NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

MAKE-UP ONLINE QUIZ
AY2019/2020 Semester 2

CS2100 — COMPUTER ORGANISATION

25 March 2020 Time Allowed: **1 hour 40 minutes**

INSTRUCTIONS

1. This question paper contains **FIVE (5)** questions and comprises **FIVE (5)** printed pages. (Question 0 is on the Answer Sheet).
2. The last page is MIPS reference sheet.
3. Answer **ALL** questions within the space provided on the Answer Sheet.
4. Please type **ALL** your answers. If you are writing your answers, ensure that your handwriting is legible, or marks may be deducted.
5. Submit only the Answer Sheet.
6. Maximum score of this quiz is **40 marks**.

——— END OF INSTRUCTIONS ———

Extending MIPS Language

MIPS reference sheet is attached at the end of this quiz. Our aim here is to extend MIPS language with several instructions.

Question 1: Uncommon Instructions *[8 marks]*

Consider the two instructions `ulw` and `usw`. These instructions are not commonly used in our module. In fact, our processor cannot interpret any of the two instructions. To make `ulw` and `usw` works in our processor implementation, we will need to convert them into other instructions.

Consider `ulw $rt, offset($rs)`. There are 4 cases to consider based on the value of `R[$rs] + offset`:

- `(R[$rs] + offset) % 4 == 0`: The instruction can be replaced with `lw`.
- `(R[$rs] + offset) % 4 == 1`: ?.
- `(R[$rs] + offset) % 4 == 2`: ?.
- `(R[$rs] + offset) % 4 == 3`: ?.

Consider only the case of `(R[$rs] + offset) % 4 == 3`. Replace the following code with MIPS code without `ulw` and `usw` but only `lw` and `sw`. You may also use `lb`, `sb`, `lhw`, or `shw`. You are guaranteed that the value of `(R[$rs] + offset) % 4` is going to be exactly 3 for the codes below. You may use temporary registers. Your code should have at most 6 MIPS instructions.

a) `ulw $t0,  7($t1)` [*4 marks*]
b) `usw $t9, -2($t8)` [*4 marks*]

Question 2: Pseudo-Instructions *[8 marks]*

MIPS compiler typically has a lot of pseudo-instructions available that will eventually be converted into actual MIPS code. Two of them are introduced below.

i. `abs $rd, $rs`
`abs` puts the absolute value of the integer from register `$rs` into register `$rd`.

ii. `rdiv $rem, $div, $rs, $rt`
`rdiv` computes both the integer division and remainder at the same time. The `$rem` stores the result of `$rs % $rt` and `$div` stores the result of `$rs` / `$rt` where the division is an integer division.

Write an equivalent MIPS code for the following two instructions involving `abs` and `rdiv`. You may assume that whatever label you write will be unique. You may use temporary variables. You don't have to worry about the number of instructions.

a) `abs $t1, $s0` [*4 marks*]
b) `rdiv $t0, $t1, $s0, $s1` [*4 marks*]

Question 3: Encoding *[8 marks]*

Consider the pseudo-instruction `abs` in the previous question. We wish to make this pseudo-instruction to be an actual instruction supported by our MIPS interpreter. There are three types of instructions discussed in lecture. They are R-format, I-format, and J-format.

a) Which instruction format should `abs` be included into such that it shares as many fields as possible? Briefly justify your answer. [*2 marks*]
b) Explain why other formats are not a good match to include `abs` instruction into. [*3 marks*]
c) Consider your format in part (a). Assume all unused fields will be filled with bit 1. Encode `abs $t1, $s0` in your encoding scheme. Write your answer as ***hexadecimal***. [*3 marks*]

Question 4: Datapath and Control ***[16 marks]***

To incorporate `ulw` and `usw` into our processor implementation, we will need to modify the Data Memory component of our processor. In particular, the data memory must be able to retrieve unaligned memory. Processor control and datapath are unchanged.

On the other hand, there will be many changes that is required on both datapath and control to implement `rdiv` instruction. We will discuss the changes necessary on the processor datapath to include `rdiv` instruction correctly into our processor implementation. Our processor will still have to be *single-cycle*.

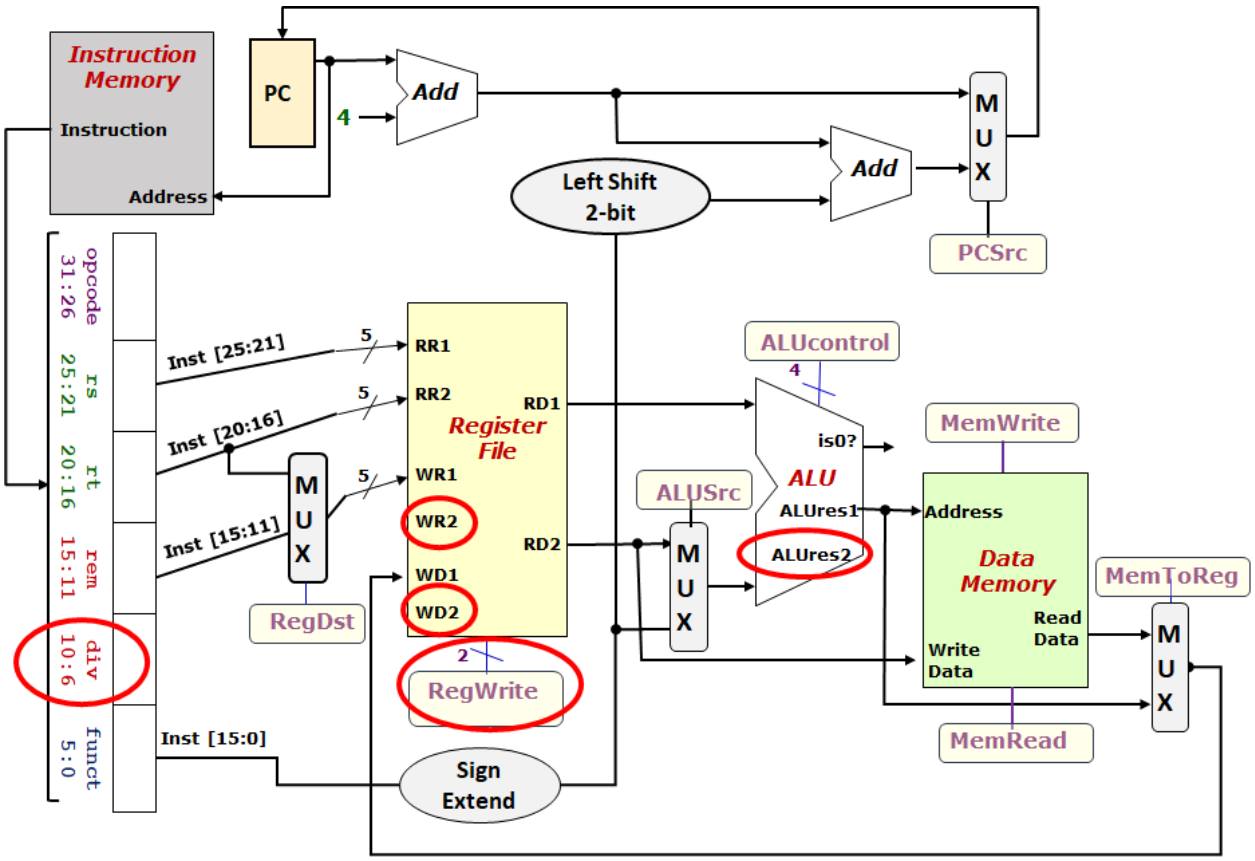
Our instruction format for `rdiv` instruction uses the following fields:

opcode	$rs	$rt	$rem	$div	funct

The opcode and funct for `rdiv` are both `0x3F`.

a) Encode the instruction `rdiv $t0, $t1, $s0, $s1`. Write your answer in ***hexadecimal***. [*2 marks*]

For the rdiv instruction to be incorporated into our processor implementation, we will need to add the following changes to our processor implementation. The summary of the changes describing all circled component is described below.



- The instruction will have `div` instead of `shamt`.
- The Register File component will have to add 2 additional inputs called `WR2` and `WD2`. Additionally, `WD` will need to be renamed `WD1`.

- The `RegWrite` control signal for Register File will have to be modified to be 2-bits instead of 1-bit. The meaning of `RegWrite` will be as follows:
  - `00`: Not writing into both `WR1` and `WR2`
  - `01`: Write `WD1` into `WR1` but not writing into `WR2`.
  - `10`: Write `WD2` into `WR2` but not writing into `WR1`.
  - `11`: Write `WD1` into `WR1` and write `WD2` into `WR2`.
- The ALU component will have to add an additional result output called `ALUres2`. The original output `ALUresult` is renamed into `ALUres1`.
- The behavior of `ALUres1` should be identical to `ALUresult` for instructions other than `rdiv`.
- `ALUres2` may give any value for instructions other than `rdiv`.

b) Describe how `WR2` is going to be connected without using any additional multiplexer. You may use existing multiplexer without changing the control signal (*shown in part (d)*). You should write which component needs to be connected to `WR2`. [*2 marks*]

c) Describe how `ALUres2` is going to be connected without using any additional multiplexer. You may use existing multiplexer without changing the control signal (*shown in part (d)*). You should write which component needs to be connected to `ALUres2`. [*2 marks*]

d) Complete the output of the control signal for `rdiv` instruction below. Some are already filled in for you. [*4 marks*]

	RegDst	ALUsrc	MemToReg	RegWrite	MemRead	MemWrite	Branch
R	1	0	0	01	0	0	0
lw	0	1	1	01	0	0	0
sw	X	1	X	00	0	1	0
beq	X	0	X	00	0	0	1
rdiv					0	0	0

Use the following convention for your answer in part (e) below.

- Given a register `$r`, the content of the register `$r` is given as `R[$r]`.
- Given a memory location `addr`, the content of the memory location at `addr` is given as `M[addr]`.
- The notation `A + B` is used to denote the arithmetic + operation where the operands are `A` and `B`.
- The notation `A - B` is used to denote the arithmetic - operation where the operands are `A` and `B`.
- The notation `A / B` is used to denote the arithmetic / operation where the operands are `A` and `B`.
- The notation `A % B` is used to denote the arithmetic % operation where the operands are `A` and `B`.
- The notation `~A` is used to denote the bitwise NOT operation where the operand is `A`.
- `PC` is used for the special register holding the address of the current instruction.
- Use decimal values for constants

e) Fill in the table on the answer sheet for the input/output value of each component in the datapath of the processor when executing the instruction `lw $t0, 128($t1)`. `RR1` and `RR2` have been filled for you. Use `X` for unknown value/don't care. [*6 marks*]

RR1	RR2	WR1	WR2	WD1	WD2	OP1	OP2	addr	MWD
$t1	$t0								

=== END OF QUESTION ===

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

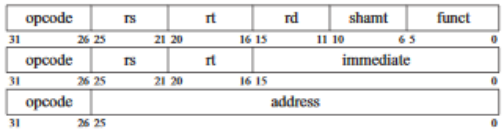
①

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

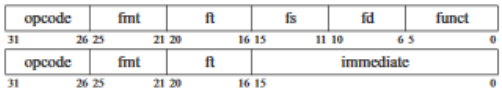


ARITHMETIC CORE INSTRUCTION SET

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