THE HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY

Computer Organization (COMP 2611)

Spring Semester, 2014

Final Examination

May 23, 2014

Name: _____ **Student ID:** _____

Email: _____ **Lab Section Number:** _____

Instructions:

1. **This examination paper consists of 13 pages in total, including 7 questions within 11 pages, 2 appendices.**
2. **Please do NOT use pencils in answering the questions.**
3. **Please write your name, student ID, email and lab section number on this page.**
4. **Please answer all the questions in the spaces provided on the examination paper.**
5. **Please read each question very carefully, answer clearly and to the point. Make sure that your answers are neatly written.**
6. **Keep all pages stapled together. You can tear off the appendix only.**
7. **Calculator and electronic devices are not allowed.**
8. **The examination period will last for 2.5 hours.**
9. **Stop writing immediately when the time is up.**

Question	Percentage %	Marker
1 Cache Performance	12	
2 Cache Architecture	15	
3 Single-cycle Datapath & Control	13	
4 Multi-cycle Datapath & Control	20	
5 MIPS Recursion	15	
6 MIPS Programming	10	
7 Pipeline	15	
TOTAL	100	

Question 1: Cache Performance (12 marks)

a) The average memory access latency for a microprocessor with a single level cache is 2.4 clock cycles. If the data is present in the cache, it takes 1 clock cycle to fetch. Otherwise, (if data is not in the cache), 80 clock cycles are necessary to get it from the main memory.
We want to improve the average memory access latency to 1.5 clock cycles by adding a 2nd level of cache on chip. This 2nd level of cache can be accessed in 6 clock cycles, and does not affect the first level cache access patterns and hit times, nor the access time to the main memory (i.e., 80 additional clock cycles).

1. Give the general equation for the average memory access latency in a memory hierarchy with two-level cache, using the following notations: L1 Cache, hit time t_1, hit rate h_1; L2 Cache, hit time t_2, hit rate h_2; main memory access time t_{mem}. (2 marks)

   Avg Latency = $t_1 + (1-h_1)*t_2 + (1-h_1)*(1-h_2)*t_{mem}$

2. What is the expected hit rate in the 2nd level cache to achieve the target speedup (from 2.4 to 1 cycle)? Briefly show your calculation steps. (6 marks)

   With one level cache:
   AMAT = hit time + miss rate1 * miss penalty
   2.4 = 1 + miss rate * 80
   Miss rate 1 = 1.75%

   With two level cache:
   1.5 = 1 + 0.0175 * (6 + miss rate2 * 80)
   Miss rate 2 = 28.2%
   Hit rate 2 = 71.8%

b) For a given machine, and a given program, if all the data and instructions could always be found in the first level cache, then on average the processor can finish an instruction within 2 clock cycles (i.e. the ideal CPI, $CPI_{optimum} = 2$).

   However, measurements obtained show that the instruction miss rate is 12% and the data miss rate is 6%, and that on average, 30% of all instructions contain a data reference. The miss penalty for the cache is 10 cycles. What is the actual CPI for this program on this machine if we consider memory access overhead? (4 marks)

   CPI = 2 + instruction miss cycles + data miss cycles
   = 2 + 0.12*10 + 0.3*0.06*10
   = 3.38

Question 2: Cache Architecture (15 marks)

a) Consider a 16-way set associative cache of size 64K bytes. If a memory address is 32 bits and the block size is 64 bytes, answer the following questions. Briefly show your calculations. (5 marks)

Number of blocks in the cache =
Answer: 64K / 64 = 1024 = 2^{10}

Number of sets in the cache =
Answer: 1024 / 16 = 64 = 2^{6}

Number of bits for the byte offset =
Answer: 6

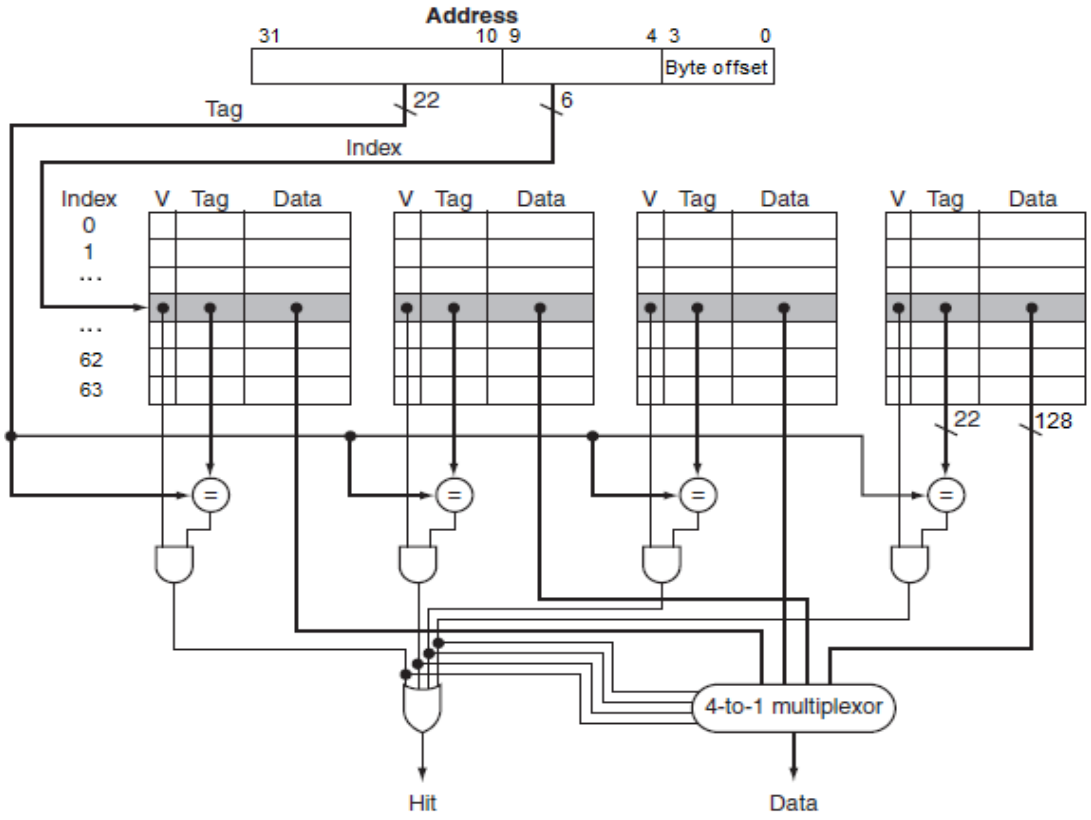
Number of bits for the Index field =
Answer: 6

Number of bits for the Tag field =
Answer: 32 - 6 - 6 = 20

b) Given a 2-way set-associative cache with 3-bit tag, below is a sequence of memory accesses which are mapped to the same cache set. Assume the cache is initially empty. If LRU cache replacement strategy is used, fill in the blank below to indicate if a Hit or Miss happened to the each access. (5 marks)

	Tag field of the memory accesses generated by CPU (from left to right):								
	101	101	100	111	011	100	011	011	100
Hit/Miss	Miss	Hit	Miss	Miss	Miss	Miss	Hit	Hit	Hit
	Cache Access Tracking:								
MRU	101	101	100	111	011	100	011	011	100
LRU			101	100	111	011	100	100	011

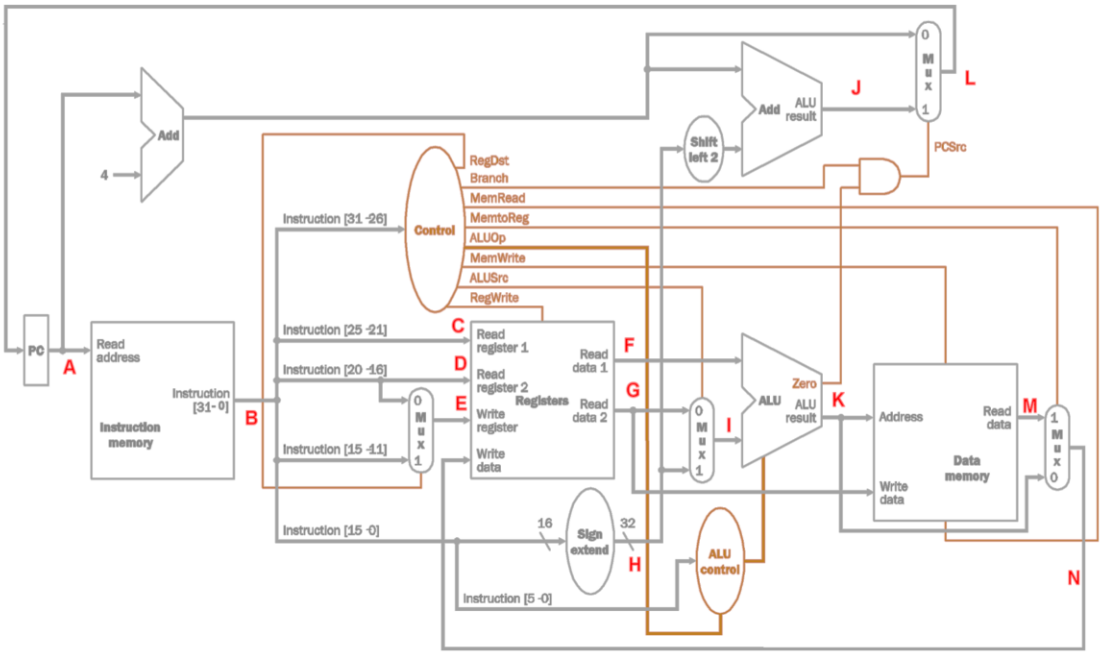
c) Consider the following 4-way set associative cache.



Complete the last two columns of the following table for the given sequence of memory accesses. (5 marks)

Address of the memory access generated by CPU	Assigned cache set	Hit or miss?
0000 1111 0101 0010 0011 0111 0100 0110	110100	Miss
0000 1111 0101 1010 0011 0111 0100 0110	110100	Miss
0000 1111 0101 0010 0011 0111 0111 1101	110111	Miss
0000 1111 0101 0010 0011 0111 0100 0110	110100	Hit
0000 1111 0101 0010 0011 0111 0111 0110	110111	Hit
0000 1111 0101 0010 0011 0111 0100 1001	110100	Hit

Question 3: Single Cycle Datapath & Control (13 marks)



a) Instruction `ori $s0, $t0, 20` is executed in the above single cycle datapath. Assume the values stored in registers `$s0` and `$t0` are 7 and 9 respectively. Fill in the table with the proper binary numbers that correspond to the labels in the figure above. (5 marks)

A	Address of the instruction
B	001101 01000 10000 0000 0000 0001 0100
F	0000 0000 0000 0000 0000 0000 0000 1001
G	0000 0000 0000 0000 0000 0000 0000 0111
I	0000 0000 0000 0000 0000 0000 0001 0100
N	0000 0000 0000 0000 0000 0000 0001 1101

b) Complete the following table with control signal values to execute the `ori` instruction. Don't care signals are represented by 'x'. (8 marks)

RegDst	Branch	MemRead	MemtoReg	ALU Control Input (4 bits)	MemWrite	ALUSrc	RegWrite
0	0	0	0	0001	0	1	1

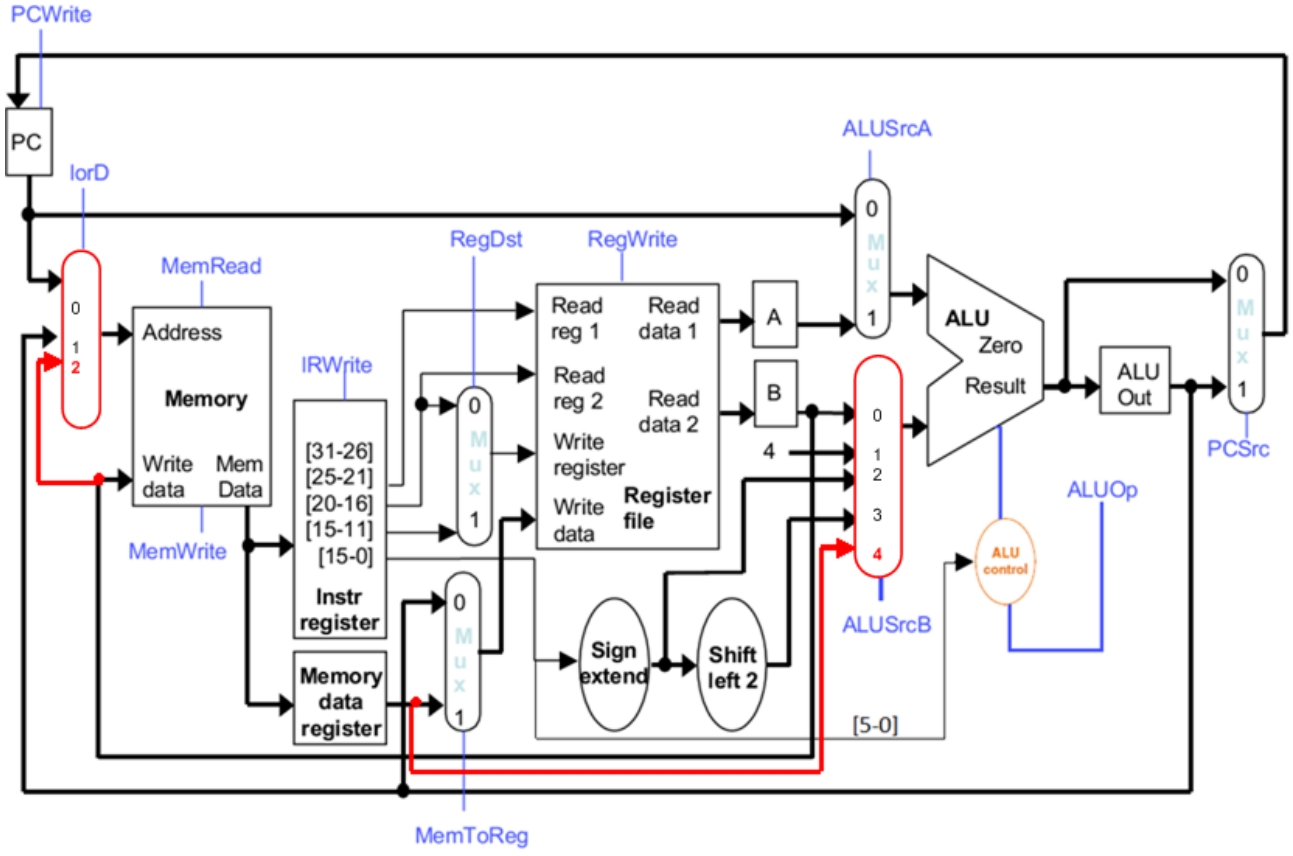
Question 4: Multi-Cycle Datapath & Control (20 marks)

Consider implementing an imaginary R-type instruction; *subtract memory* (`submem`), in MIPS. The instruction is similar to the *subtract* (`sub`) instruction, except that it takes the second operand (i.e. the subtrahend) from the memory address stored in register `rt`. The instruction format and its syntax and meaning are shown below:

Field	op	rs	rt	rd	shamt	func
Bits	31-26	25-21	20-16	15-11	10-6	5-0

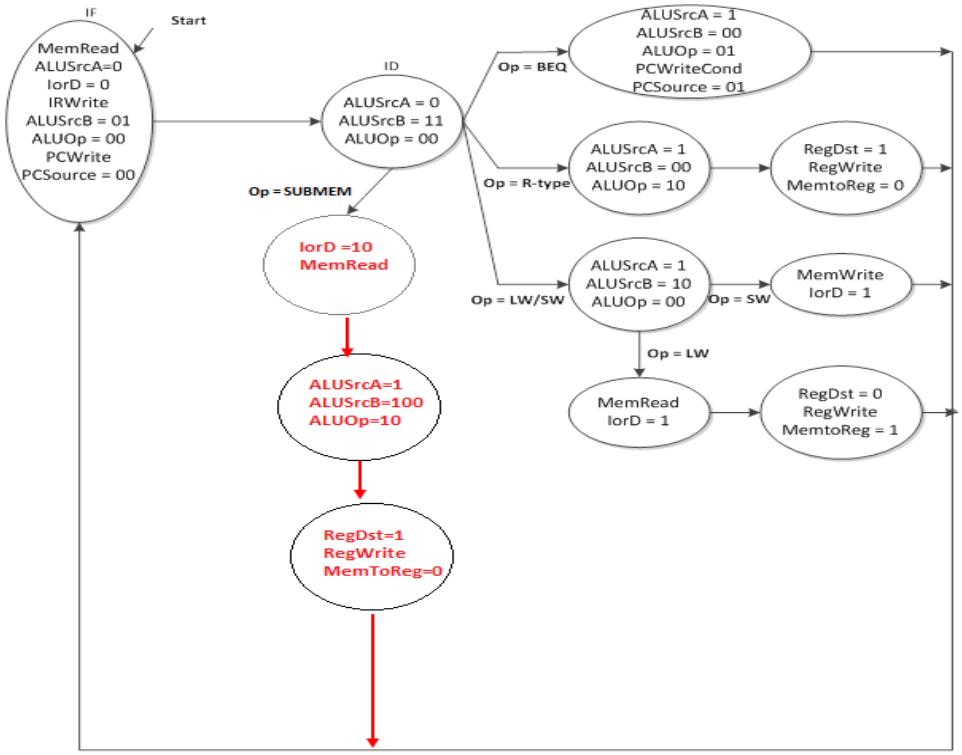
```
submem $rd, $rs, $rt      # Reg[$rd] = Reg[$rs] - mem[Reg[$rt]]
```

a) Make the necessary changes to the multicycle datapath below to support `submem`. (8 marks)
Rules/Hints: i) No modification to the main functional units. No new control signals. You should only add wires, and/or expand existing multiplexers. ii) You may assume the ALU control module recognizes the instruction (from its `func` field) and supply the correct control signal to the ALU. iii) With assumption that memory access or ALU operations take one clock cycle to finish, the `submem` instruction takes 5 cycles to finish in the modified multicycle datatpath.



Cycle	Operations
Cycle 1	- Fetch - PC = PC + 4
Cycle 2	- Decode - Read rs and rt - Calculate the branch target (for possible beq)
Cycle 3	- Do a memory read: MDR = Mem[Reg[rt]] (memory read takes one cycle)
Cycle 4	- Send the content of MDR to the ALU as the second operand - Calculate Reg[rs]- Mem[Reg[rt]] (ALU operations take one cycle)
Cycle 5	- Write back the result (ALUOut) to the rd register

b) Specify the operations done in each cycle in the table below. (6 marks)



Question 5: MIPS Recursion (15 marks)

Fill in the blanks below to implement a recursive Fibonacci function in MIPS code. The C definition of the function is given below in the comments of the MIPS program. You can use any of the instructions from Appendix 1, including pseudo-instructions.

```
fibo:
 #$a0 = n, $v0 = fibo(n)
 #if (n==0) return 0;
 #if (n==1) return 1;
 #return( fibo(n-1)+fibo(n-2) );
 #Step1: save preserved registers in the stack if needed
 1. addi $sp,$sp,-12
 2. sw $ra,0($sp)
 3. sw $s0,4($sp)
 4. sw $s1,8($sp)          # $s0, $s1 are optional
 #Step 2: check the base case
 #Jump to return0 if n==0, jump to return1 if n==1
 5. add $s0,$a0,$zero
 6. addi $t1,$zero,1
 7. beq $s0,$zero,return0
 8. beq $s0,$t1,return1

 #Step 3: handle the recursive case
 9. addi $a0,$s0,-1
 10. jal fibo
 11. add $s1,$zero,$v0      #s1=fibo(n-1)

 12. addi $a0,$s0,-2
 13. jal fibo               #v0=fibo(n-2)
 14. add $v0,$v0,$s1        #v0=fibo(n-2)+$s1


 #Step 4: restore the registers from stack and fib exits
exitfib:
 15. lw $ra,0($sp)         #read registers from stack
 16. lw $s0,4($sp)
 17. lw $s1,8($sp)
 18. addi $sp,$sp,12       #bring back stack pointer
 19. jr $ra

 return1:
  li $v0,1
  j exitfib
 return0:
  li $v0,0
  j exitfib
```

Question 6: MIPS Programming (10 marks)

a) A single precision IEEE 754 number is stored in memory at address labelled X. Write the shortest sequence of MIPS instructions to multiply this number by 2, and store the result back at memory address X. Accomplish this without using any floating point instructions. You can assume no overflow happens. (4 marks)

```
la $s0, X           #load the address of label X in $s0
lw $t0, 0($s0)      # load the floating point number from memory
lui $t1, 0x0080     # mult by 2 means add 1 to the exponent
addu $t0, $t0, $t1     # adds effectively 1 to the exponent
sw $t0, 0($s0)         # store the floating point number back to
memory
```

b) Write a MIPS procedure to detect if overflow took place or not for the addition of two unsigned integers stored in registers `$t1` and `$t2`.

i) Write the condition under which overflow occurs for the addition of `$t1` and `$t2` (2 marks)

$$\$t1 + \$t2 > 2^{32}-1$$

Other correct answers:
$t1 + $t2 < $t1 or $t1 + $t2 < $t2

($t1 + $t2) - $t1($t2) neq $t2($t1)

ii) Provide a sequence of MIPS instructions to detect such overflow. (4 marks)

Based on c, the condition of overflow for the addition of two unsigned numbers leads to

$\$t2 > 2^{32} - \$t1 - 1 = \text{not } \$t1$

So we have

```
addu $t0, $t1, $t2      # $t0 = sum
nor  $t3, $t1, $zero    # $t3 = Not $t1

sltu $t3, $t3, $t2      # $t3 = 1 if $t2 > not $t1 (unsigned operation
                        # Because bit 31 of one or both operands may be 1
bne  $t3, $zero, Ovfl   # Goto Overflow
```

Question 7: Pipelining (15 marks)

a) The pipelined datapath that we have discussed in class is broken down into 5 steps: instruction fetch (IF), decode and register read (ID), ALU operation (EX), memory access (MEM) and result write back (WB).

Assuming each step taks a number of picoseconds (10^{-12} s) to finish as follows:

IF	ID	EX	MEM	WB
305ps	275ps	280ps	305ps	250ps

i) What would be the clock cycle duration for this datapath if we want to build an efficient pipeline and why (2 marks)

Answer: clock cycle needs to be long enough to cover the slowest stage, so 305ps

ii) In this datapath, assuming there are neither hazards nor stalls (ideal case), how long does it take to execute an instruction? (2 marks)

Answer:
1 instruction would need to proceed through all 5 stages of the pipeline
So 305*5 =1525 ps

iii) Assuming N add instructions are executed. If they don't have any data dependencies, what is the speedup of a pipelined implementation when compared with a multi-cycle implementation with the same number of stages? Your answer should be an expression that is a function of N. (3 marks)

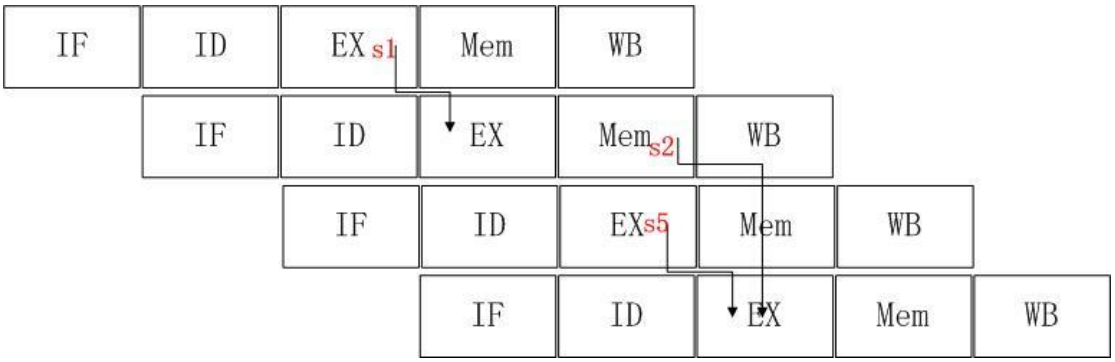
Answer:
For the multi-cycle approach:
- Each add instruction would take 4 clock cycles and each clock cycle would take 305 ps.
- Thus, the total time would be: 1220(N)

For the pipelined approach:
- For N instructions, we can apply the formula: NT + (S-1)T
- Thus, the total time would be:
  o = 305(N) + (5-1)(305)
  o = 305N + 1220 ps

Thus, the overall speedup is:
1220(N) / [305(N) + 1220]

b) Assume we execute the following instruction sequence in the pipeline. Our pipeline uses all possible forwarding from any stage to any stage (provided it does not violate the timing consistency). Fill in the table below with the appropriate pipeline stages (IF, ID, EX, Mem, WB) or bubbles (BUB). (4 marks)

	1	2	3	4	5	6	7	8	9	10	11
`sub $s1, $t0, $t1`	IF	ID	EX	Mem	WB						
`add $s2, $s0, $s1`		IF	ID	EX	Mem	WB					
`add $s5, $s3, $s4`			IF	ID	EX	Mem	WB				
`add $s6, $s5, $s2`				IF	ID	EX	Mem	WB			



------ End of Exam Paper ------

Appendix 1: MIPS instructions 1

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

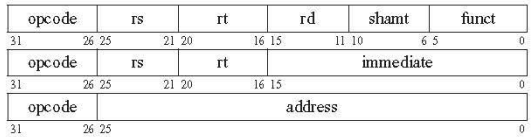
MIPS Reference Data ①

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

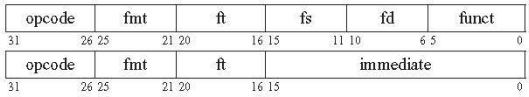
BASIC INSTRUCTION FORMATS



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

Appendix 2: MIPS instructions 2

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

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

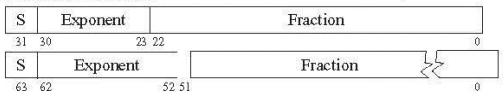
where Single Precision Bias = 127, Double Precision Bias = 1023.

④ IEEE 754 Symbols

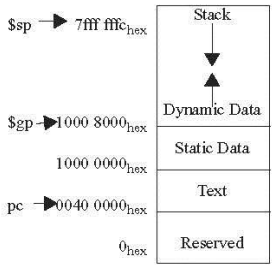
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

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

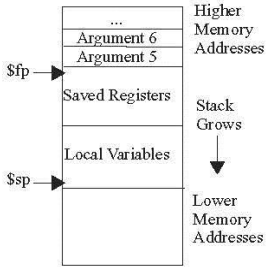
IEEE Single Precision and Double Precision Formats:



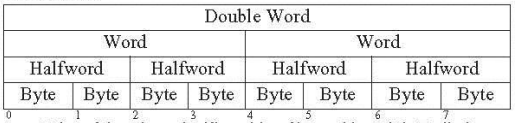
MEMORY ALLOCATION



STACK FRAME

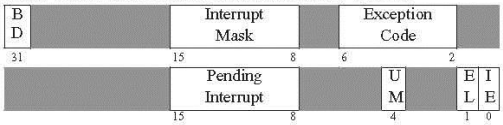


DATA ALIGNMENT



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	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
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