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:	Solutions
EID:	Extra explanations in green

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?

Y, Z are tag and value.
Tag is relevant if the register is waiting to be updated.
Value is relevant if the register contains the value to be used.

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.

LC-3b ISA because it has fixed length instructions, so it is easy to know the addresses of all 4 instructions.

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?

- When the profiling data is not representative of the real run
- When the program has phase behavior

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?

It gets the sources by popping from the stack and pushes the result back into the stack.

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)

`      ADD   R0, R0, #0`
`      BRnp  SKIP`
`      ADD   R1, R1, #1`
`      ADD   R2, R2, #1`
`SKIP  ADD   R3, R3, #1`
	`      ADD   R0, R0, #0`
	`      ADDz  R1, R1, #1`
	`      ADDz  R2, R2, #1`
	`      ADD   R3, R3, #1`

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
	F	D	E	E	E	W																			
		F	D				B																		
			F	D	E	E	E	W																	

How many cycles does it take to execute the program?

8

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
	F	D	E	E	E	W																			
		F	D				B																		
								F	D	E	E	E	W												

How many cycles does it take to execute the program?

13

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
	F	D	E	E	E	W																			
		F	D	E	E	E	W																		
			F	D	E	E	E	W																	
				F	D	E	E	E	W																

How many cycles does it take to execute the program?

9

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

Branch prediction should be more than 80% accurate
A execution time = 8p + (1-p)13 <= 9
4 <= 5p
p > 4/5 = 80%

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.

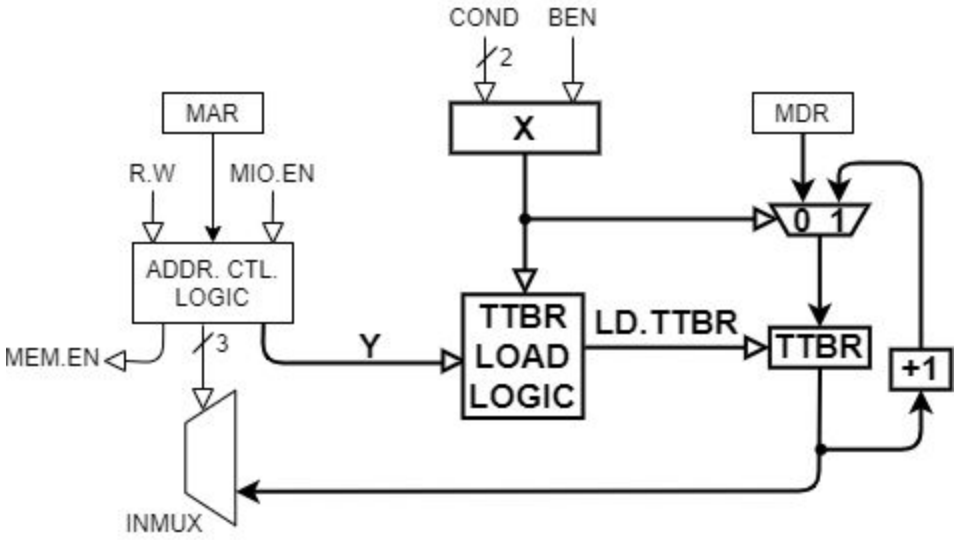
To use the TTBR, we

- 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:



Explanation:

The TTBR is updated in two different ways:

1. When a store instruction writes to TTBR. This is analogous to writes to other memory mapped devices such as KBSR, DSR, and DDR. In this case, TTBR needs to be updated with the content of MDR. The signal Y from the Address Control Logic detects this case when we are storing to TTBR.
2. When a BR instruction is taken. In this case, TTBR needs to be updated with TTBR + 1. The logic block 'X' detects this case when a taken BR instruction is being executed.

Together, the two combinational logic blocks 'X' and 'TTBR Load Logic' should ensure that:

- LD.TTBR is set only in the two cases above when TTBR needs to be updated, and is 0 otherwise
- the mux selects the MDR when TTBR is being stored to (case 1), and selects TTBR + 1 when TTBR is being incremented on a taken BR (case 2).

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	0
00	1	0
01	0	0

01	1	0
10	0	0
10	1	1
11	0	0
11	1	0

Explanation:
As the select signal for the mux, 'X' needs to select TTBR + 1 on a taken BR.
Hence when COND==10 and BEN==1, X = 1.

In all other cases X should be 0; otherwise, the TTBR Load Logic will not be able to differentiate the taken BR case from all other cases.

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	0xFE08	WR	1
			1
			1
			1

Explanation:
The signal Y is used to identify when we are storing to the TTBR. It should only be set when MIO.EN==1, MAR==0xFE08, and R.W==WR.
It should **NOT** be set when MIO.EN==1, MAR==0xFE08, and R.W==**RD**, because then we cannot distinguish stores to TTBR from loads. We need to be able to differentiate between stores to TTBR from loads, because only stores should update the TTBR.

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
0	1	1
1	0	1
1	1	

Explanation:

- X==0, Y==0: neither a store to TTBR nor a taken BR, so we should not update the TTBR.
  LD.TTBR = 0
- X==0, Y==1: storing to TTBR, so TTBR should be updated with the MDR.
  LD.TTBR = 1
- X==1, Y==0: taken BR, so TTBR should be updated with TTBR + 1
  LD.TTBR = 1
- X==1, Y==1: since it's impossible for an instruction to be both a taken BR and and a store to TTBR, this is a don't care

Part d (3 points):

In which state(s) is the TTBR incremented?

State 0

Explanation:

TTBR is incremented when COND==10 and BEN==1, which only occurs in state 0 of the state diagram.

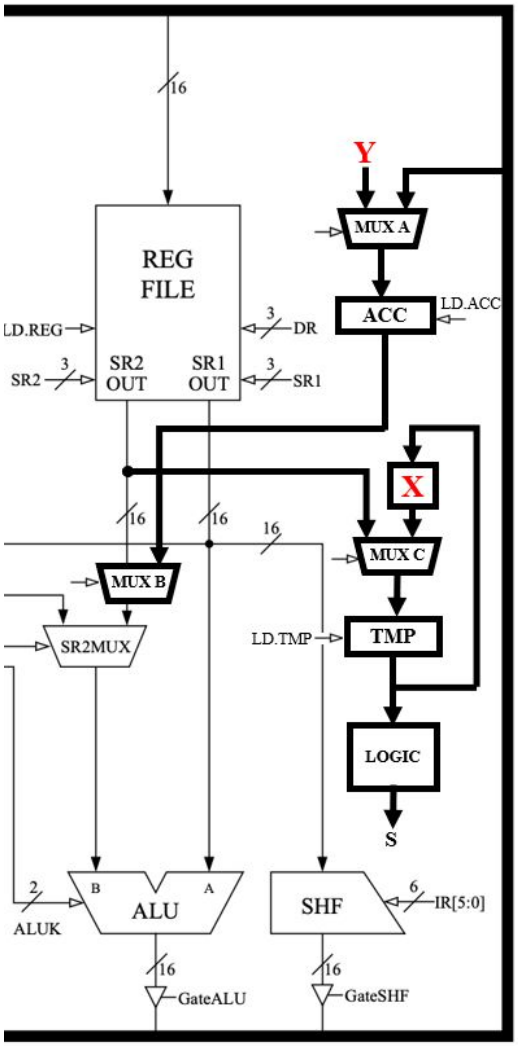
Problem 4 (25 points):

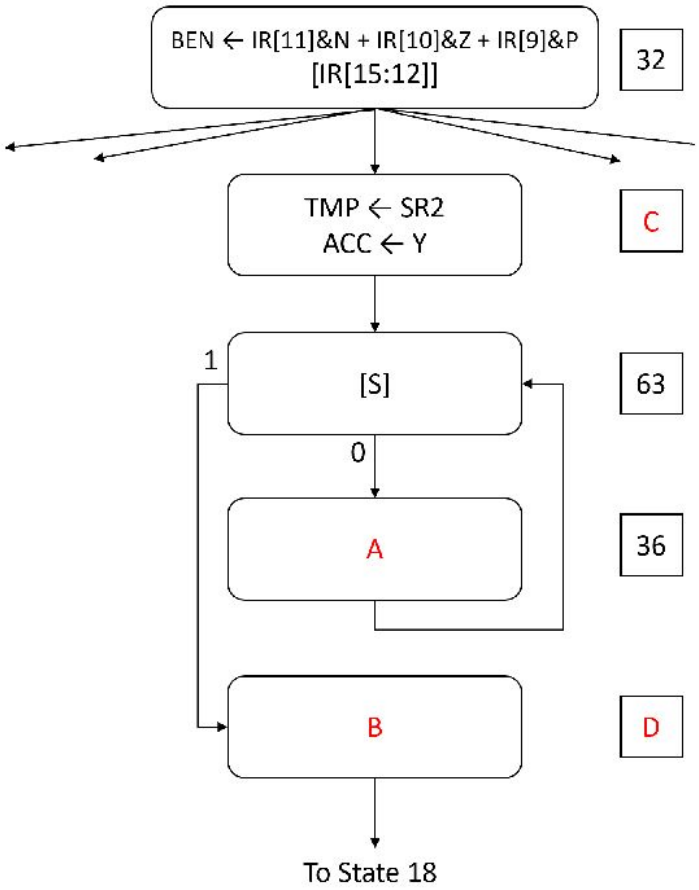
Multiply-Accumulate, often called a MAC operation, refers to an operation that multiplies two operands and subsequently adds the product to a third operand, as follows: *MAC(x, y, z) = x + (y*z)*

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:

ACC ← ACC + SR1
TMP ← TMP - 1

Fill out state B:

DR ← DR + ACC
setCC

What is state number C?

11

What is state number D?

44

Part b (6 points):

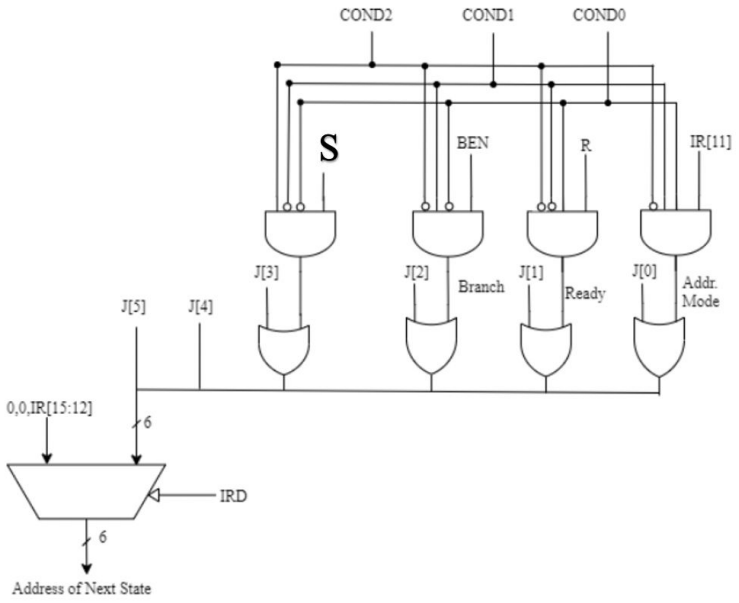
What does operation X represent? Explain.

X is a decrement operation, so the output is TMP-1. This allows us to have the second operand serve as a counter for how many times to add the first operand to itself.

What constant value should input Y be? Explain.

Y should be 0 so that we can initialize the ACC register to 0.

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:

S is TMP==0. If TMP is 0, then we should stop looping because the multiplication portion is complete.

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	1	1	Y		SR2		0
63	0	0					100
36	1	1	Bus	ACC	X	IR[8:6]	0
D	0	0		ACC		IR[11:9]	0

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 , R1 , R0
I2		ADD R2 , R1 , R7
I3		BRnp LOOP
I4		ADD R5 , R3 , R4
I5		ADD R7 , R1 , R1

Solution strategy:

- Look at the reservation stations at cycle 9, there's only 1 multiply and 3 adds (The first multiply has already finished at cycle 8, so the top MUL RS is empty). The only way that is possible is: 2nd and 4th instructions are ADD instructions, and BR is taken once and not taken the second time.
- In the reservation station, we see a value 8 as one of the operands for the MUL. Since there's no value 8 in the register alias table, it must be the result of I1. So, I1 both reads and writes to R1.
    - Note that we know the only instruction that is finished by cycle 9 is I1 because the I2 depends on I1 and if you fill in the timing diagram, later instructions cannot finish by cycle 9 even if they have no dependencies.
- Now that we know the first instance of I1 produces an 8 and reads from R1, the other operand has to be R0 = 2
- We see in the reservation stations that an operand of the second ADD has the value -16. This has to be R7 since the value is ready at cycle 9.
- Now, we confirm that after the 2nd iteration of the loop, R1 = 16, and R2 = 0. Therefore, the branch is not taken.
- The operands of the last ADD instruction in the RS are both ready (-1 and 9) at cycle 9, so they have to be R3 and R4. Since the valid of R5 is 0 in the register alias table at cycle 9, the destination register has to be R5
- Now, all the missing entries in the register alias table and reservation stations entries could be filled by following the Tomasulo algorithm step by step. You could have also done this as you figured out individual pieces of information.
- The only missing instruction is the last one. Note that the value of R7 changes by the end of execution, so it must be I5 that writes to R7. There are two ways to produce a 32: ADD R7, R1, R1 and MUL R7, R1, R0. However, if I5 was a MUL, it would have shown up in the reservation station at cycle 9. Since it does not show up, it must be an ADD that has been stalled at decode because the ADD reservation station entries are full.
- The timing diagram could now be filled in.

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

Before ***Cycle 1***

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

After ***Cycle 9***

	V	Tag	Value
R0	1	-	2
R1	1	-	16
R2	1	-	0
R3	1	-	-1
R4	1	-	9
R5	1	-	8
R6	1	-	-8
R7	1	-	32

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	-	8	1	-	-16
β	0	**ρ**	-	1	-	-16
γ	1	-	-1	1	-	9

ADD RS

	Valid	*Tag*	*Value*	*Valid*	*Tag*	*Value*
π						
ρ	1	-	8	1	-	2
σ						

MUL RS

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																	
		F	D	*	*	*	*	*	A	A	A	W													
			F	D	*	*	*	*	*	*	*	*	B												
				F	D	*	*	*	M	M	M	M	M	W											
					F	D	*	*	*	*	*	*	*	*	A	A	A	W							
						F	D	*	*	*	*	*	*	*	*	*	*	*	B						
							F	D	*	A	A	A	W												
								F	*	*	*	D	*	*	*	A	A	A	W						