McGill University
ECSE 425: COMPUTER ORGANIZATION AND ARCHITECTURE
Winter 2016
Final Examination

2:00 PM – 5:00 PM, April 25th, 2016

Duration: 3 hours

Examiner: Prof. B. H. Meyer **Associate Examiner: Prof. W. J. Gross**

- **Write your name and student number in the space below. Do the same on the top of each page of this exam.**
- **The exam is 20 pages long. Please check that you have all 20 pages.**
- **There are six questions for a total of 160 points. Not all parts of all questions are worth the same number of points; read the whole exam first and spend your time wisely!**
- **This is a closed-book exam.**
- **Faculty standard calculators are permitted; no cell phones or laptops.**
- **Clearly state any assumptions you make.**
- **Write your answers in the space provided. Show your work to receive full credit, and clearly indicate your final answer.**

Name: _____

Student Number: _____

Q1: _____/40 **Q2: _____/20** **Q3: _____/20**

Q4: _____/20 **Q5: _____/30** **Q6: _____/30**

Total:

Question 1: Get Shorty (40 pts total, 4 pts each)

Please provide precise, justified responses; no credit will be given for vague generalities.

a. *Virtual memory* requires *precise exceptions*.
   i. What are *precise exceptions*?

   ii. Why are *precise exceptions* needed to support *virtual memory*?

b. Describe two different ways of measuring *cache performance*. Why does the choice of performance metric matter?

c. What are the advantages and disadvantages of a virtually indexed, physically tagged L1 cache?

d. Describe the advantages and disadvantages of *pipelining*.

e. What is a *branch target buffer*? Why use one?

f. Describe two sources of overhead unique to VLIW processors.

g. Explain how each of the following can uniquely limit ILP (when all other resources are unlimited):

i. The size of the *re-order buffer*.

ii. The number of registers available for *register renaming*.

h. Does single-threaded performance matter in the multi-core era? Justify your claim with support from the principles of design in computer architecture.

i. Briefly describe how standard vector architecture hides memory access latency. How does this differ from the approach taken in GPU architecture?

j. Identify all dependencies in the following code. Which, if any, prevent the loop from being vectorized?

```
for (i=0; i<100; i=i+1) {
     A[i+1] = A[i] + C[i];
     B[i+1] = B[i] + A[i+1];
}
```

Question 2: Where Does This Pipeline End? (20 pts)

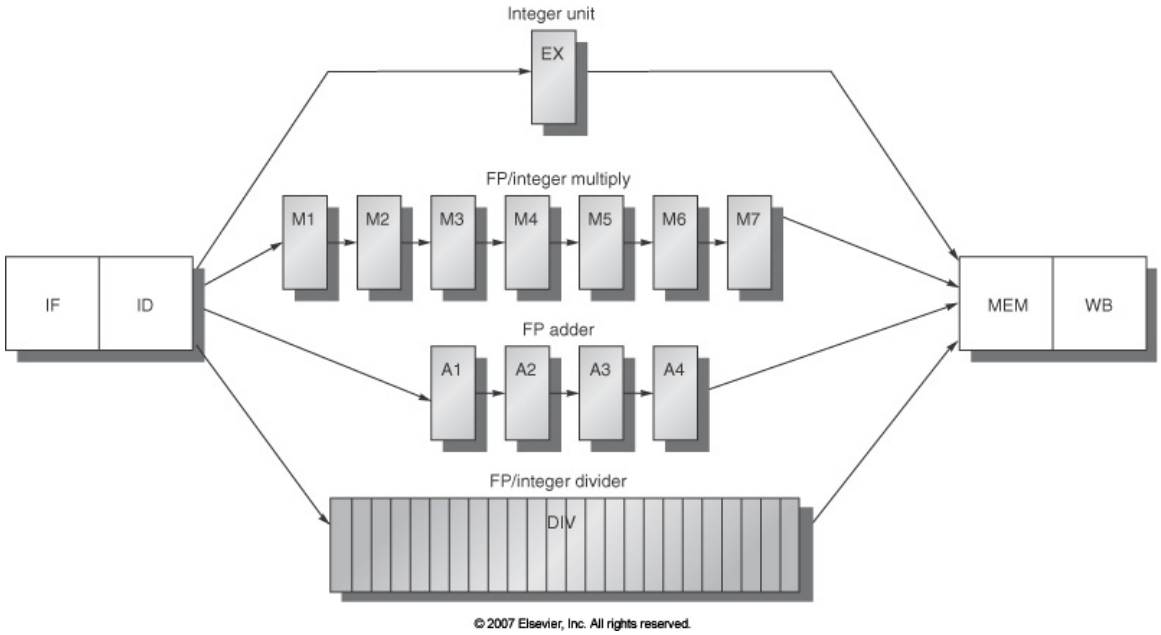
Consider the following code:

```
      LW    R6, 12(R1)
      ADDI  R7, R0, 2
EQ2:  BNEQ  R6, R7, EQ1
      ADD   R8, R2, R3
      MULT  R6, R5
      MFLO  R10
      J     END
EQ1:  ADDI  R7, R0, 1
      BNEQ  R6, R7, DEF
      SUB   R8, R2, R3
      LW    R9, 20(R1)
      MULT  R8, R9
      MFLO  R10
      J     END
DEF:  ADDI  R6, R0, 0
      ADDI  R8, R0, 0
      ADDI  R10, R0, 0
END:  SW    R6, 24(R1)
      SW    R8, 28(R1)
      SW    R10, 32(R1)
```

and corresponding initial memory state:

Address	Value
0x1000	5
0x1004	1
0x1008	3
0x100C	1
0x1010	2
0x1014	4
0x1018	8
0x101C	12
0x1020	3

This code will run on the standard MIPS floating point pipeline, illustrated below.



Assume:

- Full hazard detection and forwarding logic;
- The register file supports one write and two reads per clock cycle;
- Branch targets and conditions and jump targets are resolved in ID;
- Branches are handled by predicting not taken;
- J is treated as a branch instruction for branch prediction purposes;
- MFLO is treated as an ALU operation for hazard detection and forwarding purposes;
- Split L1 instruction and data caches service all requests in one clock cycle;
- Two or more instructions may simultaneously pass through MEM (WB) as long as only one makes use of the memory (register file);
- Structural hazards are resolved by giving priority to older instructions.

Complete the chart on the next page to show the execution of the loop if R1 = 0x1000 initially. Indicate pipeline stages with {**F**, **D**, **X**, **Mi**, **Ai**, **M**, **W**}, and stalls with **S**.

																		Instruction
																		1
																		2
																		3
																		4
																		5
																		6
																		7
																		8
																		9
																		10
																		11
																		12
																		13
																		14
																		15
																		16
																		17
																		18
																		19
																		20
																		21
																		22
																		23
																		24
																		25
																		26
																		27
																		28
																		29
																		30
																		31
																		32
																		33
																		34
																		35
																		36
																		37
																		38
																		39
																		40

Question 3: You Should Have Guessed You'd See This One Again (20 pts)

a. (10 pts) Two computers A and B are identical except for their branch prediction schemes. Each computer uses an eight-stage pipeline:

IF1 ID1 ID2 EX1 EX2 MEM WB

Computer A uses a static predicted not-taken scheme. Computer B uses a static predicted taken scheme. In each case, branch targets are calculated in ID1, and branch conditions are resolved in EX1.

If 25% of instructions are conditional branches, what fraction of these branches must be taken for the two computers to have equivalent performance? Assume each computer achieves the ideal CPI for every other instruction other than conditional branches.

b. (10 pts) Now consider the following C code:

```
for (int i=0; i<n; i++) {
     if (i % 2 == 0)
          … ;
     if (i % 3 == 0)
          … ;
}
```

This loop can be implemented with three branches and a jump:

- *A*: a branch conditioned on the loop bound: branch to the end when the loop is over;
- *B* and *C*: two branches conditioned on the the value of `i`;
- a jump from the end of the loop to the beginning of it (where the loops bound is checked once again …)

Assume a (2, 2) correlating branch predictor sufficiently large enough that the above branches do not alias (conflict), and that all predictor state is initialized to taken (*T*). Complete the following table for the first four iterations of the loop. Indicate the order in which branches are predicted (A, B, and C), the predictor state at the time of the prediction, the branch history register (BHR) contents (*newest*, …, *oldest*), resulting prediction, and the outcome.

Branch	Predictor State	BHR	Prediction	Outcome

Question 4: This One is Double-Wide! (20 pts)

Consider the following C code.

```
for (int i=0; i<n-1; i++) {
     sum[i] = sum[i] + a*sum[i+1];
}
```

This code can be implemented, in part, with the following MIPS assembly (refer to the attached MIPS Green Card as necessary). `$t0` is previously initialized to the base address of sum; `$t2` is previously initialized to the address of the last element of sum. `a` is already in `$f0`.

```
        addi   $t1, $t0, 4       # get address for i+1
top:    sltu   $t3, $t1, $t2     # compare i+1 and n (in $t2)
        beq    $t3, $zero, done  # branch if we are done
        lwc1   $f1, 0($t0)       # single precision load
        lwc1   $f2, 0($t1)       # single precision load
        mul.s  $f2, $f0, $f2     # single precision mult
        add.s  $f3, $f1, $f2     # single precision add
        addi   $t0, $t0, 4       # inc i
        addi   $t1, $t1, 4       # inc i+1
        j      top
done:
```

Execute the code through the completion of the first `j` instruction, assuming a *super-scalar* out-of-order processor with support for *hardware speculation*, and the following functional units:

Functional Unit	Cycles to Execute	No. of Functional Units
Integer ALU	1	2
FP Adder (used by LI.D, too)	4	2
FP Multiplier	7	1
Load/Store	*2	2

* The first cycle of Load/Store execution is used to calculate the effective address, and can proceed even if other operands are not yet ready.

Assume: the processor can issue and commit *two instructions per cycle*; all functional units are fully pipelined; *perfect branch prediction*; and, *all instructions* write the common data bus (CDB). Complete the following table.

Inst.	Inst. Issue	Begin EX	Finish EX	Write CDB	Commit
`addi $t1, $t0, 4`	1	2	2	3	4

Question 5: High Thread Count (30 pts)

Consider again the MIPS assembly from Q4:

```
        addi   $t1, $t0, 4        # get address for i+1
top:    sltu   $t3, $t1, $t2      # compare i+1 and n (in $t2)
        beq    $t3, $zero, done   # branch if we are done
        lwc1   $f1, 0($t0)        # single precision load
        lwc1   $f2, 0($t1)        # single precision load
        mul.s  $f2, $f0, $f2      # single precision mult
        add.s  $f3, $f1, $f2      # single precision add
        addi   $t0, $t0, 4        # inc i
        addi   $t1, $t1, 4        # inc i+1
        j      top
done:
```

a. (10 pts) Schedule the code for a processor capable of *fine-grained multi-threading* (FGMT). Assume that two identical threads, A and B, are available; schedule a single iteration of the loop for each. Assume the processor issues up two instructions of any type per cycle. Assume the same execution delays and functional units as in Q4. Complete the table, indicating the instruction (Inst) issued and the thread (Thrd, either A or B) from which it is issued, in each cycle.

Cycle	Instruction Slot 1		Instruction Slot 2	
	Thrd	Inst	Thrd	Inst
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

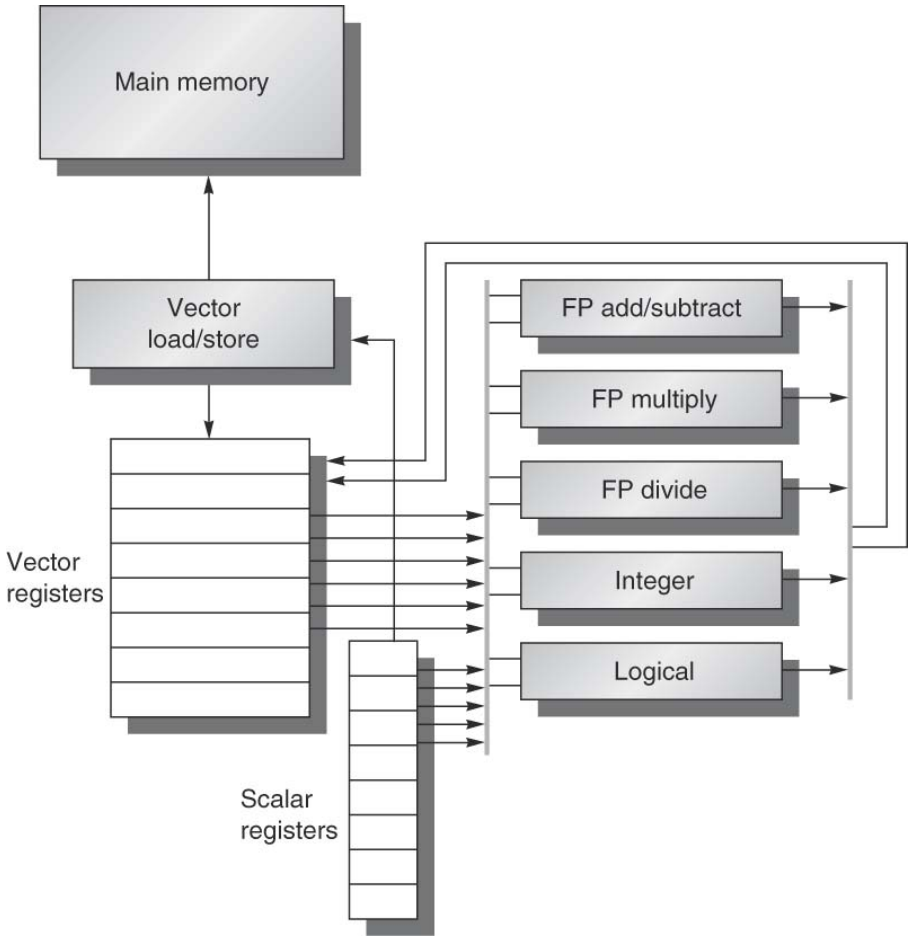
b. (10 pts) Now unroll the loop twice (so three copies of the loop body execute per iteration) and re-schedule the code to minimize stalls.

c. (10 pts) Schedule your unrolled code for execution on a VLIW processor capable of issuing an integer (Int) instruction, floating-point instruction (FP), and memory access instruction (Mem) each cycle. Assume the same execution delays and functional units as in Q4. Complete the table, indicating which instructions are issued each cycle.

Cycle	Int	FP	Mem
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			

Question 6: Architecture Unchained (30 pts total)

Assume the following vector architecture with 64-element vector registers, and one functional unit of each type.



Further assume the following *start-up times* for each functional unit above:

Functional Unit	Start-up Time
FP add/subtract	**6**
FP multiply	**7**
FP divide	**20**
Vector load	**12**

Each functional unit may begin a new operation the cycle after the last element of the previous operation enters the pipeline. *I.e.*, when two vector store instructions execute one after the other, the start-up time penalty is paid only once.

Now consider the following VMIPS assembly sequence for DAXPY.

```
L.D      F0,a      ;load scalar a
LV       V1,Rx     ;load vector X
MULVS.D  V2,V1,F0  ;vector-scalar multiply
LV       V3,Ry     ;load vector Y
ADDVV.D  V4,V2,V3  ;add
SV       V4,Ry     ;store the result
```

a. (6 pts) Assuming *no chaining* is available, identify the *convoys* in the above assembly and use *chimes* to estimate the number of cycles required to execute the above VMIPS assembly.

b. (8 pts) Now assume *chaining* is available. Identify the *convoys* in the above assembly and use *chimes* to estimate the speedup that is possible.

c. (10 pts) Assuming *chaining* is available, now use the start-up latencies for each functional unit to determine the number of cycles required to execute the above assembly. How much error is introduced when using *chimes* to approximate latency?

d. (6 pts) Now consider a multi-lane architecture. Given *two lanes*, use chimes to estimate speedup relative to part (a) for a system (i) *without chaining*, and (ii) *with chaining*.

Additional Page

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data

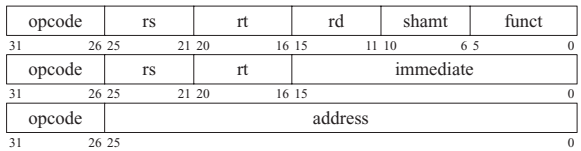
①

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

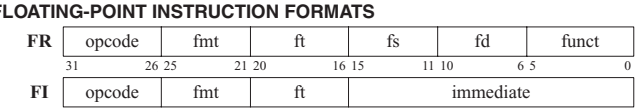
②

NAME, MNEMONIC		FORMAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0

(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127, Double Precision Bias = 1023.

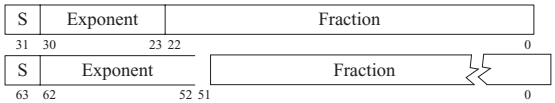
④

IEEE 754 Symbols

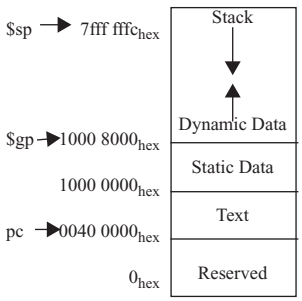
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

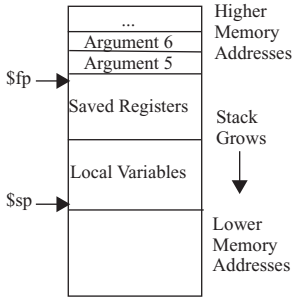
IEEE Single Precision and Double Precision Formats:



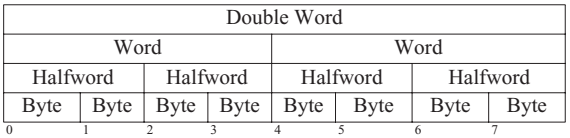
MEMORY ALLOCATION



STACK FRAME

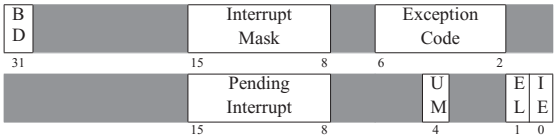


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together