

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

ONLINE QUIZ AY2019/2020 Semester 2

CS2100 — COMPUTER ORGANISATION

11 March 2020

Time Allowed: **1 hour 40 minutes**

INSTRUCTIONS

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

$$\frac{25}{40} \quad 62.5\%$$

——— **END OF INSTRUCTIONS** ———

SIMP Language

SIMP is a 16-bit general-purpose register architecture processor with the specification below. Registers `$rs`, `$rt`, and `$rd` are placeholders for actual general-purpose registers `$1`, `$2`, ..., `$6`, each holding a 16-bit value. `const` refers to a constant value and `label` refers to label in the instruction corresponding to a specific line in the code. All **constants** are given as 4 bits 2's complement signed values. All **labels** will be converted into actual addresses using direct addressing mode.

SIMP can only accommodate up to 128 addressable memory where each memory location holds 1 byte. The memory locations are numbered from `00000002` to `11111112`. All memory locations are given as unsigned binary values. Additionally, each word in this processor consists of 2 bytes (16 bits) and every instruction is word-aligned.

We will use the following convention to simplify our discussion:

- 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 bitwise AND operation where the operands are `A` and `B`.
- The notation `A ^ B` is used to denote the bitwise XOR 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.

Round brackets `()` are used to disambiguate order of operations.

Addressing Architecture:	General-Purpose Register
Number of General-Purpose Registers:	Six (<code>\$1</code> , <code>\$2</code> , ..., <code>\$6</code>)
Special Registers (<i>addressable</i>): <i>These registers cannot be written</i>	<code>R[\$0] = 0x0000</code> [i.e., all 0s] <code>R[\$7] = 0xFFFF</code> [i.e., all 1s]
Special Register (<i>non-addressable</i>):	Program Counter (<code>\$pc</code>)
Instruction Format:	Fixed length 16-bit instructions
Arithmetic Instructions: <i>These are arithmetic operations and they must come with 3 operands</i>	<ul style="list-style-type: none"> <code>ADD \$rd, \$rs, \$rt, const</code> <ul style="list-style-type: none"> <code>R[\$rd] = R[\$rs] + R[\$rt] + const</code> <code>SUB \$rd, \$rs, \$rt, const</code> <ul style="list-style-type: none"> <code>R[\$rd] = (R[\$rs] - R[\$rt]) - const</code>
Logical Instructions: <i>These are bitwise operations and they must come with 3 operands</i>	<ul style="list-style-type: none"> <code>AND \$rd, \$rs, \$rt, const</code> <ul style="list-style-type: none"> <code>R[\$rd] = R[\$rs] & R[\$rt] & const</code> <code>XOR \$rd, \$rs, \$rt, const</code> <ul style="list-style-type: none"> <code>R[\$rd] = R[\$rs] ^ R[\$rt] ^ const</code>
Load/Store Instructions: <i>All addresses are byte addresses and they are word-aligned</i>	<ul style="list-style-type: none"> <code>LW \$rt, \$rs, const</code> <ul style="list-style-type: none"> <code>R[\$rt] = M[R[\$rs] + const]</code> <code>SW \$rt, \$rs, const</code> <ul style="list-style-type: none"> <code>M[R[\$rs] + const] = R[\$rt]</code>
Branch Instruction: <i>This is a compare and branch instruction</i>	<ul style="list-style-type: none"> <code>BEQ \$rs, \$rt, label</code> <ul style="list-style-type: none"> Branch to label if <code>R[\$rs] == R[\$rt]</code> Otherwise, go to next instruction

Question 1: Warmup Questions**[8 marks]**

Consider 4 statements written in C below with the following variable-to-register mapping:

x: \$1	y: \$2	z: \$3
--------	--------	--------

1100 ^ 1111 ^ 0 = 0011 ^ 0000
= 0011

- 1) `x = y * 2;`
- 2) `x = ~y;`
- 3) `x = y + z - 3;`
- 4) `x = y - 2;`

XOR \$1, \$2, \$7, 0

Answer the following questions below. Each answer must be a single SIMP instruction.

- a) How do we perform instruction (1) in SIMP? **Add \$1, \$2, \$2, 0** [2 marks]
- b) How do we perform instruction (2) in SIMP? ~~Xor \$1, \$2, \$2, 0~~ [2 marks]
- c) How do we perform instruction (3) in SIMP? **Add \$1, \$2, \$3, -3** [2 marks]
- d) How do we perform instruction (4) in SIMP? **Sub \$1, \$2, \$0, 2** [2 marks]

Question 2: Compilation**[8 marks]**


Using the given SIMP instruction set, complete the SIMP equivalent of the C code below on the answer sheet. Each array element is two bytes long. Ensure that your code is properly commented to ease understanding.

```
sum = 0;
for (i=0; sum==0; i++) {
    switch (x[i]) {
        case 0: x[i] = -1;
        case -1: break;
        default: sum = x[i] + i + 5;
    }
}
```

Assume that you have the following variable-to-register mapping.

sum: \$1	i: \$2	base address of x: \$3
----------	--------	------------------------

Recap that in C, switch-case statement will continue execution from one case to the next unless there is a **break**. In particular, for the code above, **case 0** will “spill over” to **case -1**. Additionally, any **break** statement within switch-case will exit the switch-case only. You are already given the first 4 lines as well as the last 2 lines of the SIMP code. Your task is to fill in the middle section. You are not to modify the given lines of code on the answer sheet.

<div style="border: 1px solid black; padding: 5px; width: 30px; text-align: center;">  </div>	<pre> add \$1, \$0, \$0, 0 #sum = 0 add \$2, \$0, \$0, 0 # i = 0 beq \$1, \$0, cont #while (sum == 0) continue beq \$0, \$0, end #otherwise end </pre>
<pre> cont: ADD\$4,\$2,\$2,0 # array element is 2 bytes, so 2i ADD\$4,\$4,\$3,0 # \$4=addr of x[i] LW\$5,\$4,0 # \$5=x[i] BEQ\$5,\$0,case0 #switch(x[i])-->case0: BEQ\$5,\$7,add #switch(x[i])-->case-1:(add) ADD\$1,\$5,\$2,5 #default: sum=x[i]+i+5 BEQ\$0,\$0,add #unconditional branch to add case0: SW\$7,\$4,0 #x[i] = -1 add: ADD\$2,\$2,\$0,1 #i = i + 1 </pre>	<pre> break: BEQ \$0, \$0, loop #goto loop end: </pre>

Question 3: Encoding**[14 marks]**

We can categorize the set of instructions in SIMP into three categories, with instruction formats as shown below:

Class A	Bits	3-bit	3-bit	3-bit	3-bit	4-bit
	Fields	opcode	rs	rt	rd	const
Class B	Bits	3-bit	3-bit	3-bit	3-bit	4-bit
	Fields	opcode	rs	rt	funct	const
Class C	Bits	3-bit	3-bit	3-bit	7-bit	
	Fields	opcode	rs	rt	label	

The following rules summarize how to encode a SIMP instruction to binary.

- Register is encoded as its number. For instance, \$6 is encoded as 110_2 .
- Constant is encoded as 2's complement signed value. For instance, -2 is encoded as 1110_2 .
- Label is first converted into address and the address is encoded directly. For instance, to branch to address 0011010_2 , the label field is specified as 0011010_2 .
- **funct** is not used in this encoding, it will always be 0 in question 3a. It is reserved for future use (see question 3b).
- Opcode is encoded using the table below where the value is given as *decimal*, then converted to binary. For instance, for LW instruction, the opcode is 5, the encoded value is 101_2 .

Instruction	Opcode	Instruction	Opcode
ADD	1	LW	5
SUB	2	SW	6
AND	3	BEQ	7
XOR	4		

- a) Consider the following SIMP code which has been partially encoded. Assume that the first instruction is at address 0000000_2 . Fill in the missing hexadecimal encodings or the instructions on the answer sheet.

Hexadecimal	SIMP	
0x402F	010 000 000 010 1111 = SUB \$2, \$0, \$0, -1	[2 marks]
0010 1001 0010 0000 = 0x2920	L: ADD \$2, \$2, \$2, 0	[2 marks]
0x241F	001 001 000 001 1111 = ADD \$1, \$1, \$0, -1	[2 marks]
0xE08A	111 000 001 0001010 = BEQ \$0, \$1, E	[2 marks]
1110 0100 1000 0010 = 0xE482	BEQ \$1, \$1, L	[2 marks]
	E:	

\downarrow func
 3, 3, 3, 3, 4
 000 x x 2^3 x

only have opcode: 000

b) Suppose besides the 7 instructions given (ADD/SUB/AND/XOR in class A, LW/SW in class B and BEQ in class C), you want to add more instructions into the SIMP instruction set. Assuming the encoding space is completely utilized, *including* func, and note that the opcodes of the given 7 instructions should not be used for other instructions. What is the

$$2^3 + 7 = 15$$

(i) maximum total number of instructions (including the 7 given instructions)? [2 marks]

(ii) minimum total number of instructions (including the 7 given instructions)? [2 marks]

You do not need to show workings for the above.

Let 000 = class w/o func $\therefore 7+1=8$

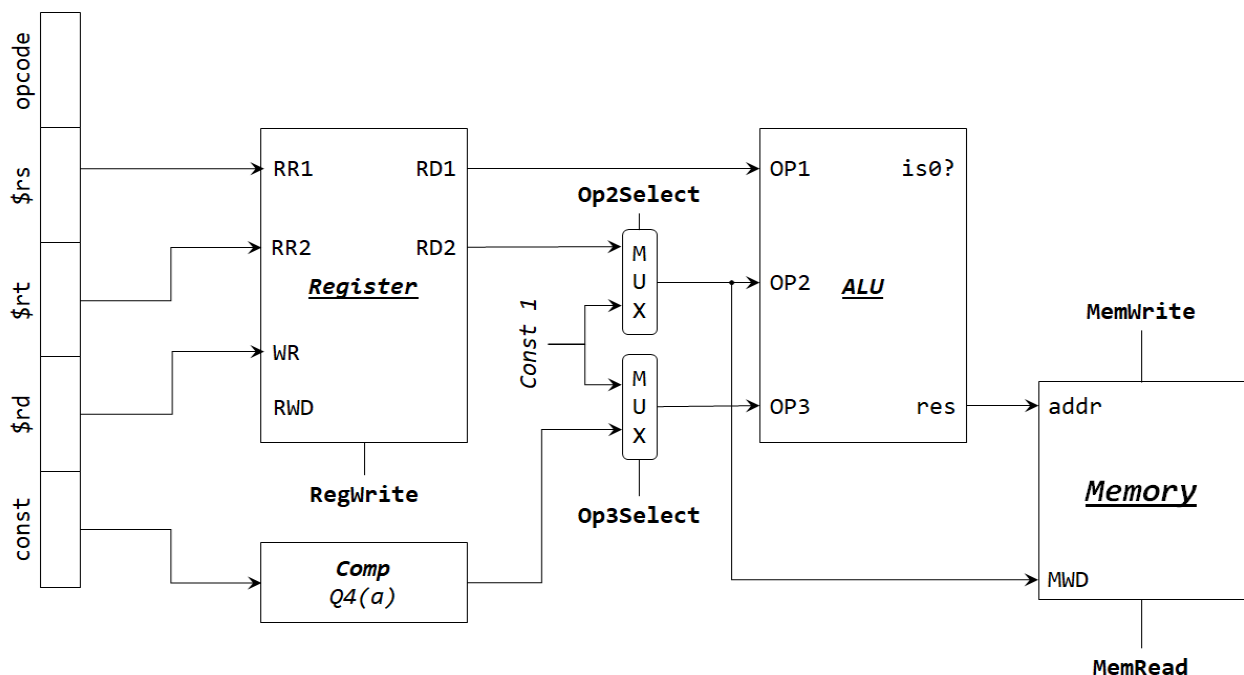
$$2 \times 2^3 - 2 + 7 = 8$$

Question 4: Datapath and Control

[9 marks]

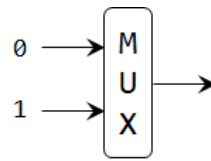
Study the datapath of the processor below. It partially implements the fetch, decode, and ALU stage for the SIMP language. Const 1 is to be determined in part (b). The ALU is special as it can accept 3 operands instead of the usual 2. Operations in the ALU require all operands to have 16-bit value. The implementation is currently limited to the store-to-memory stage.

The control signals are summarized in the table below.



Instruction	Op2Select	Op3Select	MemRead	MemWrite	RegWrite
ADD	0	1	0	0	1
SUB	0	1	0	0	1
AND	0	1	0	0	1
XOR	0	1	0	0	1
LW	1	1	1	0	1
SW	1	1	0	1	0
BEQ	0	0	0	0	0

Note that for the multiplexer, the input line is selected as follows:



Since the inputs to the ALU must be 16-bit values, we will need the component marked as **Comp**. Additionally, for **BEQ** operation, the input to **OP3** of ALU must be chosen from either **label** or **Const 1**. Similarly, for **LW** and **SW** operation, the input to **OP2** of ALU must be chosen from either **\$rt** or **Const 1**.

For **BEQ**, the operation performed by the ALU is subtraction (i.e., $OP1 - OP2 - OP3$). For **LW** and **SW**, the operation performed by the ALU is addition (i.e., $OP1 + OP2 + OP3$).

Given the information above, answer the questions below.

- a) What is the component marked as **Comp**? **sign extension** [1 mark]
- b) What is the value of **Const 1**? Give your answer in **hexadecimal**. **0x0000** [1 mark]

Consider executing the instruction **BEQ \$4, \$5, 54**. Note that **54** is already the target address and not a label anymore. The encoding for this instruction is: **0xF2B6** or **1111 0010 1011 0110₂**.

Use the following convention in your answer:

- 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 bitwise AND operation where the operands are **A** and **B**.
- The notation **A ^ B** is used to denote the bitwise XOR 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

- c) Fill in the table on the answer sheet for the input/output value of each component in the datapath of the processor given the control signal and encoding above. **RR1** has been filled for you. [7 marks]

RR1	RR2	WR	OP1	OP2	OP3	addr	MWD
\$4	\$5	\$3	R[\$4]	R[\$5]	0	R[\$4] - R[\$5]	R[\$5]

=== **END OF PAPER** ===