UNIVERSITY of HOUSTON

Computer Organization & Architecture
College of Natural Sciences and Mathematics
Fall 2020

Exam 1.

This exam is given under the University of Houston Honor Code System. You must observe and sign the Honor Pledge: "I have neither given nor received aid on this exam." Your print name and signature below signifies your compliance with this honor code. Note that we will not grade your exam unless it is acknowledged.

Student ID (last four digits) 0918

Name: SamitaDamani

By clicking the checkbox below, I acknowledge my responsibility and commitment to the Academic Honor Code.

☒ Acknowledgment

1. _____ (25 pts)
2. _____ (20 pts)
3. _____ (15 pts)
4. _____ (40 pts)

Total (100 pts) _____

Make sure you use a software that allows to fill PDF forms

- *Sejda (Web, Windows, Mac, Linux) to edit text and create PDF forms for free*
- *Foxit (Web, Android, iOS, Windows, Mac) to edit PDFs everywhere*
- *PDF Expert (iOS, Mac) to quickly edit PDF text and images*
- *PDFelement (Android, iOS,Windows, Mac) to edit PDFs and add forms in an Office-like editor*
- *Adobe Acrobat (Windows, Mac) to create detailed PDFs and forms*

How to draw on a PDF in Adobe Acrobat Reader
https://www.youtube.com/watch?v=E_3N1foNosg
How to insert Image in PDF file using Foxit Reader's Image
https://www.youtube.com/watch?v=c6oWUkYNGZw

Make sure you save your answers (doublecheck before submit). In addition, doublecheck that you have submitted your filled pdf. If you have a separate file for Q1, please make sure that it is uploaded as well.

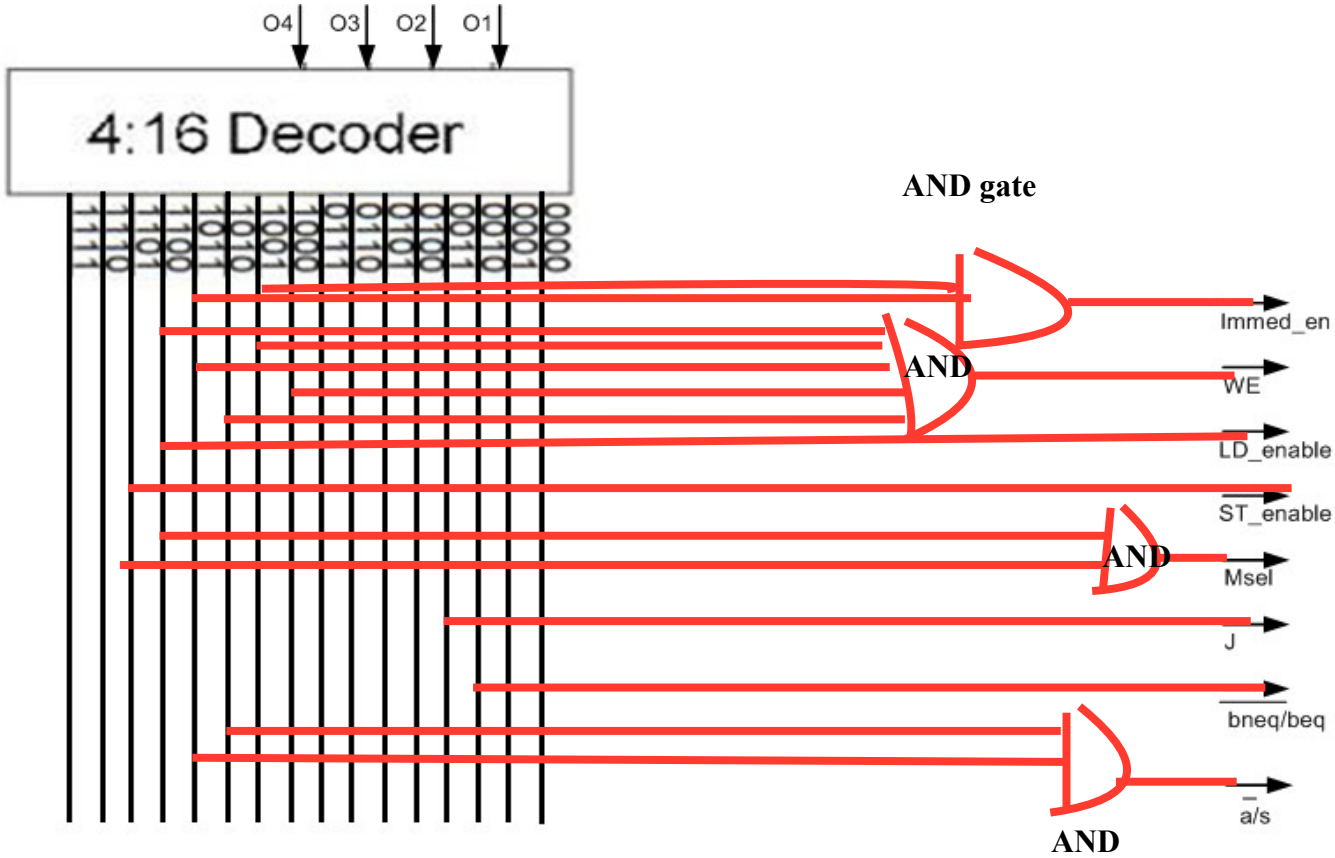
```
cd lab0-your-github-account-name/exams/
git add "*"
git commit -m "Exam1"
git push origin master
```

Budget your time wisely

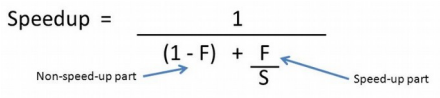
1. Decode logic (25 pts). When an instruction is executed by a processor, the instruction is decoded into a set of control signals that are used to control the various processor components (e.g., register file, ALU) and data path. This decoding task can be implemented using either a ROM or decoders.

You are asked to implement such a decoder for a simplified MIPS instruction set, called mini-MIPS. Mini-MIPS uses 4 bit opcode (O1-O4) and supports 9 instructions. The table below shows all the 9 instructions and their opcodes. For each instruction, the table also lists the corresponding values of eight control signals. Please design a decoder for mini-MIPS using decoders and gates and show your logic circuits schematic. **You can draw in PDF using annotation, or insert your drawing. Or you can upload your drawing as a separate file to github. If you upload a separate file, name it as exam1_q1.png. Please label your gates (OR, AND) in the drawing.**

Instruction	O4	O3	O2	O1	Immed_en	WE	LD_enable	ST_enable	Msel	J	___ bneq/beq	_ a/s
LW	1	1	0	0	0	1	1	0	1	0	0	0
SW	1	1	0	1	0	0	0	1	1	0	0	0
ADDI	1	0	0	1	1	1	0	0	0	0	0	0
SUBI	1	0	1	1	1	1	0	0	0	0	0	1
ADD	1	0	0	0	0	1	0	0	0	0	0	0
SUB	1	0	1	0	0	1	0	0	0	0	0	1
J	0	0	1	1	0	0	0	0	0	1	0	0
BEQ	0	0	1	0	0	0	0	0	0	0	1	0
BNEQ	0	0	0	1	0	0	0	0	0	0	0	0



3. Amdahl's Law (20 pts). A program takes 5 days execution time on current machine



25% of time doing integer instructions
40% percent of time doing I/O (input/output)

Which one below is the better tradeoff?

A. Compiler optimization that reduces number of integer instructions by 40% (assume each integer inst takes the same amount of time)
B. Hardware optimization that reduces the latency of each I/O operations from 8us to 5us.

Show your calculation using Amdahl's law

Compiler optimization overall speedup (show your calculation)

Amdahl'slaw=1/(1-f)+f/pf=0.25reduceINTinstructionby40%.1/1-0.4=5/3Speedup=(1/(1-0.25)+0.25/(5/3))=10/9=1.11

Speedup of A (result) = 1.11

I/O optimization overall speedup (show your calculation)

f=0.4.8us/5us=1.6latencySpeedup=1/(1-0.4)+(0.4/1.6)=20/17=1.176470588

Speedup of B (result) = 1.18

Which is the better tradeoff?

☐ A is better ☒ B is better

3. CPI (15 pts). What is the average CPI of a 2.5 GHz machine that executes 8 billion instructions in 10 seconds?

Show your calculation

Instructionspersecond=clockrate/CPI..CPI=clockrate/IPS=(2.5*10^9)/(8*10^9/10))=25/8=3.125CPI

CPI = 3.125

4. 8-bit CPU Datapath (40 pts). The 8-bit CPU has an ISA in the Appendix. The table next page lists the instructions. The are 14 control signals. ALUCtrl has 8 bits using the same definitions in Logisim ALU lab (Logisim Lab 4).

ALUCtrl (see also Logisim Lab 4: 8-Bit ALU – page 2)								
ALS1	ALS0	A/S	LF1	LF0	ST	SD	SA	
0	0	x	x	x	x	x	x	beq
0	1	x	0	0	x	x	x	and
0	1	x	0	1	x	x	x	ori, or
0	1	x	1	0	x	x	x	not
0	1	x	1	1	x	x	x	forward input (can be used to realize mov2h, mov2l, disp)
1	0	0	x	x	x	x	x	add, lb, sb
1	0	1	x	x	x	x	x	sub, subi
1	1	x	x	x	0	0	0	sll
1	1	x	x	x	0	1	0	srl
1	1	x	x	x	1	1	0	sra
1	1	x	x	x	0	0	1	lui

x for field that doesn't matter.

Rdsel and Rxsel select one of the four registers where the MSB indicates group number. Rdsel specifies destination register of an instruction. WBDatsel selects data source used to update the selected destination register (by Rdsel).

	0	1	2	3
WBDatsel (2bits)	Use ALU output	Use RAM output	Update for dech and inch instructions	Use next PC to support jal

Immsel (1bit)	PCselbeq (1bit)	PCseljmp(1bit)	PCseljr (1bit)	RAMload(1bit)	RAMwrite(1bit)	ALUenable (1bit)	Regwrite (1bit)	Addr (5bit) – jump, beq	Imm (4bit)	Disp (1bit)
Set for lui, ori, lb. sb	Set for beq	Set for jump	Set for jr	Set when instruction loads data from RAM	Set when instruction updates RAM	Set when ALU is used by the instruction	Set when a destination register is updated	5 bit jump target address or beq offset	Immediate value for lui, ori, lb. sb	Set for disp instruction

Note: ALU is used to compute memory location for lb and sb. See definitions below.
lb: $rd = MEM[imm+$rx]
sb: MEM[imm+$rx] = $rd

8-bit CPU control signals (complete rows with first column in red) and opcode. **To make grading easier, except AluCtrl, Rdsel, Rxsel, WBDatsel, for other columns, please skip entries that do not care. For AluCtrl, Rdsel, Rxsel, WBDatsel, put x for bits that do not care.**

Location	Instruction	AluCtrl (8bits)	Rdsel (2bits)	Rxsel (2bits)	WBDatsel (2bits)	Immsel (1bit)	Pcselbeq (1bit)	PCseljmp (1bit)	PCseljr (1bit)	RAMload (1bit)	RAMwrite (1bit)	ALUenable (1bit)	Regwrite (1bit)	Addr (5bit) – jump, beq	Imm (3 or 4bit)	Opcode (3bit)
0	lui $r0, 0															
1	ori $r0, 4															
2	sb $r0, $r0, 0															
3	lui $r0, 2															
4	ori $r0, 7	01x01xxx	00	00	00	1						1	1		0111	011
5	sb $r0, $r0, 2															
6	lui $r0, 0															
7	mov2h $r1, $r0															
8	lb $r0, $r1, 3															
9	mov2h $r0, $r0															
10	lui $r1, 1															
11	ori $r1, 8															
12	lb $r0, $r1, 1	100xxxxx	00	10	01	1				1		1	1		0001	010
13	jal $r1, $r1	xxxxxxxx	10	00	01			1						00001		110
14	lui $r1, 0															
15	mov2h $r1, $r1															
16	sb $r0, $r1, 2															
17	lui $r1, 0															
18	disp $r0, $r1															

Location	Instruction	AluCtrl (8bits)	Rdsel (2bits)	Rxsel (2bits)	WBDasel (2bits)	Immsel (1bit)	Pcselbeq (1bit)	PCseljmp (1bit)	PCseljr (1bit)	RAMload (1bit)	RAMwrite (1bit)	ALUenable (1bit)	Regwrite (1bit)	Addr (5bit) – jump , beq	Imm (3 or 4bit)	Opcode (3bit)
19	lui $r0, 0															
20	beq -1	00xxxxxx	x	x	x		1					1		11111		111
21	beq -1															
22	beq -1															
23	beq -1															
24	mov2l $r1, $r1															
25	sb $r1, $r0, 0															
26	dech $r0															
27	lui $r1, 0															
28	ori $r1, 1															
29	beq 8															
30	sub $r0, $r1	101xxxxx	00	01	00							1	1			000
31	lui $r1, 1															
32	ori $r1, 8															
33	jal $r1, $r1	xxxxxxxx	10	00	01			1						00001		110
34	sll $r0, $r0	11xxx000	00	00	00							1				000
35	lui $r1, 0															
36	ori $r1, 1															
37	add $r0, $r1															
38	lb $r1, $r0, 1															
39	inch $r0															
40	jr $r1	xxxxxxxx	10	00	01			1						00001		110

Appendix

8-bit CPU ISA

The 8-bit CPU is based on an ISA below and uses only fours registers ($r0, $r1, $r2, and $r3). It will have separate data and instruction memory. The four registers are organized in a 4x8 register file. They are divided into two groups, each with two registers. $r0 and $r1 belong to group 0, and $r2 and $r3 belong to group 1. Within each group, a single bit (LSB) is used to index the two registers ($r0, or $r1). Another bit (MSB) is used as group index. For instance, 11 means register $r1 in group 1.

Register $r0 and $r1 (group 0 registers) can be used by R-type instructions (opcode 0). Another set of instructions (opcode 6) can work with register $r2 and $r3 (group 1 registers), called Special R-type instructions. Value stored in a group 0 register can be transferred to a group 1 register, and vice versa. Group 1 registers are mainly used for storing memory base address, and procedure call return address. For instance, one group 0 register can be used as stack pointer, and the other one used for storing procedure call return address.

Memory addressing uses base address plus offset format where base address is stored in a group 1 register, and offset is an immediate number. A base address can be loaded to a group 1 register by first storing it in a group 0 register, and then use mov2h instruction to transfer the value to a group 1 register.

The instruction encoding is given below. **You can determine which instruction a byte encodes by looking at the opcode (the top three bits), and funct code**. Despite simple and only working with 8 bits, the ISA supports many operations including Jr and Jal that can be applied to implement procedure call.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
opcode									
0			rd	rx	**funct**			**See R-type Instructions**	
1			rd	immediate				lui: $rd = imm << 4	rd is R0 or R1
2			rd	rx	immediate			lb: $rd = MEM[imm+$rx]	rd belongs to group 0; rx belongs to group 1
3			rd	immediate				ori: $rd = $rd | imm	
4			rd	rx	immediate			sb: MEM[imm+$rx] = $rd	rd belongs to group 0; rx belongs to group 1
5			target addr					jump	
6			rd	rx	**funct**			**See Special R-Type Definitions**	
7			offset					beq	

Special R-Type Instructions (all share the same opcode 6)

Funct	Meaning	
0	disp: DISP[$rx] = $rd	rd belongs to group 1; rx belongs to group 0
1	jr $rx	jump register, $rx belongs to group 0
2	mov2l**:** $rd = $rx	rd belongs to group 0; rx belongs to group 1 (move data from group 1 register to group 0 register)
3	RESERVED	
4	mov2h**:** $rd = $rx	rd belongs to group 1; rx belongs to group 0 (move data from group 0 register to group 1 register)
5	jal $rd, $rx	jump and link, $rx belongs to group 0, $rd belongs to group 1
6	inc: $rd = $rd + 1	rd belongs to group 1
7	dec: $rd = $rd - 1	rd belongs to group 1

R-Type Instructions (all share the same opcode 0)

Funct	Meaning
0	sll: $rd = $rx << 1
1	and: $rd = $rd ^ $rx
2	srl: $rd = $rx >> 1
3	or: $rd = $rd | $rx
4	add: $rd = $rd + $rx
5	not: $rd = ~$rx
6	sra: $rd = $rx / 2
7	sub: $rd = $rd - $rx

srl vs. sra

Just as in MIPS, srl and sra differ here by sign extension. Since sra stands for shift right arithmetic, it considers it's operand a two's complement signed number and sign extends appropriately. srl considers it's operand a set of separate logical values, and zero extends instead.

jump

The jump instruction's argument is a pseudoabsolute address, just as in MIPS. address is an unsigned number representing the lower five bits of the next instruction to be executed. The upper three bits are taken from the current PC.

```
PC = (PC & 0xe0) | address
```

beq

The beq instruction's argument is a **signed** offset relative to the next instruction to be executed normally, also as in MIPS. beq can be represented as the following:

```
if $r0 == $r1
	PC = PC + 1 + offset
else
	PC = PC + 1
```

immediate fields

All immediate fields are treated as unsigned numbers and are zero-extended accordingly.

jr $rx

$rx is a register in group 0. It stores an unsigned 8-bit number representing location of the next instruction to be executed.

```
PC = $rx
```

jal $rd, $rx

$rx is a register in group 0. $rd is a register in group 1. $rx stores an unsigned 8-bit number representing location of the next instruction to be executed.

```
$rd = PC + 1
PC = $rx
```