Name Key _____

ISU ID # _____

Lab Section (circle one—your exam will be given a 0 if you do this incorrectly):

A (W 10-Noon), **B** (R 10-Noon), **C** (F 10-Noon), **D** (W 4-6)

CprE 381
Computer Organization and Assembly Level Programming

Practice Exam #1
2/8/2018 6:00-6:50PM

Directions: There are 9 11 questions in this exam. Each question is worth points indicated along with the problem. You should roughly spend 1 minute for every two points—plan accordingly. If a problem appears to be hard, move on and come back. Please read the questions carefully. Show your work, including any assumptions you need to make in order to solve the problems.

Calculators should NOT be used.

Problem	Score
1	_____ / 7 points
2	_____ / 8 points
3	_____ / 15 points
4	_____ / 9 points
5	_____ / 6 points
6	_____ / 5 points
7	_____ / 15 points
8	_____ / 10 points
9	_____ / 10 points
10	_____ / 10 points
11	_____ / 5 points
Total	_____ / 100 points

1. **MIPS Instruction Formats (7 points).** A new MIPS ISA sub-cult has decided to eliminate the I-format from MIPS (after all, simpler and more regular is better, right?). What impact will this have on conditional branch instructions? Make sure to include what format conditional branches will have to take.

Without I-format, cond. branches will have to be encoded as R-format (two registers are accessed for comparison and J-format doesn't have any register operands). rs and rt are still used for the equality comparison, rd will be unused and assigned to 00000, and shamt will be used as the immediate value. Therefore, branches will be +/- 2^4 instructions away.

2. **MIPS ABI (8 points).** Why does the MIPS ABI reserve a register, $at, just for the assembler? Provide a specific MIPS assemble example where $at needs to be used.

Some MIPS assembly instructions are pseudo instructions that take multiple MIPS ISA instructions to implement. Intermediate data w/in a pseudo inst. needs to be stored somewhere w/o overwriting or contending w/ other general purpose registers. $at is that somewhere. bge $t0, $t1, label → slt $at, $t0, $t1
beq $at, $zero, label

3. **Max Instruction (15 points).** We will be adding the **max** instruction to MIPS. This instruction takes the maximum value of two source registers and places it in a destination register. Of course, it has both a signed and unsigned version.

   (a) Translate the following instance of the max instruction into machine code. Use the MIPS ISA reference sheet and our lecture discussions to choose the appropriate format and select appropriate values for each field. Justify your answers.

```
max $v0, $a0, $t0
```

R-format since it has 3 reg operands
→ opcode → 00 0000
r_s → $a0 → 4 → 00100
r_t → $t0 → 8 → 01000
r_d → $v0 → 2 → 00010
shamt → unused → 00000
func → max → choose an unused code
→ 010100

(b) For backwards compatibility, older machines that don't have hardware that implements the **max** instruction will trap the illegal/unrecognized instruction and execute it in an exception handler (i.e., software). To this end, write a series of instructions that emulates the function of **max**.

```
slt $v0, $a0, $t0
beq $v0, $zero GREAT
add $v0, $t0, $zero
  EXIT
GREAT: add $v0, $a0, $zero
EXIT:
```

4. **NOOP instructions (9 points)**. No operation (NOOP) instructions are those whose execution has no side effects on the processor's architectural state (i.e., no registers are update and memory is not written). However, using an additional opcode for such an instruction seems wasteful. Provide three *approaches* to implementing a NOOP instruction using other MIPS instructions. Provide a specific example for each approach.

1) Arithmetic instruction produces same result as is already in dest. reg.
e.g., add $t0, $t0, $zero

2) Jump to next instruction
e.g.) j next
next:

3) Branch condition always false
e.g.) bne $zero, $zero, anywhere

4) Write to $zero register
e.g.) xor $zero, $t1, $s5

5. **Memory-Memory Instructions (6 points).** Why doesn't MIPS have an instruction that adds two values in memory and stores the result back into memory? Provide at least two *technical* reasons.

1) Encoding the memory operands' addresses would be impossible in a 32-bit inst, requiring larger, variable-length insts + more complicated fetch HW.

2) Execution requires many steps (load op1, load op2, perform add, store result and the corresponding address calcs. this complicates HW.

6. **Register Size (5 points)**. Suppose I decided to expand the size of the general-purpose registers ($0-$31) in MIPS to 64-bits. How would this impact the R-format instructions?

Reg width doesn't impact reg addr nor does it affect opcode or func width directly. It would imply that shamt should be 6 bits rather than 5 to provide 0 to 63 bit shifts. To maintain 32-bit insts, one would choose between reduce # of insts (smaller opcode) and needing 2 shifts for shifting between 32 and 63 bits.

7. **C to MIPS Translation (15 Points).** Translate the following C code into MIPS assembly. Assume a is in $a0, c is in $a1, and N is in $t0.

```
do {
    a += c[i++];
} while (a < N);
```

assume i is in $t1
assume word-size integers

```
loop:
sll  $t2, $t1, 2    # i*4
addu $t2, $t2, $a1  # &c[i]
lw   $t2, 0($t2)    # c[i]
addu $a0, $a0, $t2  # a += c[i]
addiu $t1, $t1, 1   # i++
slt  $t2, $a0, $t0
bne  $t2, $zero, loop
```

8. **MIPS Simulation (10 Points).** Assuming that $s0, $s2, and $t1 start out with 0x00000000, 0x0000000F, and 0xFFFFFFFF, what will they have after the execution of the following MIPS assembly? How many instructions are executed? How many instructions are executed for initial values of 0xFFFFFFFF, 0x0000000F, and 0xFFFFFFFF? Simplification is not necessary.

```
    j cond
loop:
    addiu $t1, $s0, $s1
    lw $t1, 0($t1)
    addiu $t0, $t0, $t1
    addiu $s0, $s0, 4
cond:
    slt $t1, $s0, $s2
    bne $t1, $zero, loop
```

0, 4, 8, 12, 16

$s0 → 0x 00000010
$s2 → 0x 0000000F
$t1 → 0x 00000000

∑ 3 + 4*6 = 27

-1, 3, 7, 11, 15 → same #

9. **MIPS Procedure Call (10 points).** What is the minimum size of the activation record for each of the following functions (assuming no compiler optimizations)? Justify your answer.

```
int blahblah(int a, int b) {
    int c[256];
    for (int i=0; i<256; i++){
        c[i] = i;
    }
    return a*c[blah(a)]+b*c[blah(b)];
}

int blah(int a) {
    return a & 0xFF;
}
```

assume int is word-sized
assume no FP
256 × 4 = 2^{10} bytes
$a0-$a3 → 4 × 4 = 2^4 bytes
$ra → 4 = 2^2 bytes
total: $2^{10} + 2^4 + 2^2$
word aligned
not word aligned, so need to add 2^2
leaf function, so no activation record
0
total: $2^{10} + 2^4 + 2^3$

10. **ISA vs ABI (10 points).** The same MIPS ISA sub-cult from question 1 wants to eliminate the hardware enforced registers (i.e., $zero and $ra). What former MIPS instructions will become pseudo instructions (list 2)? Provide a set of MIPS ISA instructions that implements these new pseudo instructions. How could the ABI change to keep the lack of a hardware-support $zero register from impacting most programs?

1) jal
2) jalr

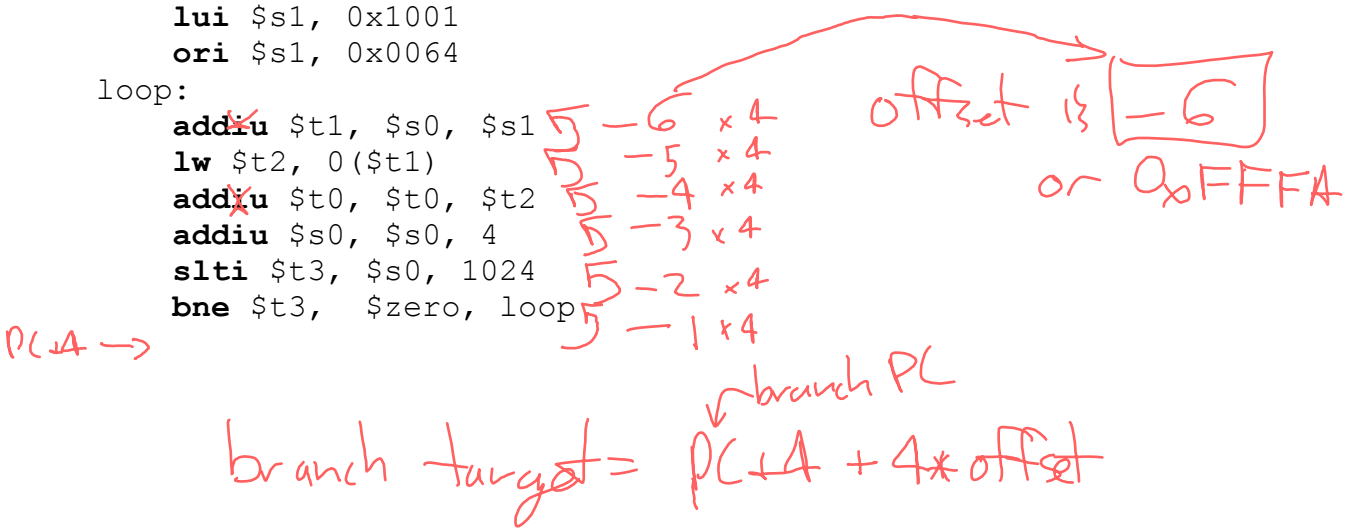
```
lui $ra, link upper
ori $ra, $ra, link lower
j label
link:
```

```
lui $ra, link upper
ori $ra, $ra, link lower
jr $t0
link:
```

The ABI should be updated to not allow $zero to be written to and to make sure the setting $zero occurs on any system boot (i.e., add and inst to clear $0 before $zero is used).

11. Branch Offset (5 points). What is the offset for the bne instruction in the following MIPS assembly? Assume the first instruction is at `0x00400000`.

```
    lui $s1, 0x1001
    ori $s1, 0x0064
loop:
    addiu $t1, $s0, $s1
    lw $t2, 0($t1)
    addiu $t0, $t0, $t2
    addiu $s0, $s0, 4
    slti $t3, $s0, 1024
    bne $t3,  $zero, loop
```



(END OF EXAM)

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

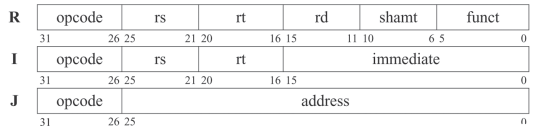
①

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

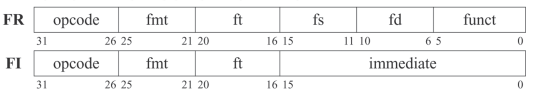


ARITHMETIC CORE INSTRUCTION SET

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	Yes

OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(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.

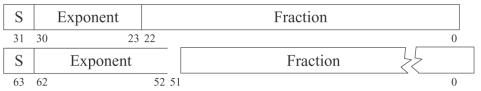
④

IEEE 754 Symbols

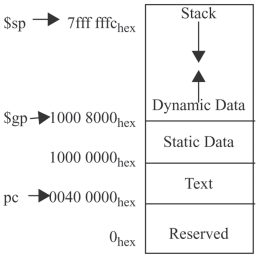
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

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

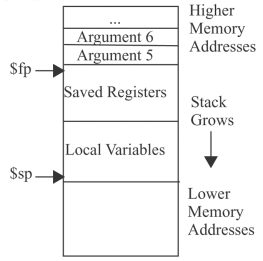
IEEE Single Precision and Double Precision Formats:



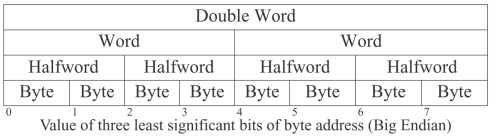
MEMORY ALLOCATION



STACK FRAME

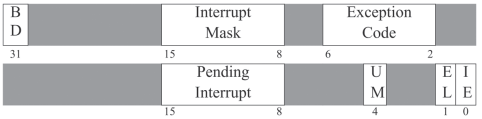


DATA ALIGNMENT



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

	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki	10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^6	Mega-	M	2^{20}	Mebi-	Mi	10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^9	Giga-	G	2^{30}	Gibi-	Gi	10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti	10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

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