ECE 511 Instruction Set Architecture Description

September 10, 2004

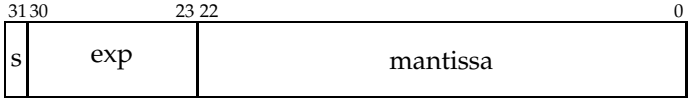
This document describes the assembly programmer's interface to the processor we are using in ECE 511. The architecture is similar, but not identical to, the MIPS R4000. The major differences are

1. Floating point:
   (a) Floating point operations use the same register file as integer operations.
   (b) Floating point compares have a destination register instead of setting a flag.
   (c) The floating point branches, BC1T and BC1F, are removed, since the integer versions have equivalent functionality.
   (d) The only available floating point rounding mode is round-to-nearest even.
2. There are no HI/LO regs. Multiply and divide instructions target GPRs instead of the HI/LO regs.
3. The instruction set and encodings are slightly different. See below for more details.
4. Because of jump instruction encodings, only the low 256Mbytes (64Mwords) of memory is usable to hold instructions.
5. The memory operations are little endian. In other words, if there is a word stored at address P, then the low order byte is stored at address P and the most significant byte is stored at address P+3. For reference, TCP/IP and SPARC are both big endian and x86 is little endian.
6. All multi-cycle operations (branches, loads, multiplies, divides) are fully interlocked. There are no delay slots.

1 Data types

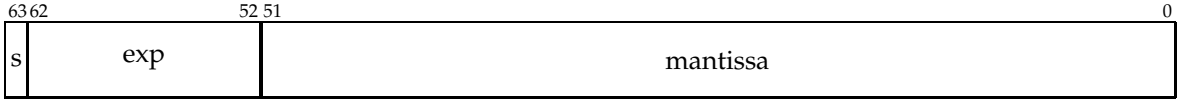
The processor supports four data types, unsigned 32 bit integers, unsigned 64 bit integers, 32 bit IEEE floating point numbers and 64 bit IEEE floating point numbers. When signed integers are required they are represented in 2s-complement.

IEEE single precision floating point format represents data as follows:



In general the number represented by a particular bit pattern is given by: $(-1)^s(1+\frac{\text{mantissa}}{2^{23}})2^{(\text{exp}-127)}$. The floating point number 0 is represented by the bit pattern 0.

IEEE double precision floating point format represents data as follows:



In general the number represented by a particular bit pattern is given by: $(-1)^s(1+\frac{\text{mantissa}}{2^{52}})2^{(\text{exp}-1023)}$. The floating point number 0 is represented by the bit pattern 0.

2 Registers

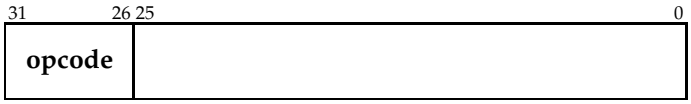
The processor state consists of the 32 bit program counter and 32 general purpose registers (each 64 bits), called, "$0" through "$31." The general purpose registers are used for all integer and floating point operations, there is no separate floating point register file.

Register $0 always contains the value 0. Register $31 is used to save the return value for jump-and-link instructions. Several other registers are managed in particular ways by the compiler:

register	alias	use
$0		Always has value zero.
$1	$at	Reserved for the assembler.
$2 - $3		Expression evaluation and procedure return values.
$4 - $7		Used to pass first 4 words of actual arguments. Not preserved across procedure calls.
$8 - $15		Temporaries. Not preserved across procedure calls.
$16 - $28		Callee saved registers.
$29	$sp	The stack pointer.
$30		A callee saved register, sometimes used by gcc as a frame pointer.
$31	$ra	The link register

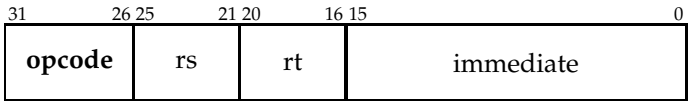
3 Instruction encodings

All instructions are 32 bit. Each instruction starts with a 6 bit primary opcode.



3.1 Primary opcode table

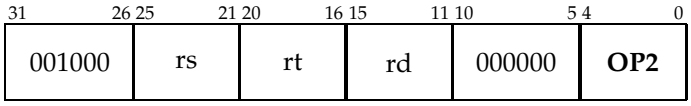
This map is for the first six bits of the instruction (the "opcode" field). It is used for instructions that require 2 registers and a 16 bit immediate field.



		instr[28:26]$_2$							
		000	001	010	011	100	101	110	111
instr [31:29]	000	**MUL**				BEQ	BNE	J	JAL
	001	**ARITH**				BLTZ	BLEZ	BGEZ	BGTZ
	010	**FPU**							
	011								
	100								
	101	DADDIU	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	AUI
	110	LB	LH	LW	LD	LBU	LHU	LWU	
	111	SB	SH	SW	SD				

3.2 ARITH opcode table

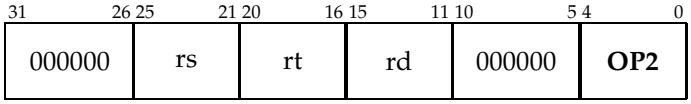
When the opcode field is set to ARITH (001000_2), the ARITH map is used. This map is for the last 5 bits of the instruction (the "op2" field). In general the instructions in this map are integer instructions that require 3 register fields, and no immediates.



		instr[2:0]$_2$							
		000	001	010	011	100	101	110	111
instr [4:3]	00		ADDU		SUBU	AND	OR	XOR	NOR
	01	JR	JALR	SLT	SLTU				
	10		DADDU		DSUBU				
	11							UNIX	HALT

3.3 MUL opcode table

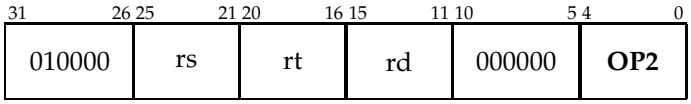
When the opcode field is set to MUL (000000_2), the MUL map is used. This map is for the last 5 bits of the instruction (the "op2" field). In general the instructions in this map are integer instructions that require 3 register fields, and no immediates.



		instr[2:0]$_2$							
		000	001	010	011	100	101	110	111
instr [4:3]	00	SLL		SRL	SRA	SLLV		SRLV	SRAV
	01	MULL		DIV	DIVU	MULH	MULHU		
	10	DSLL		DSRL	DSRA	DSLLV		DSRLV	DSRAV
	11	DMULL		DDIV	DDIVU	DMULH	DMULHU		

3.4 FPU opcode table

FPU instructions, in general, require 3 register operands. They are specified by setting the opcode field to FPU (010000_2). The 5 bit "op2" field (bits 4:0) is used to identify the particular instruction to execute.



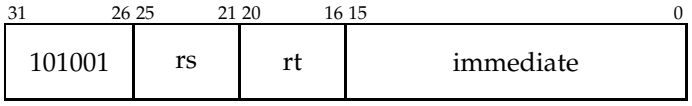
		instr[2:0]$_2$							
		000	001	010	011	100	101	110	111
instr [4:3]	00	ADD.S	SUB.S	MUL.S	DIV.S	C.EQ.S	ABS.S	TRUNC.S	NEG.S
	01		CVT.S.W	CVT.S.D	CVT.W.S			C.LT.S	C.LE.S
	10	ADD.D	SUB.D	MUL.D	DIV.D	C.EQ.D	ABS.D	TRUNC.D	NEG.D
	11		CVT.D.W	CVT.D.S	CVT.W.D			C.LT.D	C.LE.D

4 Instruction descriptions

4.1 Integer operate instructions

4.1.1 ADDIU

Encoding



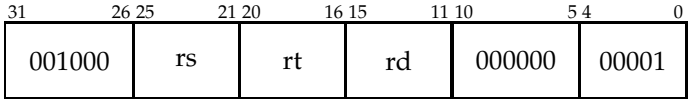
Assembly syntax `addiu $rt, $rs, immediate`

Description 32 bit add immediate unsigned. The immediate field is sign extended to 32 bits, added to the 32 bit value in register rs. The 32 bit result is sign extended to 64 bits and placed in register rt.

RTL $Reg[rt] := sign\_ext(Reg[rs]_{31:0} +_{32} sign\_ext32(imm))$

4.1.2 ADDU

Encoding



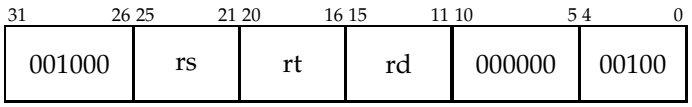
Assembly syntax `addu $rd, $rs, $rt`

Description 32 bit Add unsigned. The 32 bit values in registers rs and rt are added. The 32 bit result is sign extended to 64 bits and placed in register rd.

RTL $Reg[rd] := sign\_ext(Reg[rs]_{31:0} +_{32} Reg[rt]_{31:0})$

4.1.3 AND

Encoding



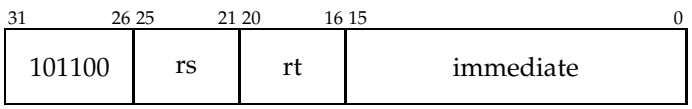
Assembly syntax `and $rd, $rs, $rt`

Description And. The values in registers rs and rt are bitwise anded and the result is placed in register rd.

RTL $Reg[rd] := Reg[rs] \& Reg[rt]$

4.1.4 ANDI

Encoding



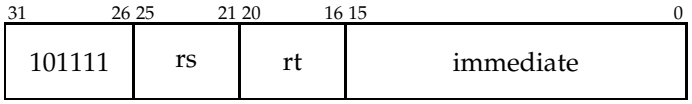
Assembly syntax `andi $rt, $rs, immediate`

Description And immediate unsigned. The immediate field is zero extended, bitwise anded with the value in register rs and the result placed in register rt.

RTL `Reg[rt] := Reg[rs] & zero_ext(imm)`

4.1.5 AUI

Encoding



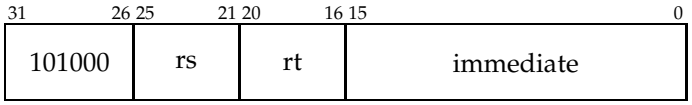
Assembly syntax `aui $rt, $rs, immediate`

Description 32 bit add upper immediate. The immediate field is shifted left 16 bits and added to the 32 bit value in register rs. The 32 bit result is sign extended to 64 bits and placed in register rt.

RTL $\texttt{Reg[rt] := sign\_ext(Reg[rs]}_{31:0} +_{32} \texttt{(imm << 16))}$

4.1.6 DADDIU

Encoding



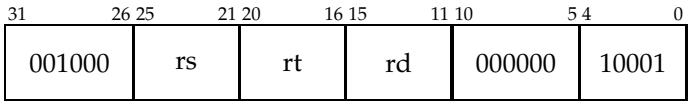
Assembly syntax `daddiu $rt, $rs, immediate`

Description Add immediate unsigned. The immediate field is sign extended to 64 bits, added to the value in register rs and the result is placed in register rt.

RTL `Reg[rt] := Reg[rs] + sign_ext(imm))`

4.1.7 DADDU

Encoding



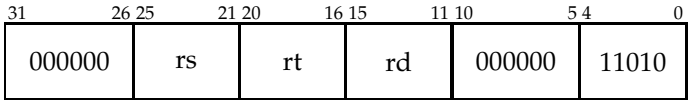
Assembly syntax `daddu $rd, $rs, $rt`

Description Add doubleword unsigned. The values in registers rs and rt are added. The result is placed in register rd.

RTL `Reg[rd] := Reg[rs] + Reg[rt]`

4.1.8 DDIV

Encoding



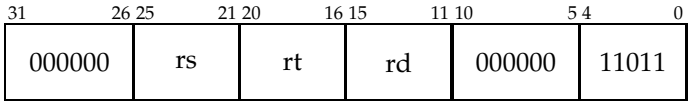
Assembly syntax `ddiv $rd, $rs, $rt`

Description Divide. The signed integer value in register rs is divided by the signed integer value in register rt. The result is placed in register rd.

RTL `Reg[rd] := Reg[rs] /s Reg[rt]`

4.1.9 DDIVU

Encoding



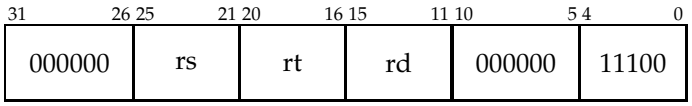
Assembly syntax `ddivu $rd, $rs, $rt`

Description Divide unsigned. The unsigned integer value in register rs is divided by the unsigned integer value in register rt. The result is placed in register rd.

RTL `Reg[rd] := Reg[rs] /u Reg[rt]`

4.1.10 DMULH

Encoding



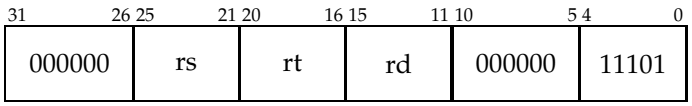
Assembly syntax `dmulh $rd, $rs, $rt`

Description Multiply high. The values in registers rs and rt are multiplied as signed 2s-complement integers to produce a 128 bit result. The high 64 bits of this result is placed in register rd.

RTL $\texttt{Reg[rd] := (Reg[rs] *}_{\texttt{s}}\texttt{ Reg[rt])}_{127:64}$

4.1.11 DMULHU

Encoding



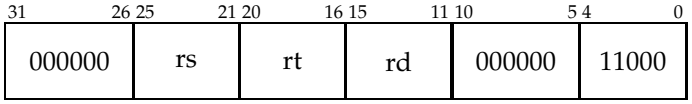
Assembly syntax `dmulhu $rd, $rs, $rt`

Description Multiply high unsigned. The values in registers rs and rt are multiplied as unsigned integers to produce a 128 bit result. The high 64 bits of this result is placed in register rd.

RTL `Reg[rd] := (Reg[rs] *u Reg[rt])127:64`

4.1.12 DMULL

Encoding



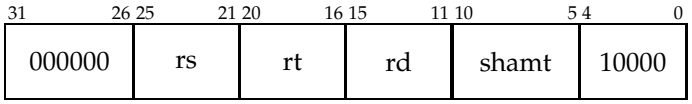
Assembly syntax `dmull $rd, $rs, $rt`

Description Multiply low. The values in registers rs and rt are multiplied to produce a 128 bit result. The low 64 bits of this result is placed in register rd. Note that this instruction produces the correct result no matter whether the values in rs and rt are interpreted as signed or unsigned values.

RTL `Reg[rd] := (Reg[rs] * Reg[rt])63:0`

4.1.13 DSLL

Encoding



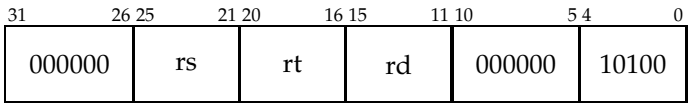
Assembly syntax `dsll $rt, $rs, shamt`

Description Shift left logical. The value in registers rt is shifted left by the number of bits specified in the shamt field. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] << shamt`

4.1.14 DSLLV

Encoding



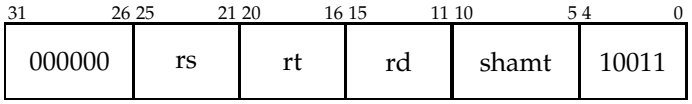
Assembly syntax `dsllv $rd, $rs, $rt`

Description Shift left logical variable. The value in registers rt is shifted left by the number of bits specified in register rs. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] << (Reg[rs] & 111111`$_2$`)`

4.1.15 DSRA

Encoding



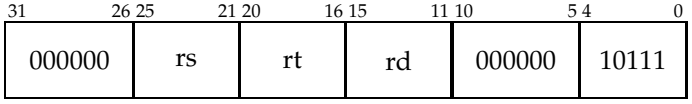
Assembly syntax `dsra $rt, $rs, shamt`

Description Shift right arithmetic. The value in registers rt is shifted right arithmetically by the number of bits specified in the shamt field. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>`$_s$ `shamt`

4.1.16 DSRAV

Encoding



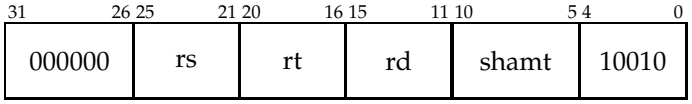
Assembly syntax `dsrav $rd, $rs, $rt`

Description Shift right arithmetic variable. The value in registers rt is shifted right arithmetically by the number of bits specified in register rs. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>`$_s$ `(Reg[rs] &` 111111_2`)`

4.1.17 DSRL

Encoding



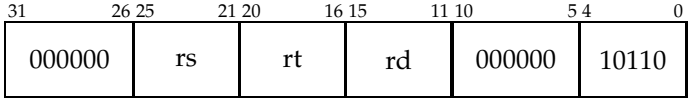
Assembly syntax `dsrl $rt, $rs, shamt`

Description Shift right logical. The value in registers rt is shifted right logically by the number of bits specified in the shamt field. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>`$_u$ `shamt`

4.1.18 DSRLV

Encoding



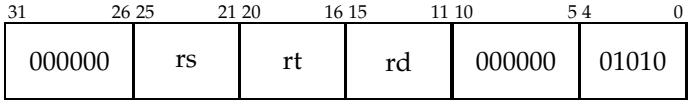
Assembly syntax `dsrlv $rd, $rs, $rt`

Description Shift right logical variable. The value in registers rt is shifted right logically by the number of bits specified in register rs. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>u (Reg[rs] & 111111₂)`

4.1.19 DIV

Encoding



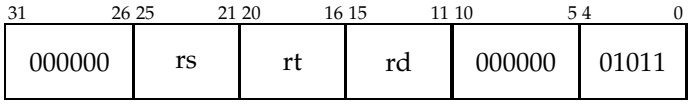
Assembly syntax `div $rd, $rs, $rt`

Description 32 bit divide. The signed 32 bit integer value in register rs is divided by the signed 32 bit integer value in register rt. The 32 bit result is sign extended to 64 bits and placed in register rd.

RTL `Reg[rd] := sign_ext(Reg[rs]31:0 /s Reg[rt]31:0)`

4.1.20 DIVU

Encoding



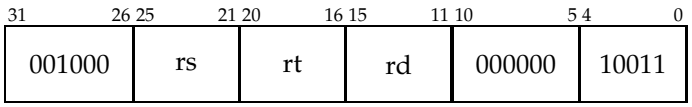
Assembly syntax `divu $rd, $rs, $rt`

Description 32 bit divide unsigned. The unsigned 32 bit integer value in register rs is divided by the unsigned 32 bit integer value in register rt. The 32 bit result is sign extended to 64 bits and placed in register rd.

RTL `Reg[rd] := sign_ext(Reg[rs]31:0 /u Reg[rt]31:0)`

4.1.21 DSUBU

Encoding



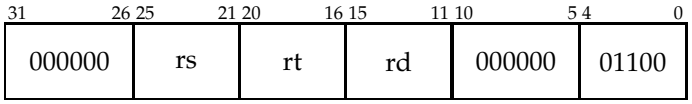
Assembly syntax `dsubu $rd, $rs, $rt`

Description Subtract unsigned. The value in register rt is subtracted from the value in register rs and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] - Reg[rt]`

4.1.22 MULH

Encoding



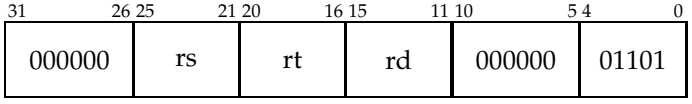
Assembly syntax `mulh $rd, $rs, $rt`

Description 32 bit multiply high. The values in registers rs and rt are multiplied as 32 bit signed 2s-complement integers to produce a 64 bit result. The high 32 bits of this result is sign extended and placed in register rd.

RTL $Reg[rd] := sign\_ext((Reg[rs]_{31:0} *_s Reg[rt]_{31:0})_{63:32})$

4.1.23 MULHU

Encoding



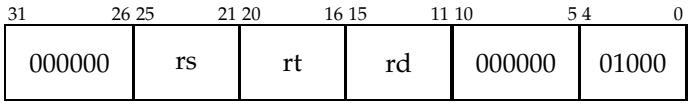
Assembly syntax `mulhu $rd, $rs, $rt`

Description 32 bit multiply high unsigned. The values in registers rs and rt are multiplied as 32 bit unsigned integers to produce a 64 bit result. The high 32 bits of this result is sign extended and placed in register rd.

RTL $Reg[rd] := sign\_ext((Reg[rs]_{31:0} *_u Reg[rt]_{31:0})_{63:32})$

4.1.24 MULL

Encoding



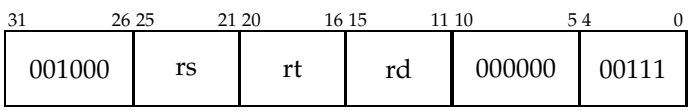
Assembly syntax `mull $rd, $rs, $rt`

Description 32 bit multiply low. The 32 bit values in registers rs and rt are multiplied to produce a 64 bit result. The low 32 bits of this result are sign extended and placed in register rd. Note that this instruction produces the correct result no matter whether the values in rs and rt are interpreted as signed or unsigned values.

RTL $Reg[rd] := sign\_ext((Reg[rs]_{31:0} * Reg[rt]_{31:0})_{31:0})$

4.1.25 NOR

Encoding



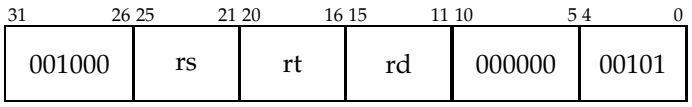
Assembly syntax `nor $rd, $rs, $rt`

Description Nor. The values in registers rs and rt are bitwise nored and the result is placed in register rd.

RTL `Reg[rd] := ~(Reg[rs] | Reg[rt])`

4.1.26 OR

Encoding



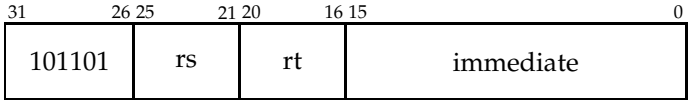
Assembly syntax `or $rd, $rs, $rt`

Description Or. The values in registers rs and rt are bitwise ored and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] | Reg[rt]`

4.1.27 ORI

Encoding



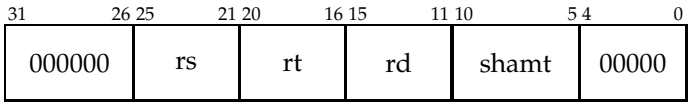
Assembly syntax `ori $rt, $rs, immediate`

Description Or immediate. The immediate field is zero extended, bitwise ored with the value in register rs and the result placed in register rt.

RTL `Reg[rt] := Reg[rs] | zero_ext(imm)`

4.1.28 SLL

Encoding



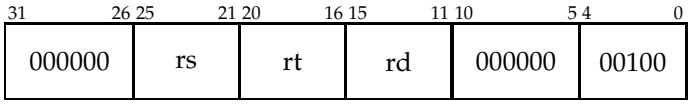
Assembly syntax `sll $rt, $rs, shamt`

Description 32 bit Shift left logical. The 32 bit value in registers rt is shifted left by the number of bits specified in the shamt field. The result is truncated to 32 bits, sign extended and placed in register rd.

RTL $\texttt{Reg[rd] := sign\_ext((Reg[rt] << shamt)}_{31:0}\texttt{)}$

4.1.29 SLLV

Encoding



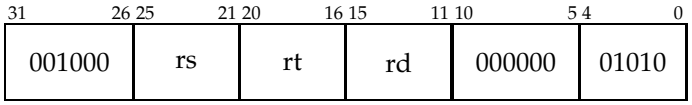
Assembly syntax `sllv $rd, $rs, $rt`

Description 32 bit shift left logical variable. The value in registers rt is shifted left by the number of bits specified in register rs. The result is truncated to 32 bits, sign extended and placed in register rd.

RTL $\texttt{Reg[rd] := sign\_ext((Reg[rt] << (Reg[rs] \& } 11111_2\texttt{))}_{31:0}$

4.1.30 SLT

Encoding



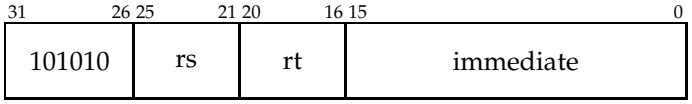
Assembly syntax `slt $rd, $rs, $rt`

Description Set less than. The values in registers rs and rt are compared as signed values. If rs is less than rt, then a 1 is placed in register rd, else a 0 is placed in register rd.

RTL $\texttt{Reg[rd] := if (Reg[rs] <}_s\texttt{ Reg[rt]) then 1 else 0}$

4.1.31 SLTI

Encoding



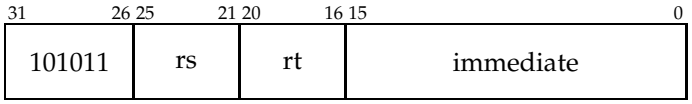
Assembly syntax `slti $rt, $rs, immediate`

Description Set less than immediate. The value in rs is compared as a signed value to the sign extended immediate. If rs is less than the immediate then a 1 is placed in register rt, else a 0 is placed in register rt.

RTL $\texttt{Reg[rt] := if (Reg[rs] <}_s\texttt{ sign\_ext(imm)) then 1 else 0}$

4.1.32 SLTIU

Encoding



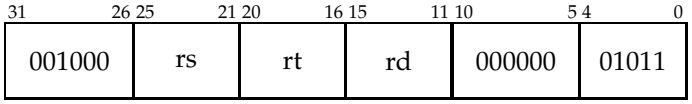
Assembly syntax `sltiu $rt, $rs, immediate`

Description Set less than immediate unsigned. The value in rs is compared as an unsigned value to the sign extended immediate. If rs is less than the immediate then a 1 is placed in register rt, else a 0 is placed in register rt.

RTL `Reg[rt] := if (Reg[rs] <u sign_ext(imm)) then 1 else 0`

4.1.33 SLTU

Encoding



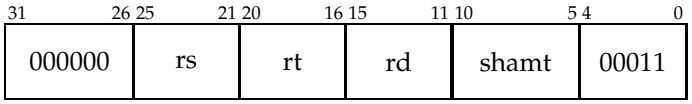
Assembly syntax `sltu $rd, $rs, $rt`

Description Set less than unsigned. The values in registers rs and rt are compared as unsigned values. If rs is less than rt, then a 1 is placed in register rd, else a 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] <u Reg[rt]) then 1 else 0`

4.1.34 SRA

Encoding



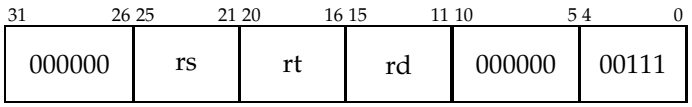
Assembly syntax `sra $rt, $rs, shamt`

Description Shift right arithmetic. The value in registers rt is shifted right arithmetically by the number of bits specified in the shamt field. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>s shamt`

4.1.35 SRAV

Encoding



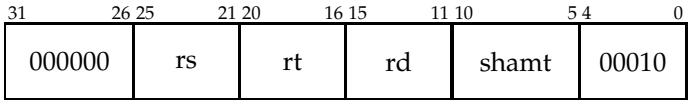
Assembly syntax `srav $rd, $rs, $rt`

Description Shift right arithmetic variable. The value in registers rt is shifted right arithmetically by the number of bits specified in register rs. The result is placed in register rd.

RTL `Reg[rd] := Reg[rt] >>s (Reg[rs] & 11111`$_2$`)`

4.1.36 SRL

Encoding



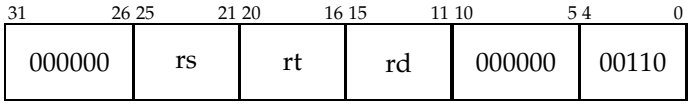
Assembly syntax `srl $rt, $rs, shamt`

Description 32 bit shift right logical. The 32 bit value in registers rt is shifted right logically by the number of bits specified in the shamt field. The result is sign extended and placed in register rd.

RTL $Reg[rd] := sign\_ext(Reg[rt]_{31:0} >>_u shamt)$

4.1.37 SRLV

Encoding



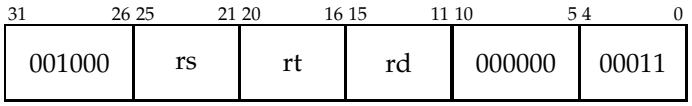
Assembly syntax `srlv $rd, $rs, $rt`

Description 32 bit shift right logical variable. The 32 bit value in registers rt is shifted right logically by the number of bits specified in register rs. The result is sign extended and placed in register rd.

RTL $Reg[rd] := sign\_ext(Reg[rt]_{31:0} >>_u (Reg[rs] \& 11111_2))$

4.1.38 SUBU

Encoding



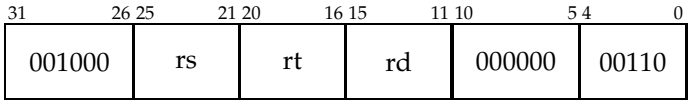
Assembly syntax `subu $rd, $rs, $rt`

Description 32 bit subtract unsigned. The 32 bit value in register rt is subtracted from the 32 bit value in register rs. The result is sign extended and placed in register rd.

RTL $Reg[rd] := sign\_ext(Reg[rs]_{31:0} -_{32} Reg[rt]_{31:0})$

4.1.39 XOR

Encoding



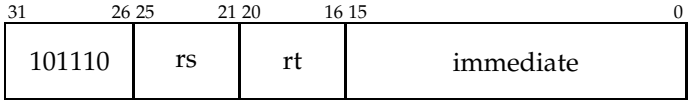
Assembly syntax `xor $rd, $rs, $rt`

Description Xor. The values in register rs and rt are bitwise exclusive-ored and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] ^ Reg[rt]`

4.1.40 XORI

Encoding



Assembly syntax `xori $rt, $rs, immediate`

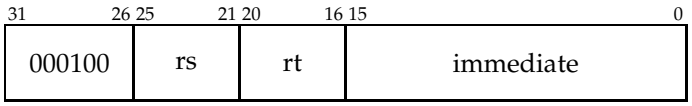
Description Xor immediate. The immediate field is zero extended, bitwise exclusive-ored with the value in register rs and the result placed in register rt.

RTL `Reg[rt] := Reg[rs] ^ zero_ext(imm)`

4.2 Branch instructions

4.2.1 BEQ

Encoding



Assembly syntax `beq $rs, $rt, immediate`

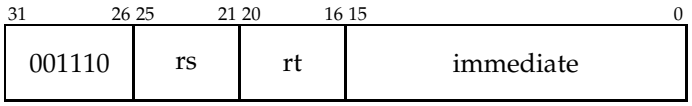
Description Branch equal. If the value in register rs is equal to the value in register rt, then the program counter is changed by the signed amount given in the immediate field.

RTL

```
program_counter := if (Reg[rs] == Reg[rt])
                   then
                     program_counter + (sign_ext(imm) * 4)
                   else
                     program_counter + 4
```

4.2.2 BGEZ

Encoding



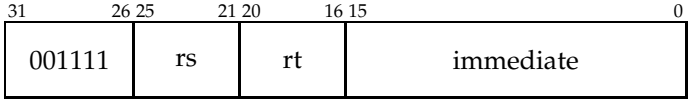
Assembly syntax `bgez $rs, immediate`

Description Branch greater than or equal to zero. If the value in register rs is greater than or equal to zero (its high bit is unset), then the program counter is changed by the signed amount given in the immediate field.

```
program_counter := if (Reg[rs] >= 0)
                   then
                     program_counter + (sign_ext(imm) * 4)
                   else
                     program_counter + 4
```

RTL

4.2.3 BGTZ



Encoding

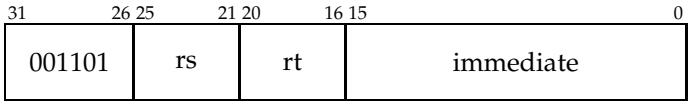
Assembly syntax `bgtz $rs, immediate`

Description Branch greater than zero. If the value in register rs is greater than zero, then the program counter is changed by the signed amount given in the immediate field.

```
program_counter := if (Reg[rs] > 0)
                   then
                     program_counter + (sign_ext(imm) * 4)
                   else
                     program_counter + 4
```

RTL

4.2.4 BLEZ



Encoding

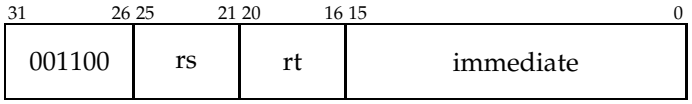
Assembly syntax `blez $rs, immediate`

Description Branch less than or equal to zero. If the value in register rs is less than or equal to zero, then the program counter is changed by the signed amount given in the immediate field.

```
program_counter := if (Reg[rs] <= 0)
                   then
                     program_counter + (sign_ext(imm) * 4)
                   else
                     program_counter + 4
```

RTL

4.2.5 BLTZ



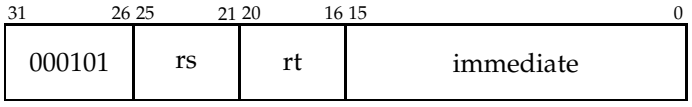
Encoding

Assembly syntax `bltz $rs, immediate`

Description Branch less than zero. If the value in register rs is less than zero, then the program counter is changed by the signed amount given in the immediate field.

```
     program_counter := if (Reg[rs] < 0)
                        then
                          program_counter + (sign_ext(imm) * 4)
                        else
RTL                       program_counter + 4
```

4.2.6 BNE



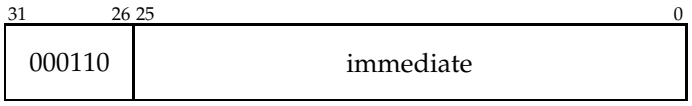
Encoding

Assembly syntax `bne $rs, $rt, immediate`

Description Branch not equal. If the value in register rs is not equal to the value in register rt, then the program counter is changed by the signed amount given in the immediate field.

```
     program_counter := if (Reg[rs] != Reg[rt])
                        then
                          program_counter + (sign_ext(imm) * 4)
                        else
RTL                       program_counter + 4
```

4.2.7 J



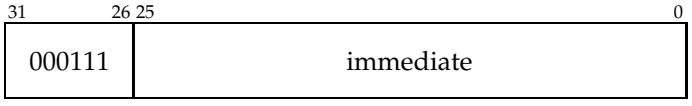
Encoding

Assembly syntax `j immediate`

Description Jump. Control jumps unconditionally to the address indicated by the 26 bit immediate field.

RTL `program_counter := zero_ext(imm) * 4`

4.2.8 JAL



Encoding

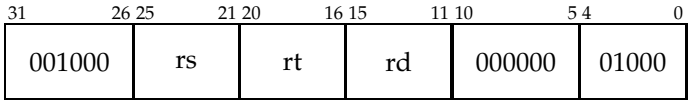
Assembly syntax `jal immediate`

Description Jump and link. The program counter + 4 is placed in the link register. The program counter unconditionally jumps to the address indicated by the 26 bit immediate field.

```
     Reg[31] := program_counter + 4
RTL  program_counter := zero_ext(imm) * 4
```

4.2.9 JR

Encoding



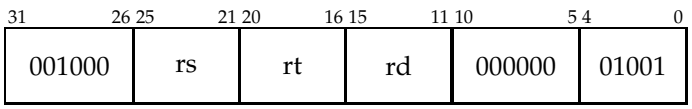
Assembly syntax `jr $rs`

Description Jump register. Control jumps to the address given in register rs.

RTL `program_counter := Reg[rs]`

4.2.10 JALR

Encoding



Assembly syntax `jalr $rs`

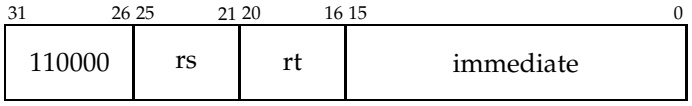
Description Jump and link register. The program counter + 4 is placed in the link register. The program counter jumps to the address given in the rs register.

```
     Reg[31] := program_counter + 4
RTL  program_counter := Reg[rs]
```

4.3 Memory operations

4.3.1 LB

Encoding



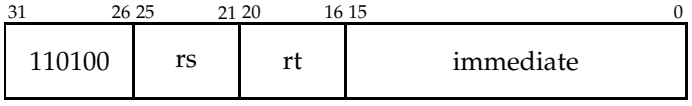
Assembly syntax `lb $rt, $rs(immediate)`

Description Load byte. The byte at the memory address given by register rs offset by the immediate value is sign extended to 32 bits and loaded into register rt.

RTL `Reg[rt] = sign_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.2 LBU

Encoding



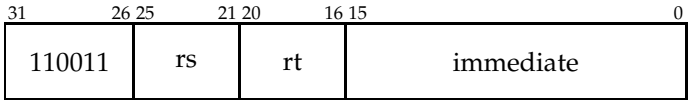
Assembly syntax `lbu $rt, $rs(immediate)`

Description Load byte unsigned. The byte at the memory address given by register rs offset by the immediate value is zero extended to 32 bits and loaded into register rt.

RTL `Reg[rt] = zero_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.3 LD

Encoding



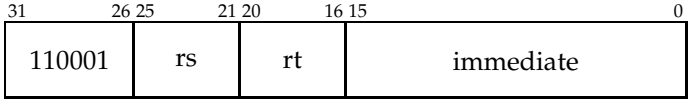
Assembly syntax `ld $rt, $rs(immediate)`

Description Load doubleword. The 64 bit word at the memory address given by register rs offset by the immediate value is loaded into register rt. The computed address must be aligned on a doubleword boundary.

RTL `Reg[rt] = Mem[Reg[rs] + sign_ext(imm)]`

4.3.4 LH

Encoding



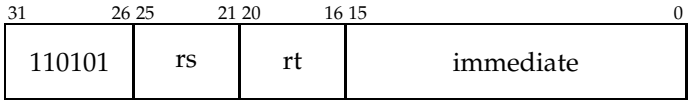
Assembly syntax `lh $rt, $rs(immediate)`

Description Load halfword. The 16 bit halfword at the memory address given by register rs offset by the immediate value is sign extended to 32 bits and loaded into register rt. The computed address must be aligned on a 2 byte boundary.

RTL `Reg[rt] = sign_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.5 LHU

Encoding



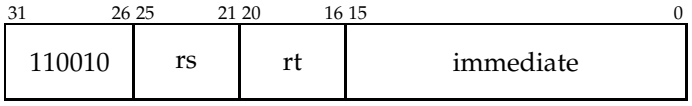
Assembly syntax `lhu $rt, $rs(immediate)`

Description Load halfword unsigned. The 16 bit halfword at the memory address given by register rs offset by the immediate value is zero extended to 32 bits and loaded into register rt. The computed address must be aligned on a 2 byte boundary.

RTL `Reg[rt] = zero_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.6 LW

Encoding



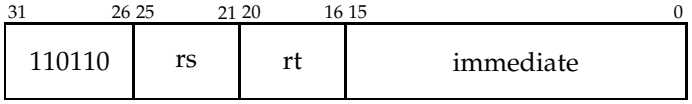
Assembly syntax `lw $rt, $rs(immediate)`

Description Load word. The 32 bit word at the memory address given by register rs offset by the immediate value is sign extended to 64 bits and loaded into register rt. The computed address must be aligned on a word boundary.

RTL `Reg[rt] = sign_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.7 LWU

Encoding



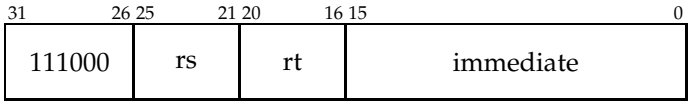
Assembly syntax `lwu $rt, $rs(immediate)`

Description Load word unsigned. The 32 bit word at the memory address given by register rs offset by the immediate value is zero extended to 64 bits and loaded into register rt. The computed address must be aligned on a word boundary.

RTL `Reg[rt] = zero_ext(Mem[Reg[rs] + sign_ext(imm)])`

4.3.8 SB

Encoding



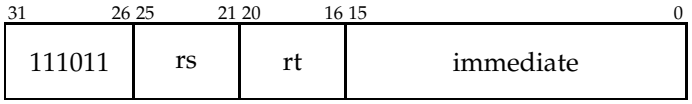
Assembly syntax `sb $rt, $rs(immediate)`

Description Store byte. The low order 8 bit byte of register rt is stored at the memory address given by register rs offset by the immediate value.

RTL `Mem[Reg[rs] + sign_ext(imm)] = Reg[rt]`$_{7:0}$

4.3.9 SD

Encoding



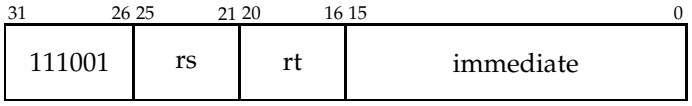
Assembly syntax `sd $rt, $rs(immediate)`

Description Store doubleword. The 64 bit word in register rt is stored at the memory address given by register rs offset by the immediate value. The computed address must be aligned on a doubleword boundary.

RTL `Mem[Reg[rs] + sign_ext(imm)] = Reg[rt]`

4.3.10 SH

Encoding



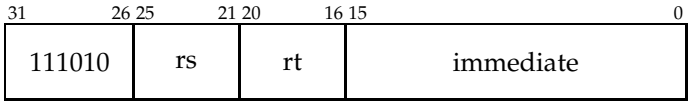
Assembly syntax `sh $rt, $rs(immediate)`

Description Store halfword. The low order 16 bit halfword of register rt is stored at the memory address given by register rs offset by the immediate value. The computed address must be aligned on a 2 byte boundary.

RTL `Mem[Reg[rs] + sign_ext(imm)] = Reg[rt]`$_{15:0}$

4.3.11 SW

Encoding



Assembly syntax `sw $rt, $rs(immediate)`

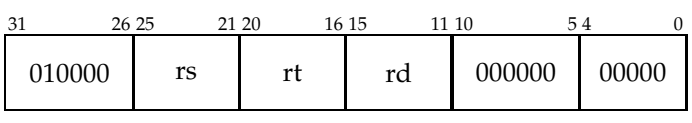
Description Store word. The 32 bit word in register rt is stored at the memory address given by register rs offset by the immediate value. The computed address must be aligned on a word boundary.

RTL `Mem[Reg[rs] + sign_ext(imm)] = Reg[rt]`$_{32:0}$

4.4 Floating point operate instructions

4.4.1 ADD.S

Encoding



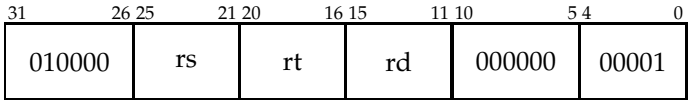
Assembly syntax `add.s $rd, $rs, $rt`

Description Add single precision floating point. The values in registers rs and rt are added and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] +f Reg[rt]`

4.4.2 SUB.S

Encoding



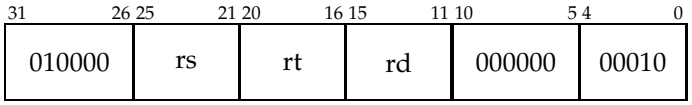
Assembly syntax `sub.s $rd, $rs, $rt`

Description Subtract single precision floating point. The values in registers rs and rt are subtracted and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] -f Reg[rt]`

4.4.3 MUL.S

Encoding



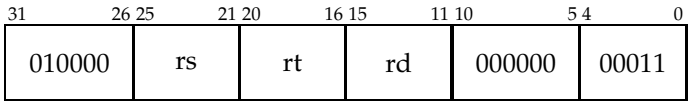
Assembly syntax `mul.s $rd, $rs, $rt`

Description Multiply single precision floating point. The values in registers rs and rt are multiplied and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] *f Reg[rt]`

4.4.4 DIV.S

Encoding



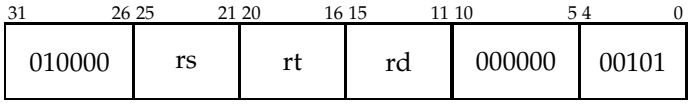
Assembly syntax `div.s $rs, $rt`

Description Divide single precision floating point. The values in registers rs and rt are added and the result is placed in the FD register.

RTL `Reg[FD] := Reg[rs] /f Reg[rt]`

4.4.5 ABS.S

Encoding



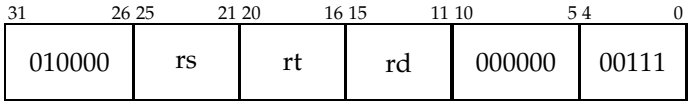
Assembly syntax `abs.s $rd, $rs`

Description Absolute value single precision floating point. The absolute value of the register rs is placed in register rd.

RTL `Reg[rd] := absolute_value(Reg[rs])`

4.4.6 NEG.S

Encoding



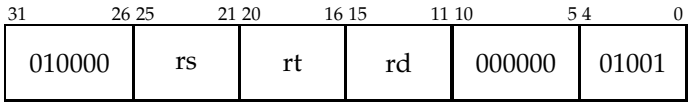
Assembly syntax `neg.s $rd, $rs`

Description Negate single precision floating point. The values in register rs is negated and the result is placed in register rd.

RTL `Reg[rd] := -Reg[rs]`

4.4.7 CVT.S.W

Encoding



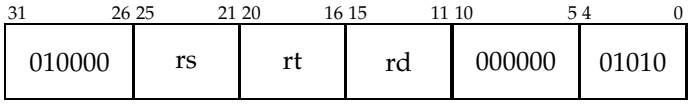
Assembly syntax `cvt.s $rd, $rs`

Description Convert integer to single precision floating point. The signed integer value in register rs is converted to floating point and the result is placed in register rd.

RTL `Reg[rd] := (float)((int)Reg[rs])`

4.4.8 CVT.S.D

Encoding



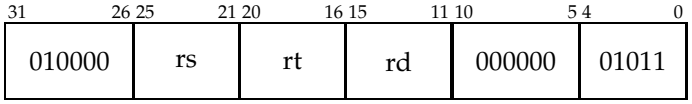
Assembly syntax `cvt.s $rd, $rs`

Description Convert double to single precision floating point. The double precision floating point value in register rs is converted to single precision floating point and the result is placed in register rd.

RTL `Reg[rd] := (float)((double)Reg[rs])`

4.4.9 CVT.W.S

Encoding



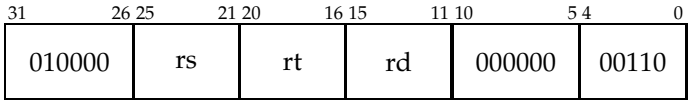
Assembly syntax `cvt.w $rd, $rs`

Description Convert single precision floating point to integer. The floating point value in register rs is converted to the fixed point and rounded to the nearest signed integer in which the value will fit.

RTL `Reg[rd] := (int)(round(Reg[rs]))`

4.4.10 TRUNC.S

Encoding



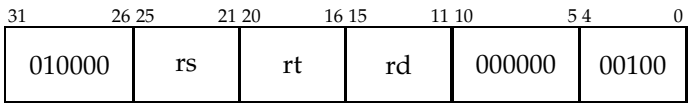
Assembly syntax `trunc.s $rd, $rs`

Description Truncate single precision floating point to integer. The floating point value in register rs is converted to a signed fixed point value. Any bits that don't fit are chopped. This is the instruction used by C to cast float to int.

RTL $\texttt{Reg[rd] := (int)Reg[rs]}_{\texttt{f}}$

4.4.11 C.EQ.S

Encoding



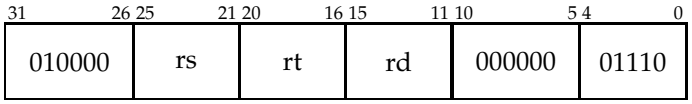
Assembly syntax `c.eq.s $rd, $rs, $rt`

Description Single precision floating point compare equal. The values in registers rs and rt are compared. If they are equal then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL $\texttt{Reg[rd] := if (Reg[rs] =}_{\texttt{f}}\texttt{ Reg[rt]) then 1 else 0}$

4.4.12 C.LT.S

Encoding



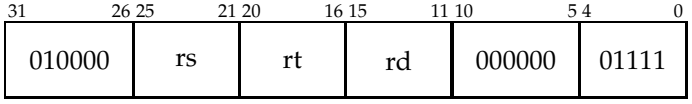
Assembly syntax `c.lt.s $rd, $rs, $rt`

Description Single precision floating point compare less than. The values in registers rs and rt are compared as floating point values. If register rs is less than register rt then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] <`$_f$` Reg[rt]) then 1 else 0`

4.4.13 C.LE.S

Encoding



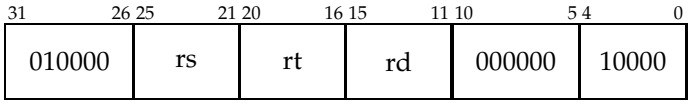
Assembly syntax `c.le.s $rd, $rs, $rt`

Description Single precision floating point compare less than or equal. The values in registers rs and rt are compared. If register rs is less than or equal to register rt then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] <=`$_f$` Reg[rt]) then 1 else 0`

4.4.14 ADD.D

Encoding



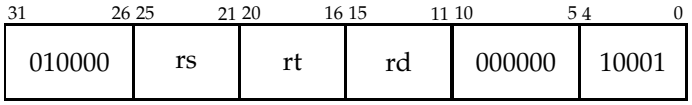
Assembly syntax `add.d $rd, $rs, $rt`

Description Add double precision floating point. The values in registers rs and rt are added and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] +`$_d$` Reg[rt]`

4.4.15 SUB.D

Encoding



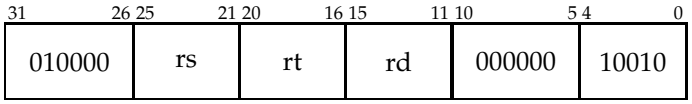
Assembly syntax `sub.d $rd, $rs, $rt`

Description Subtract double precision floating point. The values in registers rs and rt are subtracted and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] -d Reg[rt]`

4.4.16 MUL.D

Encoding



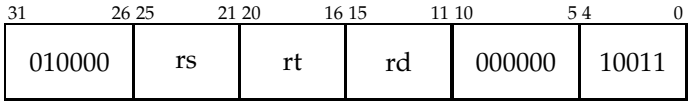
Assembly syntax `mul.d $rd, $rs, $rt`

Description Multiply double precision floating point. The values in registers rs and rt are multiplied and the result is placed in register rd.

RTL `Reg[rd] := Reg[rs] *d Reg[rt]`

4.4.17 DIV.D

Encoding



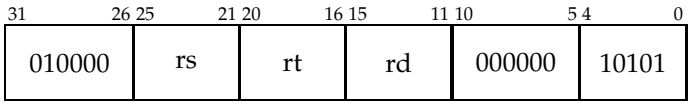
Assembly syntax `div.d $rs, $rt`

Description Divide double precision floating point. The values in registers rs and rt are added and the result is placed in the FD register.

RTL `Reg[FD] := Reg[rs] /d Reg[rt]`

4.4.18 ABS.D

Encoding



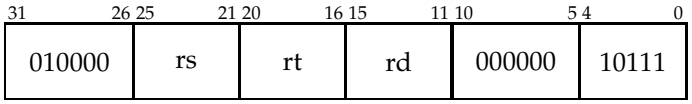
Assembly syntax `abs.d $rd, $rs`

Description Absolute value double precision floating point. The absolute value of the register rs is placed in register rd.

RTL `Reg[rd] := absolute_value(Reg[rs])`

4.4.19 NEG.D

Encoding



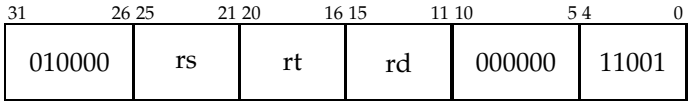
Assembly syntax `neg.d $rd, $rs`

Description Negate double precision floating point. The values in register rs is negated and the result is placed in register rd.

RTL `Reg[rd] := -Reg[rs]`

4.4.20 CVT.D.W

Encoding



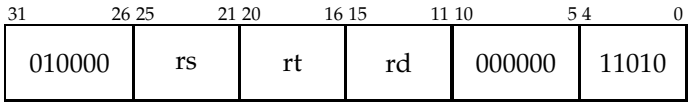
Assembly syntax `cvt.d $rd, $rs`

Description Convert integer to double precision floating point. The signed integer value in register rs is converted to floating point and the result is placed in register rd.

RTL `Reg[rd] := (double)((int)Reg[rs])`

4.4.21 CVT.D.S

Encoding



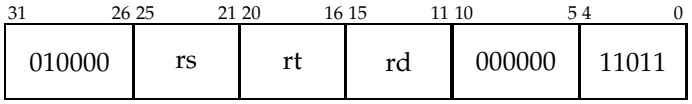
Assembly syntax `cvt.d $rd, $rs`

Description Convert single to double precision floating point. The single precision floating point value in register rs is converted to double precision and the result is placed in register rd.

RTL `Reg[rd] := (double)((float)Reg[rs])`

4.4.22 CVT.W.D

Encoding



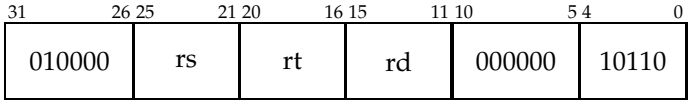
Assembly syntax `cvt.w $rd, $rs`

Description Convert double precision floating point to integer. The floating point value in register rs is converted to the fixed point and rounded to the nearest signed integer in which the value will fit.

RTL `Reg[rd] := (int)(round(Reg[rs]))`

4.4.23 TRUNC.D

Encoding



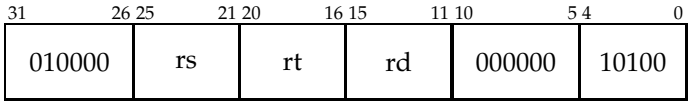
Assembly syntax `trunc.d $rd, $rs`

Description Truncate double precision floating point to integer. The floating point value in register rs is converted to a signed fixed point value. Any bits that don't fit are chopped. This is the instruction used by C to cast float to int.

RTL `Reg[rd] := (int)Reg[rs]`$_d$

4.4.24 C.EQ.D

Encoding



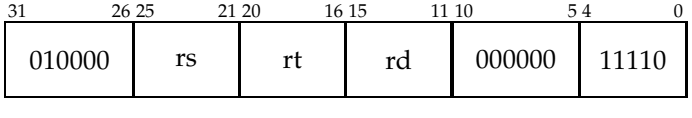
Assembly syntax `c.eq.d $rd, $rs, $rt`

Description Double precision floating point compare equal. The values in registers rs and rt are compared. If they are equal then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] =`$_d$ `Reg[rt]) then 1 else 0`

4.4.25 C.LT.D

Encoding



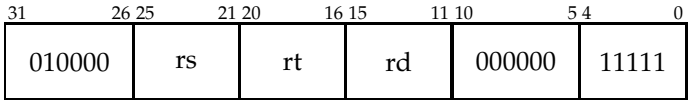
Assembly syntax `c.lt.d $rd, $rs, $rt`

Description Double precision floating point compare less than. The values in registers rs and rt are compared as floating point values. If register rs is less than register rt then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] <`$_d$ `Reg[rt]) then 1 else 0`

4.4.26 C.LE.D

Encoding



Assembly syntax `c.le.d $rd, $rs, $rt`

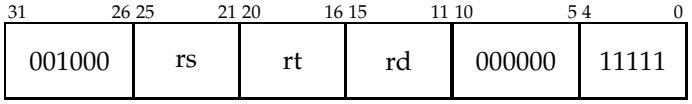
Description Double precision floating point compare less than or equal. The values in registers rs and rt are compared. If register rs is less than or equal to register rt then the value 1 is placed in register rd, else value 0 is placed in register rd.

RTL `Reg[rd] := if (Reg[rs] <=`$_{\text{d}}$ `Reg[rt]) then 1 else 0`

4.5 System administration operations

4.5.1 SYSCALL_HALT

Encoding



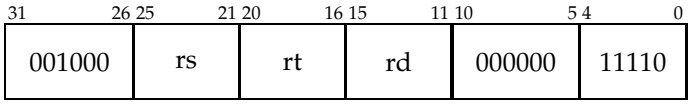
Assembly syntax `halt`

Description Halt. The simulator is halted.

RTL `halt`

4.5.2 SYSCALL_UNIX

Encoding



Assembly syntax `syscall_unix $rs`

Description UNIX system call. Register rs specifies the address of a structure in memory with the following fields:

```
uint32_t syscall_num
uint32_t result
uint32_t arg1
uint32_t arg2
uint32_t arg3
```

The simulator executes a UNIX I/O call of the form `result = sycall(arg1, arg2, arg3)`, where `syscall` is given by the following table:

Syscall Table

syscall_num	unix syscall
0x80	open
0x81	close
0x82	read
0x83	write
0x84	lseek
0x85	unlink
0x86	link
0x87	stat
0x88	fstat
0x89	lstat
0x90	chmod
0x91	rename