CSCI 2500 — Computer Organization
Fall 2023 Test 3 (December 5, 2023)

Please silence and put away all laptops, phones, watches and any other electronic devices, etc. Using any electronic devices during the test is strictly prohibited.

1. DO NOT OPEN THIS TEST UNTIL TOLD TO DO SO!
2. READ THROUGH THE ENTIRE TEST BEFORE STARTING TO WORK.
3. YOU ARE ALLOWED ANY PRINTED OR HANDWRITTEN MATERIALS AND NOTES. NO OTHER MATERIALS ARE ALLOWED.
4. ABSOLUTELY NO ELECTRONIC DEVICES ARE ALLOWED.

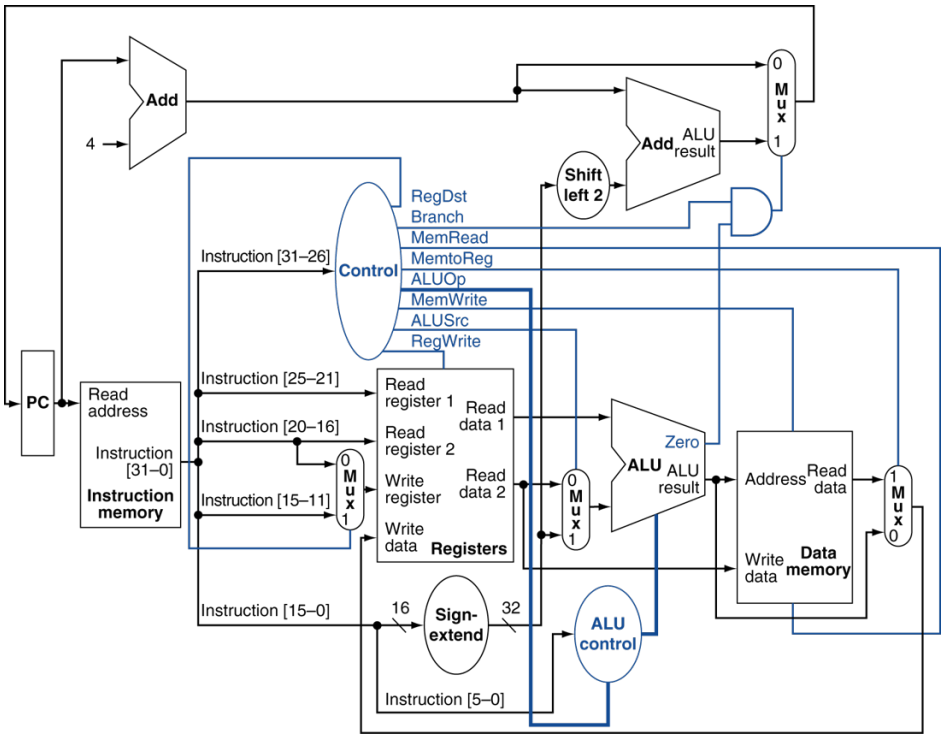
This test is designed to take 110 minutes; therefore, for 50% extra time, the expected time is 2 hours and 45 minutes and 100% extra time is 3 hours and 40 minutes. Questions will not be answered except when there is a glaring mistake or ambiguity in the statement of a question. Please do your best to interpret and answer each question. Document any assumptions that you had to make.

If you need extra space for your answers, you can add a page to your test booklet. Make sure this extra page is clearly labeled with your name and question number. Make a note on the title page of your test indicating that there is an extra page. When returning your test to a proctor, make sure you give them this extra page.

This test is out of 100 points but there are some extra credit questions or parts of questions.

1. **(20 POINTS)**

Recall "Datapath With Control" figure from the textbook and lecture slides:



Suppose the following instruction is executed in this datapath:

```
bne $8, $6, loop
```

where label `loop` refers to the instruction which is immediately preceding the given instruction . You may refer to the "MIPS Reference Data" pages at the end of this test.

Assume that data memory is all zeros and that the processor's registers have the following decimal values at the beginning of the clock cycle in which the above instruction is fetched:

r0	r1	r2	r3	r4	r5	r6	r8	r9	r31	PC
0	-2	1	-2	0	-1	2	7	6	6	4194366

What are the values of **all inputs and outputs** of the "Registers" unit, **all data inputs and outputs** of the "ALU" unit, **all inputs and outputs** of the "Add" unit in the right part of the datapath figure, and the following control signals: "Branch", "RegDst", "Zero", "ALUSrc", "RegWrite"? If there is not enough information to determine the value of any of these signals, write "N/A" for it. If the value of any signal is "Don't care", write "X" (note that "X" is not the same as "N/A").

Control
Branch = 1
RegDst = X
Zero = 0
ALUSrc = 0
RegWrite = 0

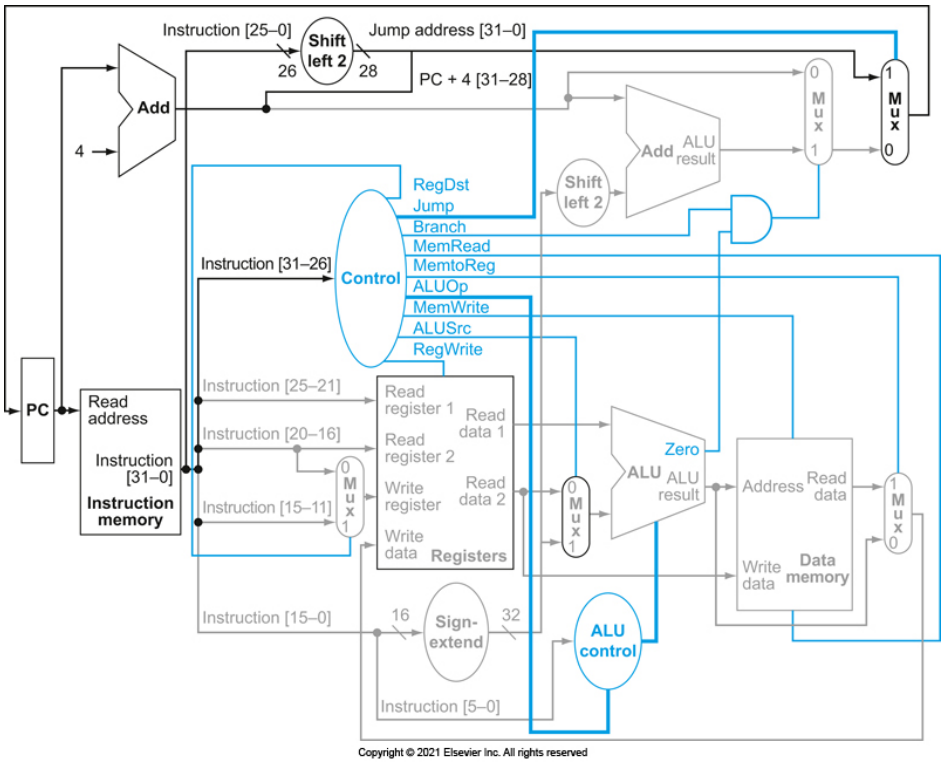
Register
Read Reg 1 = 8
Read Reg 2 = 6
Write Reg = X
Write Data = X
Read Data 1 = R[8] = 7
Read Data 2 = R[6] = 2

ALU
ALU 1 = 7
ALU 2 = 2
Zero (also part of control) = 0
ALU Result = 5 (7-2)
ALU Control = 110 (might not be required)

Right Adder
Add 1 = PC+4 (PC does not seem to be given)
Add 2 = -8 (-2 * 4)
Add Result = PC-4

2. **(20 POINTS)**

Recall "Datapath With Jumps Added" figure from the textbook and lecture slides:



Now, imagine that due to a memory chip malfunction, any words read from any address of the Instruction Memory always have 11 least significant bits (`Instruction[10-0]`) set to 11111111111.

Give a specific example of an instruction (i.e., a valid line of MIPS code) which would be affected by this malfunction and work incorrectly. Describe the exact behavior that will be observed when executing this instruction. If you believe that all instructions would still work correctly, explain why none of the instructions would be affected by the malfunction.

Instructions of the following forms are all correct:
Any R-Type instruction. For full points, students should explain how funct becomes 0x3f, which does not correspond to a defined operation.
Any I-Type instruction whose immediate value's 11 least significant bits are not 11111111111 to begin with. For full points, students should explain how the least significant 11 bits of the immediate value are altered. E.g. 'lw $t0 0($t1)' is incorrectly performed as 'lw $t0 2047($t0)'
Any J-Type instruction whose jump address's 11 least significant bits are not 11111111111 to begin with. For full points, students should explain how the least significant 11 bits of the jump location are altered. E.g. 'j L0' is incorrectly performed as 'j L1', where L0 represents address 0x0 in instruction memory and L1 represents address 0x(2047 * 4) in instruction memory.

Finally, give a specific example of an instruction (i.e., a valid line of MIPS code) which would remain unaffected by this malfunction and still work correctly. Describe the exact behavior that will be observed when executing this instruction. If you believe that not a single instruction would work correctly, explain why.

Instructions of the following forms are all correct:
Any I or J type instruction whose immediate value/jump location has 11111111111 as its 11 least significant bits. For full points, students should explain what the instruction does (similar to the cases above, but with the desired behavior), and that the mistake does not affect the instruction since the new bits are identical to the ones that should be there anyways.

3. **(20 POINTS)** For this question, you must show all work to receive credit! This question refers to the code given below and assumes a five-stage pipelined MIPS processor with stages denoted as `IF`, `ID`, `EX`, `MEM`, and `WB`. Use `*` to represent a bubble.

```
1  add   $a2, $s0, $s1
2  addi  $a0, $s0, 4
3  lw    $s0, 4($a0)
4  or    $a0, $s0, $s1
5  lw    $a0, 0($a0)
6  andi  $a1, $s1, 0xffff1200
7  sw    $s1, 4($a1)
8  addi  $a1, $s1, 8
9  sw    $a1, 0($a1)
10 add   $a2, $s0, $s1
```

Note the following assumptions:

- Forwarding is not used
- Additional hardware was used to allow us to test registers, calculate the branch address, and update the PC during the ID stage
- Statically predict branches not taken

Part a: (10/20 points) Using the notation of multi-cycle pipeline diagrams that we reviewed in class, show how all instructions go through the pipeline. Your diagram must show enough cycles until the last instruction leaves the pipeline but no more than 16 cycles. To save time, you may write just the instruction number (e.g., `#1`) instead of the whole instruction.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	F	D	E	M	W											
2		F	D	E	M	W										
3			*	*	F	D	E	M	W							
4				*	*	F	D	E	M	W						
5					*	*	F	D	E	M	W					
6						F	D	E	M	W						
7							*	*	F	D	E	M	W			
8								F	D	E	M	W				
9									*	*	F	D	E	M	W	
10										F	D	E	M	W		

Part b: (5/20 points) How many clock cycles would this code take to complete?

24

Part c: (5/20 points) Which types of hazards does this code have? Clearly circle all that apply.

(a) Structure (structural)
(b) Control
(c) Synchronization
(d) Forwarding
(e) Data
(f) None of the above

Extra credit part d: (10 points) Re-order instructions to minimize pipeline stalls without changing the semantics of the program.

```
Add $a2 $s0 $s1
Addi $a0 $s0 4
Andi $a1 $s1 0xffff1200
Addi $a1 $s1 8
Lw $s0 4($a0)
Sw $s1 4($a1)
Sw $a1 0($a1)
Or $a0 $s0 $s1
Nop
Add $a2 $s0 $s1
Lw $a0 0($a0)
```


4. **(20 POINTS)** Given a 32-bit architecture (both address and data buses are 32 bits) with byte-addressed main memory you designed a direct-mapped primary cache that has a total of 512 blocks with 1 word per block.

Part a: (10/20 points) Consider the sequence of memory accesses given below and write "hit", or "miss" next to each instruction. Compute the miss rate and express it either as a percentage or as a fraction of the form m/n.

```
load from 0xf00d0020
store  to 0xf00d0020
load from 0xf00d0021
load from 0xf00d1024
load from 0xf00d2029
load from 0xf00d1024
store  to 0xf00d0021
load from 0xf00df022
```

Miss rate: 4 / 8

512 blocks with 1 word per block.

512 indices, 9 bits for index, 2 bits needed for byte offset

load from 0xf00d0020 miss *
store to 0xf00d0020 hit *
load from 0xf00d0021 hit *
load from 0xf00d1024 miss *
load from 0xf00d2029 miss
load from 0xf00d1024 hit *
store to 0xf00d0021 hit *
load from 0xf00df022 miss

miss rate: 4/8

Part b: (4/20 points) For this cache configuration and the specific sequence of instructions given, indicate a single change that would lead to fewer misses (or write "None" if nothing could be done to decrease misses). Assume you cannot increase the total size of your cache (i.e., the total number of bits the cache occupies on the die). Be specific in describing the parameters of this change.

The main change would be to increase the size of each block to attempt to allow loading in one address to also load in some of the others that are later called. However, due to the large distance between different addresses, it is impossible to increase this block size without also changing the size of the cache.

Part c: (3/20 points) In the list of instructions above, use asterisks ("*") to mark at least two instructions that exhibit temporal locality. If there are none, clearly circle the statement below:

There are no instructions in the list above which exhibit temporal locality.

Shown above (multiple examples exist)

Part d: (3/20 points) In the list of instructions above, use hash signs ("#") to mark at least two instructions that exhibit spatial locality. If there are none, clearly circle the statement below:

No instructions

There are no instructions in the list above which exhibit spatial locality.

Extra credit part e: (5 points) Repeat the task from part (a) of this question but now with the 4-way set-associative primary cache that has a total of 512 blocks with 1 word per block and that uses LRU replacement policy.

```
load from 0xf00d0020
store  to 0xf00d0020
load from 0xf00d0021
load from 0xf00d1024
load from 0xf00d2029
load from 0xf00d1024
store  to 0xf00d0021
load from 0xf00df022
```

4 Way associative - 512/ 4 = 128 indicies
load from 0xf00d0020 miss
store to 0xf00d0020 hit
load from 0xf00d0021 hit
load from 0xf00d1024 miss
load from 0xf00d2029 miss
load from 0xf00d1024 hit
store to 0xf00d0021 hit
load from 0xf00df022 miss
miss rate: 4/8

Miss rate: 4 / 8

Extra credit part f: (5 points) Repeat the task from part (a) of this question but now with the fully set-associative primary cache that has a total of 512 blocks with 1 word per block and that uses FIFO replacement policy.

```
load from 0xf00d0020
store  to 0xf00d0020
load from 0xf00d0021
load from 0xf00d1024
load from 0xf00d2029
load from 0xf00d1024
store  to 0xf00d0021
load from 0xf00df022
```

Fully associative - one index
load from 0xf00d0020 miss
store to 0xf00d0020 hit
load from 0xf00d0021 hit
load from 0xf00d1024 miss
load from 0xf00d2029 miss
load from 0xf00d1024 hit
store to 0xf00d0021 hit
load from 0xf00df022 miss
miss rate: 4/8

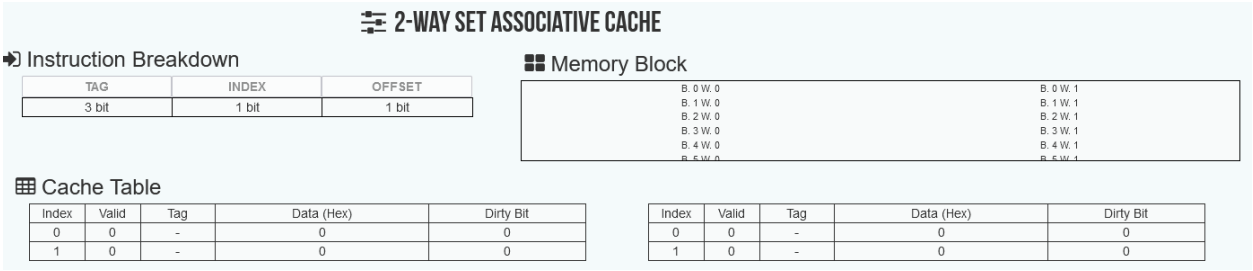
Miss rate: 4 / 8

Extra credit part g: (3 points) Discuss how miss rate is different for different cache configurations that you designed above and why. How would miss rate change for a different set of memory access instructions?

Due to the large size of the cache compared to the low size of the number of instructions, the miss rate for all 3 cache designs is the same. In order to have a different miss rate in the different cache designs, you would need more addresses accessed, for example having 5 words with the same index accessed for the 4 associative cache.

5. **Extra credit (15 POINTS)** Consider a scenario in which die size constraints limited CPU designers to only 64 Kib (i.e., 65,536 bits, not bytes) of space left for the on-chip cache. Suppose that you want the cache to be write-through, write on allocate, 4-way set-associative (SA) with a 2-word block size (and the machine word size is 16 bits).

   Draw the graphical representation of your cache using the same notation that we discussed in class. Show how memory addresses are broken down into offset, index, and tag bits, and the size of each field. Be sure to account for all required fields, not just the block data. Assume main memory is byte-addressed. The screen shot below is just a reminder on the notation, it is not the actual answer to this question (it is not even a 4-way SA cache). You need to give your answer in the space provided below the screen shot.

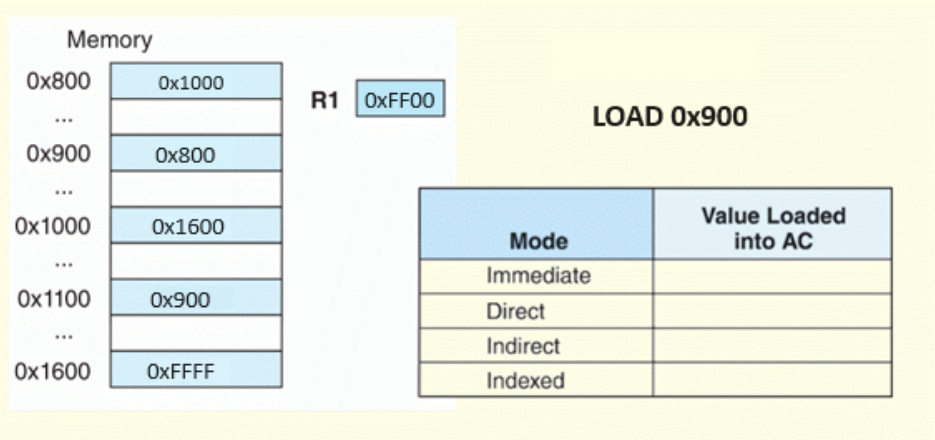


T_read = T_seek + T_rotational + T_transfer

T_rotation = 1 / 7200 RPM /60 s /2 = 8.33/2 ms = 4.17 ms
T_transfer = 512 MB / 100 MB/s= 5.12s

T_read = 5.12 s + 4.17 ms + 0.005 s

6. **(20 POINTS)** Recall a computer architecture that we reviewed in class where all instructions have at most one argument, and "LOAD" instruction loads the value into the accumulator register. For the instruction shown, what value is loaded into the accumulator ("AC") for each addressing mode? For indexed addressing mode, assume "R1" to be the register that holds the offset. Remember that it is a 16-bit architecture, so all values are 16 bits and integer values can be signed. Fill in the table directly in the figure below:



L1 Cache Access Time = Access Time for L1 Cache = 1 cycle (given CPI = 1 and clock rate = 2.5GHz)
Therefore, L1 Cache Access Time = 1 / (2.5 GHz) = 0.4 ns

Average Memory Access Time (L1-only system):

Average Access Time = L1 Cache Access Time + L1 Cache Miss Rate * L1 Cache Miss Penalty
Average Access Time (L1-only) = 0.4 ns + 0.02 * 100 ns = 0.4 ns + 2 ns = 2.4 ns

Three-Level Cache System:

Average Access Time (three-level cache system) = L1 Cache Access Time + L1 Miss Rate * (L2 Cache Access Time + L2 Miss Rate * (L3 Cache Access Time + L3 Miss Rate * Main Memory Access Time))

Substituting the values:
Average Access Time (three-level cache system) = 0.4 ns + 0.02 * (6 ns + 0.01 * (10 ns + 0.005 * 100 ns)) = 0.5221 ns

Speedup = Average Access Time (L1-only system) / Average Access Time (three-level cache system)
Speedup = 2.4 / 0.5221 ns ~ 4.60

This page is left blank for scratch work. You may show your work but do not put your solutions here.

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

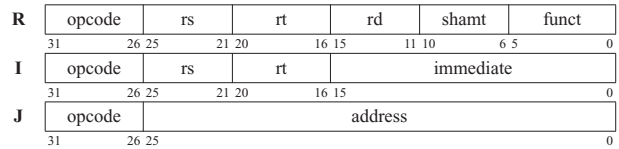
①

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

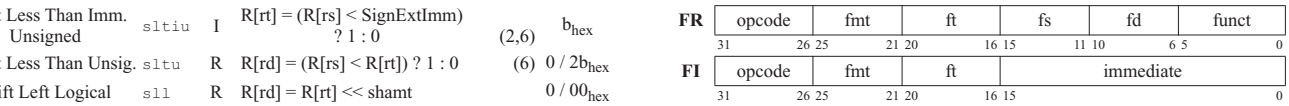


ARITHMETIC CORE INSTRUCTION SET

② OPCODE / FMT /FT / FUNCT (Hex)

NAME, MNEMONIC		FORMAT	OPERATION		
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	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.

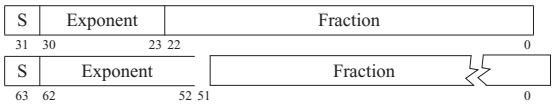
④

IEEE 754 Symbols

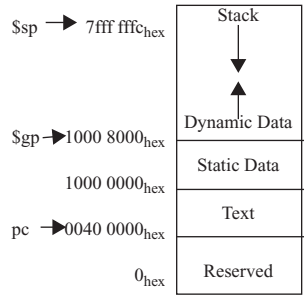
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

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

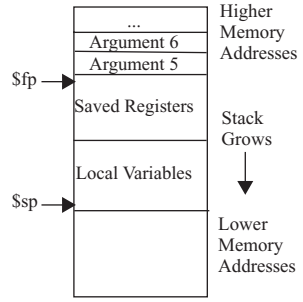
IEEE Single Precision and Double Precision Formats:



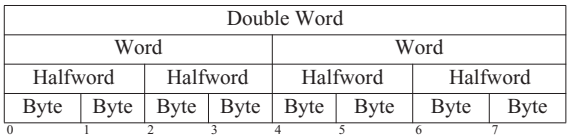
MEMORY ALLOCATION



STACK FRAME

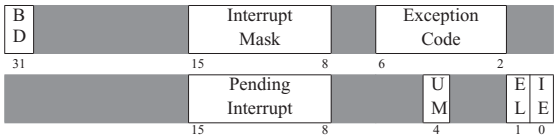


DATA ALIGNMENT



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

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
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