Department of Electrical and Computer Engineering
University of Texas at Austin

EE460N Spring 2021
Y. N. Patt, Instructor
Siavash Zangeneh, Ben Lin, Juan Paez, TAs
Exam 1
March 10th, 2021

Name:

EID:

Problem 1: 20 points
Problem 2: 15 points
Problem 3: 15 points
Problem 4: 25 points
Problem 5: 25 points
Total: 100 points

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Please read the following sentence, and if you agree, sign/print your name where requested: **I have not given or received any unauthorized help on this exam.**

Signature:

Good Luck!

General Instructions:

1. You are free to use anything in the Handouts section of the course website that is listed under "Powerpoint Presentations", "Other Handouts", "LC-3b Handouts", and "Spring 2021 Exam 1 Buzzwords." In particular, Appendix A and Appendix C may be of use. Anything else from the course website is not allowed. Anything from any textbooks or the Internet is also not allowed and considered unauthorized access.
2. Use of a calculator is not required but is permitted.
3. If you have any questions, join the class Zoom link and ask a TA. You do not need to stay on the Zoom call during the exam unless you have questions.
4. Announcements will be posted here. Check this page periodically throughout the exam.
5. You may take the exam by printing it, editing a PDF, or editing a Google Doc. Read the instructions for your preferred method below.
6. **You are required to stop working on the exam promptly at 6:30 PM.**

Printing or editing a PDF:

1. Download and save the PDF from Gradescope.
2. Edit the PDF to fill in answers with a software of your choice. Feel free to show your work in the available space. You may also choose to print the exam and solve it on paper.
3. When you are ready to submit your exam, save the edited PDF as "Exam 1 <your name>"; if you printed your exam, scan in your written answers as a PDF with the same name. You may use a scanner or an app such as CamScanner.
4. Upload the PDF to Gradescope by 6:45 PM.
5. If you fail to upload your exam to Gradescope on time, email your exam to the TAs as soon as possible. Late penalties may apply.

Editing a Google Doc:

1. Open the Google Doc version of the exam from here.
2. Save a copy of the document to your Google Drive.
3. While working on the exam, **DO NOT expand any boxes that are given to you.** Feel free to show your work in the available space. If you need more space, you are writing too much.
4. When you are ready to submit your exam, click "File"-> "Print" and select "Save as PDF". Save the edited PDF as "Exam 1 <your name>".
5. Upload the PDF to Gradescope by 6:45 PM.
6. If you fail to upload your exam to Gradescope on time, email your exam to the TAs as soon as possible. Late penalties may apply.

Problem 1 (20 points): Answer each question in 20 words or fewer. Note: For each of the four answers below, if you leave the box empty, you will receive one point of the five.

Part a (5 points): The Tomasulo Algorithm replaces the register file with a Register Alias Table. Each Register Alias Table entry consists of three items, let's call them X, Y, Z. X is a binary variable, takes on two values 0 and 1. What are Y and Z? Be specific. Either Y or Z is relevant, but not both. Under what conditions is each relevant?

Part b (5 points): We wish to design a processor that can decode 4 instructions at a time. Is it easier to do this for the x86 ISA or for the LC-3b ISA? Explain.

Part c (5 points): Profiling is a compile-time technique for determining at compile-time the best schedule (or order) to execute instructions. Sometimes profiling can produce worse results than doing nothing. When is this the case?

Part d (5 points): The LC-3b is a 3-address machine since each operate instruction explicitly identifies both source registers and the destination register. For example, ADD R1,R2,R3.

A zero-address ISA is an ISA that does not explicitly mention the source registers or the destination register for an operate instruction. For example, ADD. Only the opcode is explicitly identified. How does the microarchitecture know where to get the two source operands and where to store the sum?

Problem 2 (15 points): We want to execute a program on an in-order pipelined processor.
- It takes 1 cycle to fetch, 1 cycle to decode, 3 cycles to execute an ADD, and 1 cycle to write back the result.
- The adder is pipelined.
- Results (including condition codes) become available the cycle after the instruction writes back.
- Branches are predicted during their fetch phase, so the target instruction can be fetched in the next clock cycle.
- If the branch predictor mispredicts, the pipeline is flushed in the cycle after the condition codes are written back. The pipeline fetches the correct path in the cycle after the wrong-path instructions are flushed.
- Branch instructions do not stall the execution of subsequent ADD instructions.

The processor supports conditional execution (predication) using an ADDz instruction. ADDz always executes like a normal ADD instruction but it only writes back the result if the z condition code is 1. ADDz does not update the condition codes.

We can write the program using two approaches.

Approach A (with branch instructions)	Approach B (with predication)

<pre>   ADD   R0, R0, #0
        BRnp   SKIP
        ADD   R1, R1, #1
        ADD   R2, R2, #1
SKIP ADD   R3, R3, #1</pre>
	<pre> ADD R0, R0, #0
	        ADDz R1, R1, #1
	        ADDz R2, R2, #1
	        ADD R3, R3, #1</pre>

Answer the following questions about the two approaches:

Part a (4 points): Using approach A, suppose the branch is taken and the branch predictor correctly predicts taken. Complete the pipeline timing diagram.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

How many cycles does it take to execute the program?

Part b (4 points): Using approach A, suppose the branch is taken but the branch predictor always incorrectly predicts not taken. Complete the pipeline timing diagram. Do not show the flushed instructions.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

How many cycles does it take to execute the program?

How many cycles does it take to execute the program?

Part c (4 points): Using approach B, complete the pipeline timing diagram.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

How many cycles does it take to execute the program?

Part d (3 points): Suppose the branch is always taken, what is the minimum branch prediction accuracy such that approach A is faster than approach B? Assume we execute the program fragment many times and we care about the execution latency on average.

Hint: the average number of cycles to execute the program fragment using approach A is:

$p \times$ (cycles if correctly predicted) + ($1 - p$) $\times$ (cycles if incorrectly predicted)

where p = branch prediction accuracy

Problem 3 (15 points): In class we mentioned we sometimes wish to profile the branch behavior of programs. Modern machines have special dedicated hardware registers, called performance counters, that count hardware related activities like the number of taken branches encountered during program execution.

In this problem we will add a new memory-mapped I/O device register, the Total Taken Branches Register (TTBR), which gets incremented by the datapath on every taken BR instruction.

To use the TTBR, we
- First clear it at the start of the program by storing 0 to the memory address assigned to the TTBR. We chose the address **0xFE08** for TTBR.
- Then run the rest of the program normally. The TTBR will be incremented on every taken BR instruction
- Upon completion of the computation, the content of TTBR is stored into the memory location labeled TAKEN.

The program below is an example to help you visualize the process.

```
        .ORIG  x3000
        AND    R0, R0, #0
        LEA    R1, TTBR
        LDW    R1, R1, #0
        STW    R0, R1, #0       ; clear the Total Taken Branches Register (TTBR)

        ...the actual program   ; TTBR is incremented automatically as taken BR
                                ; instructions are executed

        LEA    R0, TAKEN
        LEA    R1, TTBR
        LDW    R1, R1, #0
        LDW    R1, R1, #0       ; read out the value of the TTBR and
        STW    R1, R0, #0       ; store it to user space memory location TAKEN

        HALT

TAKEN  .BLKW #1

TTBR   .FILL 0xFE08
        .END
```
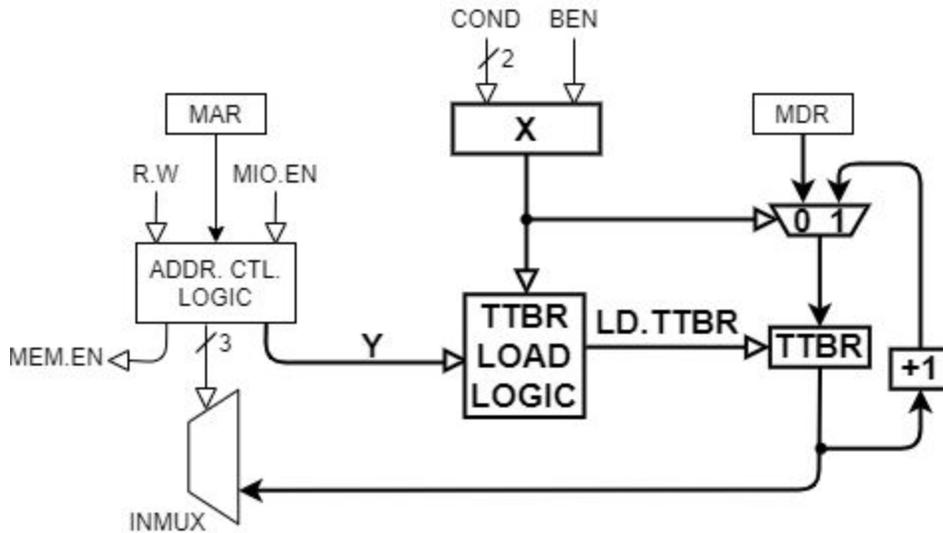
Datapath modifications needed to support the TTBR are shown below:

COND BEN

MAR

R.W   MIO.EN

ADDR. CTL.
LOGIC

MEM.EN

X

Y

TTBR
LOAD
LOGIC

LD.TTBR

MDR

0 1

TTBR

+1

INMUX

Part a (4 points):

The box labelled 'X' in the figure above contains combinational logic. Fill out its truth table. If the value does not matter, leave it blank.

COND	BEN	X
00	0	
00	1	
01	0	
01	1	
10	0	
10	1	
11	0	
11	1	

Part b (4 points):

The signal labelled 'Y' in the figure above is computed by the Address Control Logic. List all the possible inputs which results in 'Y' being a 1. You may not need all the rows provided.

MIO.EN	MAR	R.W (RD or WR)	Y
			1
			1
			1
			1

Part c (4 points):

The box labelled 'TTBR Load Logic' in the figure above contains combinational logic. Fill out its truth table. If the value does not matter, leave it blank.

X	Y	LD.TTBR
0	0	
0	1	
1	0	
1	1	

Part d (3 points):

In which state(s) is the TTBR incremented?

Problem 4 (25 points):

Multiply-Accumulate, often called a MAC operation, refers to an operation that multiplies two operands and subsequently adds the product to a third operand, as follows: $MAC(x, y, z) = x + (y*z)$

We will implement a MAC instruction for positive integers in the LC-3b as follows:

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
1 0 1 1	DR	SR1	0 0 0	SR2

After the execution of this operation, DR should contain the result of **DR + (SR1 * SR2)** and the **condition codes should be set**. To enable this functionality, we will update the LC-3b datapath and state machine as shown below. For the datapath, only the relevant portions are actually shown, and our additions are shown in bold. 'X' is a logic block and 'Y' is a constant value.
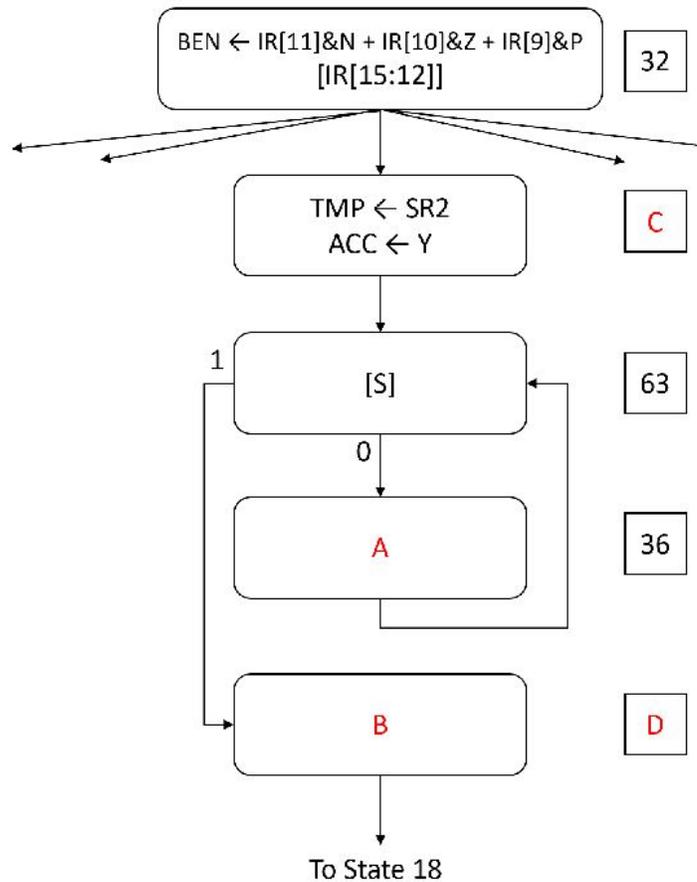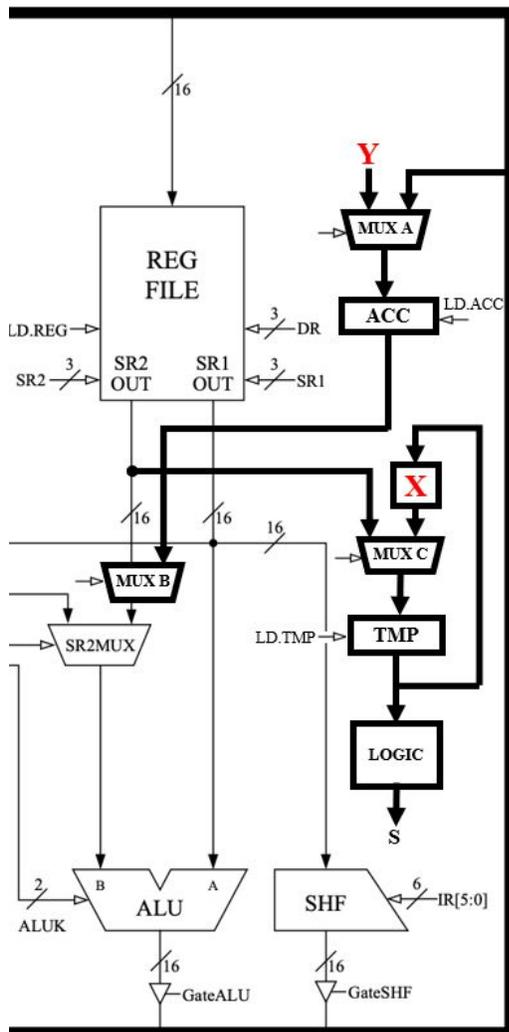
Part a (8 points):

Fill out state A:

Fill out state B:

What is state number C?

What is state number D?

Part b (6 points):

What does operation X represent? Explain.

What constant value should input Y be? Explain.

Part c (11 points): To support the branch in state 63, we need to modify the microsequencer to condition on signal S shown in the datapath and the state diagram.
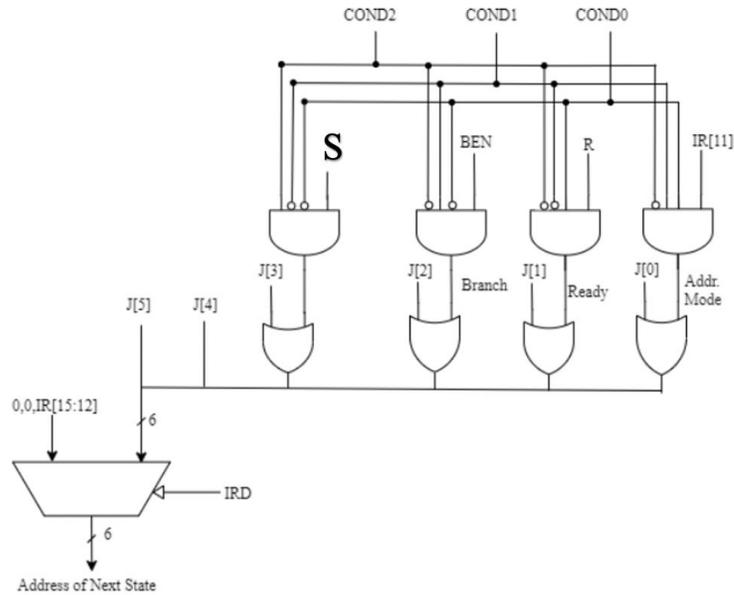
Explain what signal S is:

Complete the following table demonstrating the values taken by control signals in different states. If the value does not matter, leave it blank.

State #	LD.TMP (0 or 1)	LD.ACC (0 or 1)	MUX A (Y or Bus)	MUX B (SR2 or ACC)	MUX C (SR2 or X)	SR1MUX (IR[11:9] or IR[8:6])	COND[2:0] (000, … , 111)
C							
63							
36							
D							

Problem 5 (25 points): An out-of-order processor executes instructions based on the original Tomasulo algorithm without in-order retirement. The processor implements a 4-stage pipeline: Fetch, Decode, Execute, Writeback. The Execute stage of the pipeline contains *one pipelined adder*, *one pipelined multiplier*, and *one branch unit*.

- Fetch and Decode take one cycle each.
- The result of a functional unit is broadcast during the writeback stage and is ready for use the cycle immediately after writeback.
- *An ADD takes 3 cycles* to execute, a *MUL takes 5 cycles*, and a *BR takes 1 cycle*.
- The branch predictor predicts all the branches correctly, i.e., the processor will start fetching the correct target of the branch in the cycle immediately after it fetches the branch instruction.
- All three functional units have three-entry reservation stations that are initially empty, and are allocated in a top-to-bottom manner.
- Entries are put into reservation stations at the end of Decode and removed at the end of Writeback.
- Instructions with no dependencies can start executing immediately after Decode.

The following snippet of code is executed until the instruction at I5 has completed.

Instr #	Label	Instruction
I1	LOOP	MUL R1 , __ , __
I2		___ R2 , R1 , __
I3		BRnp LOOP
I4		___ __ , __ , __
I5		___ __ , __ , __

The state of the register alias table (RAT) is partially shown before the code starts executing, after cycle 9, and after the code completes execution.

	V	Tag	Value
R0	1	-	2
R1	1	-	4
R2	1	-	6
R3	1	-	-1
R4	1	-	9
R5	1	-	10
R6	1	-	-8
R7	1	-	-16

	V	Tag	Value
R0	1	-	2
R1	0		-
R2	0		-
R3	1	-	-1
R4	1	-	9
R5	0		-
R6	1	-	-8
R7	1	-	-16

	V	Tag	Value
R0	1	-	2
R1	1	-	
R2	1	-	
R3	1	-	-1
R4	1	-	9
R5	1	-	
R6	1	-	-8
R7	1	-	32

Before **Cycle 1**   After **Cycle 9**   After **Completion**

The state of the ADD and MUL reservation stations are partially shown at the end of cycle 9.

	Valid	Tag	Value	Valid	Tag	Value
α	1	-				
β	0		-	1	-	-16
γ	1	-	-1	1	-	9

ADD RS

	Valid	Tag	Value	Valid	Tag	Value
π						
ρ	1	-	8			
σ						

MUL RS

13

Your job:

1. Fill in the missing entries in the program snippet, RATs, and ADD and MUL reservation stations. You might find it helpful to fill in the timing diagram until cycle 9.

2. Complete the dynamic timing diagram below for the execution of the program snippet, as we have done in class.

- Each row corresponds to one dynamic instruction. The leftmost column identifies the static instruction as I1, I2, etc.

- Use F, D, M (for MUL), A (for ADD), B (for Branch), and W (for write back) to indicate what is going on with each instruction during each clock cycle. **Branch instructions do not write back**.

- Use * to indicate a clock cycle when an instruction is waiting to continue processing.

- Use as many rows/columns as needed.

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
I1	F	D	M	M	M	M	M	W																	