
$\texttt{SR[EX\_BITS]}_{30} \leftarrow \texttt{1'b1}$

IHDR Create Internal Header

	1	5	5	5	16
1	0	IHDR 1 1 1 0 1	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

```
[rt]$_{31..29}$ ← uimm16$_{15..13}$
[rt]$_{28..24}$ ← uimm16$_{12..8}$
[rt]$_{23..20}$ ← uimm16$_{7..4}$
[rt]$_{19..15}$ ← DN_YPOS$_{4..0}$
[rt]$_{14..10}$ ← DN_XPOS$_{4..0}$
[rt]$_{09..05}$ ← ([rs]$_{4..0}$ & GDN_XMASK$_{4..0}$) + GDN_XADJ$_{4..0}$
[rt]$_{04..00}$ ← (([rs]$_{11..0}$ >> GDN_YSHIFT$_{2..0}$)
                   & GDN_YMASK$_{4..0}$)
                  + GDN_YADJ$_{4..0}$
```

3	5	4	5	5	5	5
fbits	length	user	src Y	src X	dest Y	dest X
31 29	28 24	23 20	19 15	14 10	9 5	4 0

ILW Instruction Load Word

	1	5	5	5	16
5 2	s	ILW 0 0 0 0 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea$_{31..2}$ ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..2}$
ea$_{1..0}$  ← 0
[rt]       ← (*proc-imem-load* ea)
```

The additional cycle of occupancy is a cycle stolen from the fetch unit on access.

INTOFF Disable System Interrupts

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	INTOFF 0 0 0 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
SR[EX_BITS]$_{30}$ ← 1'b0
```

INTON Enable System Interrupts

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	INTON 0 0 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
SR[EX_BITS]$_{30}$ ← 1'b1
```

ISW Instruction Store Word

	1	5	5	5	16
2	0	ISW 0 1 0 0 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea$_{31..2}$  ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..2}$
ea$_{1..0}$   ← 0
(*proc-imem-store* ea [rt])
```

Steals one fetch cycle from compute processor fetch unit.

MFEC Move From Event Counter

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	rs	0 0 0 0 0	rd	0 0 0 0 0	MFEC 0 1 0 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
[rd]$_{31..00}$ ← EC[rs]
```

Note: MFEC captures its value in the RF stage. This is because the event counters are located physically quite distant from the bypass paths of the processor, so the address is transmitted in RF, and the output given in EXE. For example,

```
lw $0,4($0)              # cache miss in TV stage, pipeline frozen
nop                      # occupies TL stage
mfec $4, EC_CACHE_MISS   # EXE stage -- will not register cache miss
mfec $4, EC_CACHE_MISS   # RF        -- will register cache miss
```

Additionally, there is one cycle of lag between when the event actually occurs and when the event counter is updated. For example, assuming no outside stalls like cache misses or interrupts,

```
mtec EC_xxx, $4          # write an event counter
mfec $5, EC_xxx          # reads old value
mfec $5, EC_xxx          # reads new value $
```

MFSR Move From Status / Control Register

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	rs	0 0 0 0 0	rd	0 0 0 0 0	MFSR 0 1 0 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
[rd]$_{31..00}$ ← SR[rs]
```

Section 1.4 describes the status registers.

MLK MDN Lock **RawH**

	1	5	5	5	5	5	6
v	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	uimm5	0 0 0 0 1	MLK 0 0 0 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
SR[EX_BITS]$_{30}$ ← 1'b0;
(*icache-prefetch* PC uimm5)
```

Signals to hardware or software caching system that the following `uimm5` cache lines needs to be resident in the instruction cache for correct execution to occur. Disables interrupts. This allows instruction sequences to access the memory network without concern that the i-caching system will also access it.

MUNLK MDN Unlock **RawH**

	1	5	5	5	5	5	6
v	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 1	MUNLK 0 0 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
SR[EX_BITS]$_{30}$ ← 1'b1
```

Marks end of MDN-locked region. Enables interrupts.

MTEC Move To Event Counter

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	rs	rt	0 0 0 0 0	0 0 0 0 0	MTEC 0 1 0 0 1 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
EC[rt] ← [rs]$_{31..00}$
```

MTSR Move To Status / Control Register

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	rs	rt	0 0 0 0 0	0 0 0 0 0	MTSR 0 1 0 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
SR[rt] $\leftarrow$ [rs]$_{31..00}$
```

Section 1.4 describes the status registers. Note that not all status register bits are fully writable, so some bits may not be updated as a result of an MTSR instruction.

MTSRi Move to Status / Control Immediate

	1	5	5	5	16
1	0	MTSRi 1 1 1 0 0	0 0 0 0 0	rt	uimm16
	31	30 26	25 21	20 16	15 0

```
SR[rt]$_{31..16}$ $\leftarrow$ 0;
SR[rt]$_{15..00}$ $\leftarrow$ uimm16;
```

Section 1.4 describes the status registers. Note that not all status register bits are fully writable, so some bits may not be updated as a result of an MTSR instruction.

OHDR Create Outside Header

	1	5	5	5	16
1	0	OHDR 0 1 1 1 0	rs	rt	uimm16
	31	30 26	25 21	20 16	15 0

```
horiz ← [rs]$_{31}$
side  ← [rs]$_{30}$ & MDN_EXTEND
bits  ← MDN_EXTEND ? [rs]$_{29..25}$ : [rs]$_{30..26}$

[rt]$_{31..29}$ ← side ? 0
                  : (horiz ? MDN_YMAX : MDN_XMAX);
[rt]$_{28..24}$ ← uimm16$_{12..8}$
[rt]$_{23..20}$ ← uimm16$_{7..4}$
[rt]$_{19..15}$ ← DN_YPOS$_{4..0}$
[rt]$_{14..10}$ ← DN_XPOS$_{4..0}$
[rt]$_{09..05}$ ← horiz ? (side ? 0 : MDN_YMAX)
                   : (bits >> MDN_YSHIFT)
[rt]$_{04..00}$ ← horiz ? (bits >> MDN_XSHIFT)
                   : (side ? 0 : MDN_XMAX)
```

3	5	4	5	5	5	5
fbits	length	user	src Y	src X	dest Y	dest X
31 29	28 24	23 20	19 15	14 10	9 5	4 0

OHDR takes an address and an immediate field and produces a header suitable for injecting into the MDN. The immediate field specifies the user and length fields of the message header. OHDR maps the address to an I/O port, which effectively wraps the address space around the periphery of the chip. Raw's hardware data cache uses a private copy of this logic to implement Raw's *memory hash function*.

OHDRX Create Outside Header; Disable System Interrupts

	1	5	5	5	16
1	0	OHDRX 0 1 1 1 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
horiz ← $[rs]_{31}$
side  ← $[rs]_{30}$ & MDN_EXTEND
bits  ← MDN_EXTEND ? $[rs]_{29..25}$ : $[rs]_{30..26}$

$SR[EX\_BITS]_{30}$ ← 1'b0

$[rt]_{31..29}$ ← side ? 0
                    : (horiz ? MDN_YMAX : MDN_XMAX);
$[rt]_{28..24}$ ← $uimm16_{12..8}$
$[rt]_{23..20}$ ← $uimm16_{7..4}$
$[rt]_{19..15}$ ← $DN\_YPOS_{4..0}$
$[rt]_{14..10}$ ← $DN\_XPOS_{4..0}$
$[rt]_{09..05}$ ← horiz ? (side ? 0 : MDN_YMAX)
                    : (bits >> MDN_YSHIFT)
$[rt]_{04..00}$ ← horiz ? (bits >> MDN_XSHIFT)
                    : (side ? 0 : MDN_XMAX)
```

3	5	4	5	5	5	5
fbits	length	user	src Y	src X	dest Y	dest X
31 29	28 24	23 20	19 15	14 10	9 5	4 0

OHDRX takes an address and an immediate field and produces a header suitable for injecting into the MDN. The immediate field specifies the user and length fields of the message header. OHDRX maps the address to an I/O port, which effectively wraps the address space around the periphery of the chip. Since the MDN must be accessed with interrupts disabled, OHDRX provides a cheap way of doing this. Raw's hardware data cache uses a private copy of this logic to implement Raw's *memory hash function*.

PWRBLK Power Block

	1	5	5	5	5	5	6
1+	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	rd	0 0 0 0 0	PWRBLK 1 0 0 0 0 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
[rd]$_{31}$       ← data available in cgni
[rd]$_{30..13}$   ← 0;
[rd]$_{12}$       ← data available in cNi
[rd]$_{11}$       ← data available in cEi
[rd]$_{10}$       ← data available in cSi
[rd]$_{9}$        ← data available in cWi
[rd]$_{8}$        ← data available in csti
[rd]$_{7}$        ← data available in cNi2
[rd]$_{6}$        ← data available in cEi2
[rd]$_{5}$        ← data available in cSi2
[rd]$_{4}$        ← data available in cWi2
[rd]$_{3}$        ← data available in csti2
[rd]$_{2}$        ← data available in cmni
[rd]$_{1}$        ← timer interrupt went off
[rd]$_{0}$        ← external interrupt went off
```

Stalls in RF stage until output is non-zero.

SWLW Switch Load Word

	1	5	5	5	16
5 1	s	SWLW 0 0 1 0 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea$_{31..2}$  ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..2}$
ea$_{1..0}$   ← 0
[rt]          ← (*static-router-imem-load* ea)
```

Steals one fetch cycle from static router.

SWSW Switch Store Word

	1	5	5	5	16
1	0	SWSW 0 1 1 0 1	base	rt	soffs16
	31	30 26	25 21	20 16	15 0

```
ea$_{31..2}$  ← { [base] + (*sign-extend-16-to-32* soffs16) }$_{31..2}$
ea$_{1..0}$   ← 0
(*static-router-imem-store* ea [rt])
```

Steals one fetch cycle from static router.

UINTOFF Disable User Interrupts

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	UINTOFF 0 0 0 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

SR[EX_BITS]$_{31}$ ← 1’b0;

UINTON Enable User Interrupts

	1	5	5	5	5	5	6
1	0	COMM 0 1 0 1 1	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	UINTON 0 0 1 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

SR[EX_BITS]$_{31}$ ← 1’b1;

1.1.3 Cache Management in Raw and RawH

Because Raw's memory model is shared memory but not hardware cache-coherent, effective and fast software cache management instructions are essential. One tile may modify a data structure through its caching system, and then want to make it available to a consuming tile or I/O device. To accomplish this in a cache-coherent way, the sender tile must explicitly flush and/or invalidate the data, and then send an MDN Relay message that bounces off the relevant DRAM I/O Port (indicating that all of the memory accesses have reached the DRAM) to the consumer. The consumer then knows that the DRAM has been updated with the correct values.

To provide effective cache management, there are two series of cache management instructions. Both series allow cache lines to be flushed and/or invalidated. The first series, `ainv`, `afl`, and `aflinv`, takes as input a data address. This address, if it is resident in the cache, is translated into a <set, line> which is used to identify the physical cache line. The second series of instructions, `tagsw`, `taglv`, `tagla`, and `tagfl`, takes a <set, line> pair directly.

The address-based instructions are most effective when the range of addresses residing in the cache is relatively small. If $|A|$ is the size of the address range that needs to be flushed, this series can flush the range in time $\theta(|A|)$.

The tag-based instructions are most effective when the processor needs to invalidate or flush large ranges of address space that exceed the cache size. In this case, the address range can be manipulated faster by using the tag-based instructions to scan the tags of the cache and selectively invalidate and/or flush the contents. In this case, the operations can occur in $\theta(|C|)$, where $|C|$ is the size of the cache. The `tagla` and `taglv` operations allow the cache line tags to be inspected, `tagfl` can be used to flush the contents, and `tagsw` can be used to rewrite (or zero) the tags. Of course, the `tag`xxx series of instructions can accomplish more than simply flushing or invalidating. They provide an easy way to manipulate the cache state directly for verification purposes and boot-time cache initialization.

AINV

Address Invalidate

	1	5	5	5	5	5	6
c	0	COMM 01011	rs	00000	00000	SAAAA	AINV 011110
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
   TAGS[(cache-get-tag ea)].valid ← 0     # stall 4 cycles
```

AFL

Address Flush

	1	5	5	5	5	5	6
c	0	COMM 01011	rs	00000	00000	SAAAA	AFL 011100
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
{
  <set,line> ← (cache-get-tag ea)
  TAGS[<set,line>].mru ← !set

  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     (cache-copy-back <set,line>)    # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

AFLINV

Address Flush and Invalidate

	1	5	5	5	5	5	6
c	0	COMM 01011	rs	00000	00000	SAAAA	AFLINV 011101
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
ea ← [rs] + (S << 14) + (AAAA << kDataCacheLineSize)

if (cache-contains ea)
{
  <set,line> ← (cache-get-tag ea)
  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     TAGS[<set,line>].valid ← 0
     (cache-copy-back ea)    # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

TAGFL Tag Flush

	1	5	5	5	5	5	6
c	0	COMM 0 1 0 1 1	rs	0 0 0 0 0	0 0 0 0 0	S A A A A	TAGFL 0 1 1 0 0 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
set         ← [rs]$_{14}$ ^ S
line$_{8..0}$   ← [rs]$_{13..5}$ + AAAA

if (TAGS[<set,line>].valid)
{
  TAGS[<set,line>].mru ← !set
  if (TAGS[<set,line>].dirty)
  {
     TAGS[<set,line>].dirty ← 0
     (*cache-copy-back* <set,line>)     # stall >= 13 cycles
  }
  else
     # stall 5 cycles
}
```

TAGLA Tag Load Address

	1	5	5	5	5	5	6
3 1	0	COMM 0 1 0 1 1	rs	0 0 0 0 0	rd	S A A A A	TAGLA 0 1 1 0 1 0
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
set         ← [rs]$_{14}$ ^ S
line$_{8..0}$   ← [rs]$_{13..5}$ + AAAA

[rd] ← { TAGS[<set,line>].addr$_{17..00}$ line$_{8..0}$ [rs]$_{4..0}$ }
```

TAGLV Tag Load Valid

	1	5	5	5	5	5	6
3 1	0	COMM 0 1 0 1 1	rs	0 0 0 0 0	rd	S A A A A	TAGLV 0 1 1 0 1 1
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
set       ← [rs]$_{14}$ ^ S
line$_{8..0}$ ← [rs]$_{13..5}$ + AAAA

[rd] ← TAGS[<set,line>].valid
```

TAGSW Tag Store Word

	1	5	5	5	5	5	6
1	0	COMM 01011	rs	rt	00000	SAAAA	TAGSW 011000
	31	30 26	25 21	20 16	15 11	10 6	5 0

```
set          ← $[rs]_{14}$ ^ S
$line_{8..0}$ ← $[rs]_{13..5}$ + AAAA

TAGS[<set,line>].$valid_{0}$      ← $[rt]_{18}$
TAGS[<set,line>].$addr_{17..00}$  ← $[rt]_{17..00}$
TAGS[<set,line>].$dirty_{0}$      ← 0
```

Should not be issued the cycle after a load or store instruction because of write-after-write hazards on the tag memory.

1.2 Semantic Helper Functions

This section gives the semantics of the helper-functions used in the previous section. This thesis uses little-endian bit-ordering exclusively.

$\mathtt{w}_{x..y}$ ⟶ Bits x..y, inclusive, of w.
If (x < y), the empty string.

$\{\ \mathtt{w}\ \mathtt{z}\ \}$ ⟶ Concatenate the bits of w and z together.
w will occupy the more significant bits.

$\mathtt{z}^{n}$ ⟶ Concatenate n copies of z together.

(*sign-extend-16-to-32* simm16) ⟶ $\{\ (\mathtt{simm16}_{15})^{16}\ \mathtt{simm16}_{15..00}\ \}$
(*sign-extend-26-to-30* simm26) ⟶ $\{\ (\mathtt{simm26}_{25})^{4}\ \mathtt{simm26}_{25..00}\ \}$
(*sign-extend-16-to-30* simm16) ⟶ $\{\ (\mathtt{simm16}_{15})^{14}\ \mathtt{simm16}_{15..00}\ \}$
(*zero-extend-16-to-32* uimm16) ⟶ $\{\ 0^{15..0}\ \mathtt{uimm16}_{15..00}\ \}$

(*left-rotate* uimm32 ra) ⟶ $\{\ \mathtt{uimm32}_{(31-ra)..0}\ \mathtt{uimm32}_{31..(32-ra)}\ \}$
(*right-rotate* uimm32 ra) ⟶ $\{\ \mathtt{uimm32}_{(ra-1)..0}\ \mathtt{uimm32}_{31..ra}\ \}$

(*cache-contains* addr) ⟶ Returns 1 if valid cache line corresponding to addr
is in cache, otherwise 0.

(*cache-get-tag* addr) ⟶ Returns <set,line> pair corresponding to addr
in cache.

(*cache-copy-back* tagid) ⟶ Sends update message containing data
corresponding to tagid to owner DRAM.

(*cache-read-byte* addr) ⟶ Ensure cache line corresponding to addr
is in cache, return byte at addr.

(*cache-read-half-word* addr) ⟶ Ensure cache line corresponding to addr
is in cache, return half-word at $\{\ \mathtt{addr}_{31..1}\ 0^{1}\ \}$.

(*cache-read-word* addr) ⟶ Ensure cache line corresponding to addr
is in cache, return word at $\{\ \mathtt{addr}_{31..2}\ 0^{2}\ \}$.

(*cache-write-byte* addr val) ⟶ Ensure cache line corresponding to addr
is in cache, write $\mathtt{val}_{7..0}$ to addr.

(*cache-write-half-word* addr val) ⟶ Ensure cache line corresponding to addr
is in cache, write $\mathtt{val}_{15..0}$ to $\{\ \mathtt{addr}_{31..1}\ 0^{1}\ \}$.

(*cache-write-word* addr val) ⟶ Ensure cache line corresponding to addr
is in cache, write $\mathtt{val}_{31..0}$ to $\{\ \mathtt{addr}_{31..2}\ 0^{2}\ \}$.

```
(create-mask mb me z)   ⟶   if (z)
                                 if ($me_{1..0}$ == 0b00) { $mb_{4..0}$ $me_{4..2}$ }$^4$
                                 if ($me_{1..0}$ == 0b11)
                                    { $mb_4^4$ $mb_3^4$ $mb_2^4$ $mb_1^4$ $mb_0^4$ $me_4^4$ $me_3^4$ $me_2^4$ }
                              else
                                 if (mb $<=_{unsigned}$ me)
                                    { $0^{31..(me+1)}$ $1^{me..mb}$ $0^{(mb-1)..0}$ }
                                 else
```

$$\{\ 1^{31..(mb+1)}\ \ 0^{mb..me}\ \ 1^{(me-1)..0}\ \}$$

The last line was a specification bug as it does not generate every mask with a single zero. A better version is:

$$\{\ 1^{31..(mb+1)}\ \ 0^{mb..me+1}\ \ 1^{(me)..0}\ \}$$

(*icache-prefetch* `addr lines`) ⟶ Ensure `lines` instruction cache lines following cache line containing `addr` are resident in instruction cache.

(*static-router-imem-store* `addr data`) ⟶ Writes 32-bit value `data` into static router instruction cache at location `addr`.

(*static-router-imem-load* `addr data`) ⟶ Loads 32-bit value `data` from static router instruction cache at location `addr`.

(*proc-imem-store* `addr data`) ⟶ Writes 32-bit value `data` into static router instruction cache at location `addr`.

(*proc-imem-load* `addr data`) ⟶ Loads 32-bit value `data` from static router instruction cache at location `addr`.

1.3 Opcode Maps

Below are opcode maps which document the allocation of instruction encoding space.

1.3.1 High-Level (“Opcode”) Map

(Instructions with bits 31..29 set to 1 are predicted taken.) Bang instructions all have bit 31 set. however RLM, RLMI and RLVM use bit 28 to indicate bang.

bits	bits 28..26							
31..29	000	001	010	011	100	101	110	111
111	**REGIMM+**	BNEA+	BNE+	BEQ+	BL	BLAL	JNEL+	JEQL+
110	LB!	LBU!	LH!	LHU!	LW!	SLTIU!	SLTI!	ADDIU!
101	RLM	RLMI	RLVM		RLM!	RLMI!	RLVM!	
100	**SPECIAL!**	ILW!	ORI!	XORI!	ANDI!	SWLW!	**FPU!**	AUI!
011	**REGIMM-**	BNEA-	BNE-	BEQ-	MTSRI	IHDR	JNEL-	JEQL-
010	LB	LBU	LH	LHU	LW	SLTIU	SLTI	ADDIU
001	SB	ISW	SH	**COM**	SW	SWSW	OHDR	OHDRX
000	**SPECIAL**	ILW	ORI	XORI	ANDI	SWLW	**FPU**	AUI

1.3.2 SPECIAL Submap

(Applies when bits 31..26 are SPECIAL or SPECIAL!)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	SLL	MAGIC	SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR	JRHON	JRHOFF				
010	MFHI	MTHI	MFLO	MTLO	MFFD	MTFD		
011	MULLO	MULLU	DIV	DIVU				
100		ADDU		SUBU	AND	OR	XOR	NOR
101	MULHI	MULHU	SLT	SLTU				
110								
111	POPC	CLZ						

1.3.3 FPU Submap

(Applies when bits 31..26 are FPU or FPU!)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	ADD.s	SUB.s	MUL.s	DIV.s		ABS.s		NEG.s
001						TRUNC.s		
010								
011								
100	CVT.s				CVT.w			
101								
110	C.F	C.UN	C.EQ	C.UEQ	C.OLT	C.ULT	C.OLE	C.ULE
111	C.SF	C.NGLE	C.SEQ	C.NGL	C.LT	C.NGE	C.LE	C.NGT

1.3.4 COM Submap

(Applies when bits 31..26 are COM)

bits	bits 2..0							
5..3	000	001	010	011	100	101	110	111
000	DRET	INTOFF	UINTOFF	ERET				
001		INTON	UINTON					
010	MFSR	MTSR	MFEC	MTEC				
011	TAGSW	TAGFL	TAGLA	TAGLV	AFL	AFLINV	AINV	
100	PWRBLK							
101								
110								
111								

1.3.5 REGIMM Submap

(Applies when bits 31..26 are REGIMM+ or REGIMM-.) Bit 20 indicates a link instruction, and bit 18 indicates an absolute jump. The conditions are mirrored across these axes when appropriate.

bits	bits 18..16							
20..19	000	001	010	011	100	101	110	111
00	BLTZ	BLEZ	BGEZ	BGTZ				
01					J			
10	BLTZAL		BGEZAL		JLTZL	JLEZL	JGEZL	JGTZL
11					JAL			

1.4 Status and Control Registers

#	Status Reg Name	R/W	Purpose
0	SW_FREEZE	RW	Switch Processor is Frozen. [00] ← (1 Frozen) (0 Running)
1	SW_BUF1	R	# of elements in static router crossbar 1 NIBs [22:20] number of elements in c21 (≤ 4) [19:17] number of elements in cNi (≤ 4) [16:14] number of elements in cEi (≤ 4) [13:11] number of elements in cSi (≤ 4) [10:08] number of elements in cWi (≤ 4) [07:05] number of elements in csti (≤ 4) [04:00] number of elements in csto (≤ 8)
2	SW_BUF2	R	# of elements in static router crossbar 2 NIBs [22:20] number of elements in c12 (≤ 4) [19:17] number of elements in cNi_2 (≤ 4) [16:14] number of elements in cEi_2 (≤ 4) [13:11] number of elements in cSi_2 (≤ 4) [10:08] number of elements in cWi_2 (≤ 4) [07:05] number of elements in csti_2 (≤ 4) [04:00] number of elements in csto (≤ 8)
3	MDN_BUF	R	# of elements in MDN router NIBs [19:17] number of elements in cNi (≤ 4) [16:14] number of elements in cEi (≤ 4) [13:11] number of elements in cSi (≤ 4) [10:08] number of elements in cWi (≤ 4) [07:05] number of elements in cmni (≤ 4) [04:00] number of elements in cmno (≤ 16)
4	SW_PC	RW	Current PC of switch processor. Byte address aligned to eight-byte boundaries. Used primarily for context switching. Generally, writing to this register is used for context-switching purposes. It should only be performed when the switch is FROZEN or if the compute processor program knows absolutely that the switch is stalled at a known PC. Otherwise, the program can no longer assume the static ordering of operands on the SON. Writing to this register causes a branch misprediction in the switch. Allow at least three cycles for corresponding instruction to be executed.
5	BR_INCR	RW	Signed 32-bit increment value for BNEA instruction. Caller-saved.

#	Status Reg Name		Purpose
6	EC_DYN_CFG	RW	Configuration for Event Counting of Dynamic Network events [30:28] Memory Network North (D=N) Configuration [27:25] Memory Network East (D=E) Configuration [24:22] Memory Network South (D=S) Configuration [21:19] Memory Network West (D=W) Configuration [18:16] Memory Network Proc (D=P) Configuration [14:12] Memory Network North (D=N) Configuration [11:09] Memory Network East (D=E) Configuration [08:06] Memory Network South (D=S) Configuration [05:03] Memory Network West (D=W) Configuration [02:00] Memory Network Proc (D=P) Configuration
		Settings:	
		0	# of cycles output port D wants to transmit but could not because neighbor tile's input buffer is full.
		1	# of words transmitted from input port D to output port P
		2	# of words transmitted from input port D to output port W
		3	# of words transmitted from input port D to output port S
		4	# of words transmitted from input port D to output port E
		5	# of words transmitted from input port D to output port N
		6	# of words transmitted from input port D
		7	# cycles input port D had data to transmit but was not able to
7	WATCH_VAL	RW	[31:00] 32-bit timer; increments each cycle
8	WATCH_MAX	RW	[31:00] value to fire timer interrupt and then zero WATCH_VAL
9	WATCH_SET	RW	[00] zero WATCH_VAL if cgno is empty or a value was dequeued [01] zero WATCH_VAL if processor issues an instruction
10	CYCLE_HI	RW	[31:00] high 32-bits of cycle counter
11	CYCLE_LO	RW	[31:00] low 32-bits of cycle counter Note: To read the cycle counter efficiently, read CYCLE_HI, then CYCLE_LO, then subtract one from CYCLE_LO. Cycle counters are writable to make tests reproducible.
12	EVENT_CFG2	RW	[24:0] configures the set of events that causes c_trigger event counters to be incremented. See 1.5.
13	GDN_RF_VAL	RW	[31:00] GDN refill value When EX_MASK[GDN_REFILL] is enabled, a read from `$cgno` will return GDN_RF_VAL, signal an interrupt by setting EX_BITS[GDN_REFILL], and leave cgno unchanged. This allows cgno to be virtualized, e.g. for context switches and deadlock recovery.

#	Status Reg Name		Purpose
14	GDN_REMAIN	RW	[04:00] Number of words remaining to be sent to complete current message on cgno. GDN_COMPLETE interrupt fires when value transitions to zero. OS typically initializes this with GDN_PENDING value to allow GDN messages to complete when context switching.
15	EX_BASE_ADDR	RW	[31:00] Pointer to beginning of exception vector table. Set to zero at boot time. Applies to **RawH**.
16	GDN_BUF	R	# of elements in GDN router NIBs [24:20] GDN_PENDING number of elements (≤ 31) that need to be sent to cgno from processor pipeline to complete current message. Note this count does not include those instructions currently in the pipeline; the operating system should flush the pipeline before reading this value. The OS loads this value into the GDN_REMAIN SPR for the GDN_PENDING interrupt to trigger on. [19:17] number of elements (≤ 4) in cNi [16:14] number of elements (≤ 4) in cEi [13:11] number of elements (≤ 4) in cSi [10:08] number of elements (≤ 4) in cWi [07:05] number of elements (≤ 4) in cgni [04:00] number of elements (≤ 16) in cgno
17	GDN_CFG	RW	General Dynamic Network Configuration [31:27] GDN_XMASK - Masks X bits from an address [26:22] GDN_YMASK - Masks Y bits from an address [21:17] GDN_XADJ - Adjusts from local to global X address [16:12] GDN_YADJ - Adjusts from local to global Y address [11:09] GDN_YSHIFT - Gets Y bits from an address See `IHDR` instruction.

#	Status Reg Name		Purpose
18	STORE_METER	RW	STORE_ACK counters [31:27] PARTNER_Y - Y location of partner port [26:22] PARTNER_X - X location of partner port [21] ENABLE - enable store meter-based stalls [10] DECREMENT_MODE (see below; reads always zero) [9:5] COUNT_PARTNER - # of partner accesses left [4:0] COUNT_NON_PARTNER - # of non-partner accesses left Since the counts are updated as STORE_ACK messages are received over the MDN, care must be taken to update STORE_METER in a way that avoids race conditions. Ordinarily, the only way to do this is to modify the register only when all store-acks have been received. Alternatively, the user may write to the register with DECREMENT_MODE set; in this case the COUNT_NON_PARTNER will be decremented if bit 0 is set, and COUNT_PARTNER will be decremented if bit 5 is set. No other bits are changed. This handles the case where the user is directly transmitting memory packets over the MDN using explicit accesses to cmno, and needs to update the the STORE_ACK counters to reflect this.
19	MDN_CFG	RW	Memory Dynamic Network Configuration [31:27] DN_XPOS - Absolute X position of tile in array [26:22] DN_YPOS - Absolute Y position of tile in array [21:17] MDN_XMAX - X Coord of East-Most Tiles [16:12] MDN_YMAX - Y Coord of South-Most Tiles [11:09] MDN_XSHIFT - Shift Amount X [08:06] MDN_YSHIFT - Shift Amount Y [00:00] MDN_EXTEND - Use all four edge of chip. These SPRs are used to determine Raw's *memory hash function* as described in MBT's PhD thesis. This function determines where the data caches send their messages for cache fills and evictions. It also determines the functionality of the OHDR and OHDRX instructions.
20	EX_PC	RW	PC where system-level exception occurred.
21	EX_UPC	RW	PC where user-level exception occurred. (GDN_AVAIL is the only user-level exception)

#	Status Reg Name		Purpose
22	FPSR	RW	Floating Point Status Register [5] Unimplemented [4] Invalid [3] Divide by Zero [2] Overflow [1] Underflow [0] Inexact operation These bits are sticky; i.e. floating point operations can set but cannot clear these bits. However, the user can freely change the bits via `MTSR` or `MFSR`. These flags are set the cycle after the floating point instruction finishes execution; i.e., you need three nops inbetween the last floating point operation and a `MFSR` to read the correct value.
23	EVENT_BITS	R	[15:0] the list of events that have triggered
24	EX_BITS	R	Interrupt Status [31] USER - all user interrupts masked if 0 [30] SYSTEM - all interrupts masked if 0 The above can be set/cleared using `inton`, `intoff`, `uinton`, `uintoff`. [6] EVENT_COUNTER [5] GDN_AVAIL [4] TIMER [3] EXTERNAL [2] TRACE [1] GDN_COMPLETE [0] GDN_REFILL For bits 0..6, a "1" indicates a request for a given interrupt occurred but that it has not yet been serviced.
25	EX_MASK	RW	Interrupt Mask [6] EVENT_COUNTER [5] GDN_AVAIL [4] TIMER [3] EXTERNAL [2] TRACE [1] GDN_COMPLETE [0] GDN_REFILL A "0" indicates that the exception is suppressed.

#	Status Reg Name		Purpose
26	EVENT_CFG	RW	Event Counter Configuration [31:16] Enables for events 16..0 [15:01] PC to profile (omit low two bits) for single mode [00] ← (1 Single Instruction Mode) (0 Global Instruction Mode)
27	POWER_CFG	RW	Power Saving Configuration [00] Disable comparator toggle-suppression [01] Disable ALU toggle-suppression [02] Disable FPU toggle-suppression [03] Disable Multiplier toggle-suppression [04] Disable Divider toggle-suppression [05] Disable Data Cache toggle-suppression [06] Enable Instruction Memory power saving [07] Enable Data Memory power saving [08] Enable Static Router Memory power saving [09] Disable `pwrblk` wake up after TIMER interrupt [10] Disable `pwrblk` wake up after EXTERNAL interrupt [11] Timer wakeup pending on return to `pwrblk` [12] External wakeup pending on return to `pwrblk` At reset, POWER_CFG is set to zero. Bits 11 and 12 are set by the processor if the corresponding interrupt is taken while waiting on a `pwrblk`.
28	TN_CFG	W	Test Network Configuration
29	TN_DONE	W	Signal “DONE” on Test Network with value [31:0]
30	TN_PASS	W	Signal “PASS” on Test Network with value [31:0]
31	TN_FAIL	W	Signal “FAIL” on Test Network with value [31:0]

1.5 Event Counting Support

The event counters provide a facility to monitor, profile, and respond to events on a Raw tile. Each tile has a bank of 16 c_trigger modules. Each c_trigger has a 32-bit counter. These counters count down every time a particular event occurs. The EVENT_CFG2 register is used to determine which events each c_trigger responds to. When the counter transitions from 0 to -1, it will assert a line (the "trigger") which will hold steady until the user writes a new value into the counter. These triggers are visible in the EVENT_BITS register, and are OR'd together to form the EX_BITS EVENT_COUNTER bit, which can cause an interrupt. When the trigger is asserted, the c_trigger module latches the PC (without the low zero bits) of the instruction that caused the event into bits [31:16] of the counter (the `rlm` instruction can be used to extract them efficiently). The c_trigger module will continue to count down regardless of the setting of the trigger. Because the PC is stored in the high bits, there is a window of time in which subsequent events will not corrupt the captured PC. Note that if the event is not instruction related, the setting of the PC in the c_trigger is undefined. The event counters can be both read and written by the user. There is typically a one cycle delay between when an event occurs and when an `mfec` instruction will observe it; there is also a delay of two cycles before an event trigger interrupt will fire.

c_trigger #	EVENT_CFG2	Stage	Function	Notes
0	[25] ← 0	@	Cycle Count	So handler can bound sampling window.
0	[25] ← 1	F	Write Over Read	For poor man's shared memory support. Detects when a resident cache line is marked dirty by a `sw` to an odd address for the first time. Note: If the `sw` is preceded by a `lw` / `sw` / `flush` this mechanism does not have the bandwidth to verify the previous state of the bits. It will conservatively count it as an event.
1		M	Cache Writebacks	Includes flushes.
2		M	Cache Fills	
3		M	Cache Stall Cycles	Total number of cycles that the backend of the pipeline is frozen by the cache state machine. Includes write-back and fill time, as well as time stolen by non-dirty `flush` instructions.
4	[0] ← 0	E	Cache Miss Ops	Number of `flush`, `lw`, `sw` instructions issued.
4	[0] ← 1	E	FPU Ops	Number of FPU instructions issued. Includes `.s` and `.w` instructions.

c_trigger #	EVENT_CFG2	Stage	Function	Notes
5	[1] ← 0	E	Possible Mispredicts	Conditional Jumps and Branches, ERET, DRET, JR, JALR.
5	[1] ← 1	E	Possible Mispredicts	Possible mispredicts due to wrong SBIT (i.e., only conditional jumps and branches)
6	[2] ← 0	E	Actual Mispredicts	Branch mispredictions.
6	[2] ← 1	E	Actual Mispredicts	Mispredictions due to wrong SBIT
7		@	Switch Stalls	On static router (Trigger captures static router PC)
8		@	Possible Mispredicts	On static router (Trigger captures static router PC)
9		@	Actual Mispredicts	On static router (Trigger captures static router PC)
10		@	Pseudo Random LFSR	X_next = (X >> 1) | (xor(X[31,30,10,0]) << 31) Note: Sampling this more than once per 32 cycles produces highly correlated numbers.
11	[3]	R	Functional Unit Stalls	Stalls due to bypassing (e.g., the output of a preceding instruction is not available yet) or because of interlocks on the fp/int dividers.
11	[4]	@	GP	GDN Processor Port Counting
11	[5]	@	MP	MDN Processor Port Counting
11	[23]	@	Instructions Issued	# of instructions that enter Execute stage.
12	[6]	R	Non-cache stalls	# of stalls not due to cache misses. Includes `ilw`/`isw`; if trigger fires on isw/ilw PC will be the PC of the instruction in the RF stage, rather than the `ilw`/`isw` instruction.
12	[7]	@	GW	GDN West Port Counting
12	[8]	@	MW	MDN West Port Counting
13	[9]	R	`ilw`/`isw`	# of `ilw`/`isw` instructions issued.
13	[10]	@	GS	GDN South Port Counting
13	[11]	@	MS	MDN South Port Counting
13	[24]	@	Instructions Issued	# of instructions that enter Execute stage.
14	[12]	R	`$csto` stalls	Instruction issue blocked on `$csto` full
14	[13]	R	`$cgno` stalls	Instruction issue blocked on `$cgno` full
14	[14]	R	`$cmno` stalls	Instruction issue blocked on `$cmno` full
14	[15]	@	GE	GDN East Port Counting
14	[16]	@	ME	MDN East Port Counting
15	[17]	R	`$csti` stalls	Instruction issue blocked on `$csti` empty
15	[18]	R	`$csti2` stalls	Instruction issue blocked on `$csti2` empty
15	[19]	R	`$cgni` stalls	Instruction issue blocked on `$cgni` empty
15	[20]	R	`$cmni` stalls	Instruction issue blocked on `$cmni` empty
15	[15]	@	GN	GDN North Port Counting
15	[16]	@	MN	MDN North Port Counting

The previous table describes the events that the c_trigger modules can be configured to count. The EVENT_CFG2 column specifies the bit number of EVENT_CFG2 that must be set in order to enable counting of that event.

The low bits of EVENT_CFG allow the user to count events that occurs on a particular instruction at a particular PC instead of across all PCs. For this "single instruction mode", EVENT_CFG[0] is set to 1, and the PC to sample is placed into EVENT_CFG[15:1]. In cases where the event does not have an associated main processor PC (marked with the "@" in the table), the EVENT_CFG single instruction mode setting is ignored. The high bits of EVENT_CFG selectively enable counting on a per event basis, but do not suppress existing triggers.

The EVENT_CFG2 SPR allows the user to configure the events that a particular c_trigger module counts. In some cases multiple enabled events may be connected to the same trigger. In that case, the counters increments each cycle if any such enabled events has occurred. In some cases, there are nonsensical combinations that can be enabled (say GE and `$csto` stalls).

The meaning of the GN, GE, GS, GW, GP, MN, ME, MS, MW, and MP events are configured by the EC_DYN_CFG status/control register. Each event corresponds to a network N (G = general, M = memory) and a direction D (N=north, E=east, ...). The encodings are shown in the table in Section 1.4.

1.6 Exception Vectors

#	Name	Offset	Purpose
0	VEC_GDN_REFILL	0x00	Dynamic Refill Exception
1	VEC_GDN_COMPLETE	0x10	GDN Send Is Complete
2	VEC_TRACE	0x20	Trace Interrupt
3	VEC_EXTERN	0x30	External Interrupt (MDN)
4	VEC_TIMER	0x40	Timer Exception
5	VEC_GDN_AVAIL	0x50	Data Avail on GDN
6	VEC_EVENT_COUNTERS	0x60	Event Counter Interrupt

In the Raw architecture, the exception vectors are stored starting at offset zero in instruction memory. In RawH, the exception vectors are stored relative to `SR[EX_BASE_ADDR]`. When an exception occurs, the processor starts fetching from the corresponding exception location. Thus, a TIMER exception would start fetching at address `SR[EX_BASE_ADDR] + 0x40`.

Each exception has 4 contiguous instructions; this is enough to do a small amount of work; such as save a register, load a jump address, and branch there:

```
sw $3, interrupt_save($gp)
lw $3, gdn_vec($gp)
jr $3
```