Design of Digital Circuits (252-0028-00L), Spring 2022

Optional HW 4: Pipelining and Out-of-Order Execution

SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Juan Gomez Luna, Mohammad Sadrosadati, Hasan Hassan, Mohammed Alser, Ataberk Olgun, Jisung Park, Giray Yaglikci, Can Firtina, Geraldo De Oliveira Junior, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Gagandeep Singh, Behzad Salami, Rakesh Nadig, Joel Lindegger

Released: Friday, May 6th, 2022

1 Pipelining (I)

Given the following code:

```
MUL R3, R1, R2
ADD R5, R4, R3
ADD R6, R4, R1
MUL R7, R8, R9
ADD R4, R3, R7
MUL R10, R5, R6
```

Calculate the number of cycles it takes to execute the given code on the following models:

Note 1: Each instruction is specified with the destination register first.

Note 2: Do not forget to list any assumptions you make about the pipeline structure (e.g., how is data forwarding done between pipeline stages)

Note 3: For all machine models, use the basic instruction cycle as follows:

- Fetch (one clock cycle)
- Decode (one clock cycle)
- Execute (MUL takes 6, ADD takes 4 clock cycles). The multiplier and the adder are not pipelined.
- Write-back (one clock cycle)

(a) A non-pipelined machine

MUL: $1 + 1 + 6 + 1 = 9$ cycles
ADD: $1 + 1 + 4 + 1 = 7$ cycles

$9 + 7 + 7 + 9 + 7 + 9 = 48$ cycles

(b) A pipelined machine with scoreboarding and five adders and five multipliers without data forwarding

28 cycles

PC	Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	MUL R3, R1, R2	F	D	E	E	E	E	E	E	W							
2	ADD R5, R4, R3		F	D	-	-	-	-	-	-	-	E	E	E	E	W	
3	ADD R6, R4, R1			F	-	-	-	-	-	-	-	D	E	E	E	E	W
4	MUL R7, R8, R9											F	D	E	E	E	E
5	ADD R4, R3, R7												F	D	-	-	-
6	MUL R10, R5, R6													F	-	-	-

PC	Cycles	...	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	MUL R3, R1, R2	...														
2	ADD R5, R4, R3	...	W													
3	ADD R6, R4, R1	...	E	W												
4	MUL R7, R8, R9	...	E	E	E	E	W									
5	ADD R4, R3, R7	...	-	-	-	-	-	-	E	E	E	E	W			
6	MUL R10, R5, R6	...	-	-	-	-	-	-	D	E	E	E	E	E	E	W

(c) A pipelined machine with scoreboarding and five adders and five multipliers with data forwarding.

26 cycles

PC	Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	MUL R3, R1, R2	F	D	E	E	E	E	E	E	W							
2	ADD R5, R4, R3		F	D	-	-	-	-	-	-	E	E	E	E	W		
3	ADD R6, R4, R1			F	-	-	-	-	-	-	D	E	E	E	E	W	
4	MUL R7, R8, R9										F	D	E	E	E	E	E
5	ADD R4, R3, R7											F	D	-	-	-	-
6	MUL R10, R5, R6												F	-	-	-	-

PC	Cycles	...	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	MUL R3, R1, R2	...														
2	ADD R5, R4, R3	...														
3	ADD R6, R4, R1	...	W													
4	MUL R7, R8, R9	...	E	E	E	W										
5	ADD R4, R3, R7	...	-	-	-	-	E	E	E	E	W					
6	MUL R10, R5, R6	...	-	-	-	-	D	E	E	E	E	E	E	W		

(d) A pipelined machine with scoreboarding and one adder and one multiplier without data forwarding

31 cycles

PC	Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	MUL R3, R1, R2	F	D	E	E	E	E	E	E	W							
2	ADD R5, R4, R3		F	D	-	-	-	-	-	-	-	E	E	E	E	W	
3	ADD R6, R4, R1			F	-	-	-	-	-	-	-	D	-	-	-	E	E
4	MUL R7, R8, R9											F	-	-	-	D	E
5	ADD R4, R3, R7															F	-
6	MUL R10, R5, R6																

PC	Cycles	...	15	16	17	18	19	20	21	22	23	24	25	26	27	28
1	MUL R3, R1, R2	...														
2	ADD R5, R4, R3	...	W													
3	ADD R6, R4, R1	...	E	E	E	E	W									
4	MUL R7, R8, R9	...	D	E	E	E	E	E	E	W						
5	ADD R4, R3, R7	...	F	D	-	-	-	-	-	-	-	E	E	E	E	W
6	MUL R10, R5, R6	...		F	-	-	-	-	-	-	-	D	E	E	E	E

PC	Cycles	...	27	28	29	30	31	32	33	34	35	36	37	38	39	40
5	ADD R4, R3, R7	...	E	W												
6	MUL R10, R5, R6	...	E	E	E	E	W									

(e) A pipelined machine with scoreboarding and one adder and one multiplier with data forwarding

29 cycles

PC	Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	MUL R3, R1, R2	F	D	E	E	E	E	E	E	W							
2	ADD R5, R4, R3		F	D	-	-	-	-	-	-	E	E	E	E	W		
3	ADD R6, R4, R1			F	-	-	-	-	-	-	D	-	-	-	E	E	E
4	MUL R7, R8, R9										F	-	-	-	D	E	E
5	ADD R4, R3, R7														F	D	-
6	MUL R10, R5, R6															F	-

PC	Cycles	...	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	MUL R3, R1, R2	...														
2	ADD R5, R4, R3	...														
3	ADD R6, R4, R1	...	E	E	E	W										
4	MUL R7, R8, R9	...	E	E	E	E	E	W								
5	ADD R4, R3, R7	...	-	-	-	-	-	-	E	E	E	E	W			
6	MUL R10,R5, R6	...	-	-	-	-	-	-	D	E	E	E	E	E	E	W

2 Pipelining (II)

Consider two pipelined machines implementing the MIPS ISA, Machine A and Machine B. Both machines have *one ALU* and the following *five pipeline stages*, very similar to the basic 5-stage pipelined MIPS processor we discussed in lectures:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (one clock cycle)
4. Memory (one clock cycle)
5. Write-back (one clock cycle).

Machines A and B have the following specifications:

	Machine A	**Machine B**
Data Forwarding/Interlocking	Does **NOT** implement interlocking in hardware. Relies on the compiler to order instructions or insert `nop` instructions such that dependent instructions are correctly executed.	Implements data dependence detection and data forwarding in hardware. On detection of instruction dependence, it forwards an operand from the memory stage or from the write-back stage to the execute stage. The result of a load instruction (`lw`) can *only* be forwarded from the write-back stage.
Internal register file forwarding	Implemented (i.e., an instruction writes into a register in the first half of a cycle and another instruction can correctly access the same register in the second half of the cycle).	Same as Machine A
Branch Prediction	Predicts all branches as *always-taken*, and the next program counter is available after the decode stage.	Same as Machine A

Consider the following code segment:

```
Loop: lw   $1, 0($4)
      lw   $2, 400($4)
      add  $3, $1, $2
      sw   $3, 0($4)
      sub  $4, $4, #4
      bnez $4, Loop
```

Initially, `$1 = 0, $2 = 0, $3 = 0, and $4 = 400.`

(a) Re-write the code segment above *with minimal changes* so that it gets correctly executed in Machine A *with minimal latency.* You can either insert `nop` instructions or reorder instructions as needed.

```
Loop:   lw $1, 0($4)
        lw $2, 400($4)
        nop
        nop
        add $3, $1, $2
        nop
        nop
        sw $3, 0($4)
        sub $4, $4, #4
        nop
        nop
        bnez $4, Loop
```

(b) Fill the table below with the timeline of the first loop iteration of the code segment in Machine A.

Instruction	**Clock cycle number**																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`lw $1, 0($4)`	F	D	E	M	W														
`lw $2, 400($4)`		F	D	E	M	W													
`nop`			F	D	E	M	W												
`nop`				F	D	E	M	W											
`add $3, $1, $2`					F	D	E	M	W										
`nop`						F	D	E	M	W									
`nop`							F	D	E	M	W								
`sw $3, 0($4)`								F	D	E	M	W							
`sub $4, $4, #4`									F	D	E	M	W						
`nop`										F	D	E	M	W					
`nop`											F	D	E	M	W				
`bnez $4, Loop`												F	D	E	M	W			

(c) Calculate the number of cycles it takes to execute the code segment on Machine A. Show your work in the box.

Total number of cycles: 1303.

Explanation:
The compiler reorders instructions and places six `nop`-s.
This is the execution timeline of the first iteration:

Each iteration consists of 12 instructions. Since the next program counter is available after the decode stage of bnez, the next iteration starts with an additional delay of 1 cycle.
The last iteration takes 16 cycles, to drain the pipeline.
Thus the entire program runs for 99 * 13 + 16 = 1303 cycles.

(d) Fill the table below with the timeline of the first loop iteration of the code segment in Machine B.

Instruction	**Clock cycle number**																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`lw $1, 0($4)`	F	D	E	M	W														
`lw $2, 400($4)`		F	D	E	M	W													
`add $3, $1, $2`			F	D	*	E	M	W											
`sw $3, 0($4)`				F	*	D	E	M	W										
`sub $4, $4, #4`						F	D	E	M	W									
`bnez $4, Loop`							F	D	E	M	W								
`lw $1, 0($4)`								*	F	D	E	M	W						

(e) Calculate the number of cycles it takes to execute the code segment on Machine B. Show your work in the box.

Total number of cycles: 803.
Explanation:
1 - Foward $2 from W to E in cycle 6.
2 - Foward $3 from M to E in cycle 7.
3 - Foward $4 from M to E in cycle 9.

Each iteration takes 8 cycles, including one cycle delay after bnez, because to the next program counter is available only after the decode stage of bnez.
The last iteration takes 11 cycles, to drain the pipeline.
Thus total number of cycles is 99*8 + 11 = 803 cycles.

3 Pipeline - Reverse Engineering

Algorithm 1 contains a piece of assembly code. Table 1 presents the execution timeline of this code.

```
    MOVI R1, X         # R1 <- X
    MOVI R2, Y         # R2 <- Y
L1:
    ADD  R1, R1, R2    # R1 <- R1 + R2
    MUL  R4, R2, R3    # R4 <- R2 x R3
    SUBI R3, R1, 100   # R3 <- R1 - 100, set condition flags
    JNZ  L1            # Jump to L1 if zero flag is not set
    MUL  R1, R1, R2    # R1 <- R1 x R2
    MUL  R2, R3, R4    # R2 <- R3 x R4
    ADD  R5, R6, R7    # R5 <- R6 + R7
```

Algorithm 1: Assembly Program

Dyn. Instr.	Instructions	Cycles													
Number		1	2	3	4	5	6	7	8	9	10	11	12	13	...
1	MOV R1, X	F	D	E1	E2	E3	M	W							
2	MOV R2, Y		F	D	E1	E2	E3	M	W						
3	ADD R1, R1, R2			F	D	-	-	E1	E2	E3	M	W			
4	MUL R4, R2, R3				F	-	-	D	E1	E2	E3	M	W		
5	SUBI R3, R1, 100							F	D	-	E1	E2	E3	M	...
6	JNZ L1								F	-	D	-	-	E1	...
7	...														

Table 1: Execution timeline (F:Fetch, D:Decode, E:Execute, M:Memory, W:WriteBack)

Use this information to reverse engineer the architecture of this microprocessor to answer the following questions. Answer the questions as precisely as possible with the provided information. If the provided information is not sufficient to answer a question, answer "Unknown" and explain your reasoning clearly.

(a) List the necessary data forwardings between pipeline stages to exhibit this behavior.

The result of E3 stage is forwarded to E1 stage (e.g., R1's value at clock cycle 10 and R2's value at clock cycle 7).
The result of E3 stage is forwarded to the condition registers (e.g., SUBI and JNZ at clock cycle 13).
There is no other information for any other data forwarding. Therefore, other data forwardings are unknown.

(b) Does this machine use hardware-interlocking or software-interlocking? Explain.

Hardware-interlocking. It detects data dependencies and stalls the pipeline accordingly without needing any software-induced NOPs.

(c) Consider another machine that uses the opposite of your choice in the previous question. (e.g., if your answer is software-interlocking for the previous question, consider another machine using hardware-interlocking, or vice-versa). How would the execution timeline shown in Table 1 change? What would be different? Fill the following table and explain your reasoning below. (*Notice that the table below consists of two parts: the first seven cycles at the top, and the next seven cycles at the bottom.*)

We inject NOP instructions in between existing instructions to delay instructions with data dependencies.

Dyn. Instr.	Instructions	Cycles													
Number		1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	MOV R1, X	F	D	E1	E2	E3	M	W							
2	MOV R2, Y		F	D	E1	E2	E3	M							
3	NOP			F	D	E1	E2	E3	M	W					
4	NOP				F	D	E1	E2	E3	M	W				
5	ADD R1, R1, R2					F	D	E1	E2	E3	M	W			
6	MUL R4, R2, R3						F	D	E1	E2	E3	M	W		
7	NOP							F	D	E1	E2	E3	M	W	
8	SUB R3, R1, 100								F	D	E1	E2	E3	M	...
9	NOP									F	D	E1	E2	E3	...
10	NOP										F	D	E1	E2	...
11	JNZ L1											F	D	E1	...

For the rest of this question, assume the following:

- X = Y = 1 in Algorithm 1.
- Branch conditions are resolved at the stage E1.
- Branch predictor is static and predicts "always taken".
- The machine uses hardware-interlocking.

At a given clock cycle T,

- the value stored in R1 is 98.
- the processor fetches the dynamic instruction N which is `ADD R1, R1, R2`

(d) Calculate the value of T. Show your work.

T = 682.

Explanation.
Steady state throughput of an iteration is 4 instructions in 7 cycles. The first iteration takes 10 cycles as shown below.

If R1 = 98, this iteration is executed for 97 times so far.

Since in cycle T the first instruction of the loop is being fetched, no cycles of the 98th iteration have executed so far.

Then, $T = 10 + 96 \times 7 + 0 = 682$

Note that this calculation does not account for any stall cycles after the branch. For simplicity, we assume that the branch target immediate is be available after the branch's fetch stage already, i.e., there is no stall after the branch.

(e) Calculate the value of N. Show your work.

N = 390.

Explanation.
Loop iterates for 97 times before reaching to clock cycle T.

There are two instructions before the loop starts.

Then, $N = 2 + 97 \times 4 = 390$, assuming that the instruction indices start from 0.

4 Tomasulo's Algorithm (I)

Remember that Tomasulo's algorithm requires tag broadcast and comparison to enable wake-up of dependent instructions. In this question, we will calculate the number of tag comparators and size of tag storage required to implement Tomasulo's algorithm in a machine that has the following properties:

- 8 functional units where each functional unit has a dedicated separate tag and data broadcast bus
- 32 64-bit architectural registers
- 16 reservation station entries per functional unit
- Each reservation station entry can have two source registers

Answer the following questions. Show your work for credit.

(a) What is the number of tag comparators per reservation station entry?

$8 * 2$

(b) What is the total number of tag comparators in the entire machine?

$16 * 8 * 2 * 8 + 8 * 32$

(c) What is the (minimum possible) size of the tag?

$log(16 * 8) = 7$

(d) What is the (minimum possible) size of the register alias table (or, frontend register file) in bits?

$72 * 32$ (64 bits for data, 7 bits for the tag, 1 valid bit)

(e) What is the total (minimum possible) size of the tag storage in the entire machine in bits?

$7 * 32 + 7 * 16 * 8 * 2$

5 Tomasulo's Algorithm (II)

In this problem, we consider a scalar processor with in-order fetch, out-of-order dispatch, and in-order retirement execution engine that employs Tomasulo's algorithm. This processor behaves as follows:

- The processor has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The processor implements a single-level data cache.
- The processor has the following *two types of execution units* but it is *unknown* how many of each type the processor has.
  - **Integer ALU:** Executes integer instructions (i.e., addition, multiplication, move, branch).
  - **Memory Unit:** Executes load/store instructions.
- The processor is connected to a main memory that has a fixed access latency.
- Load/store instructions spend cycles in the E stage exclusively for accessing the data cache or the main memory.
- There are two reservation stations, one for each execution unit type.

The reservation stations are all initially empty. The processor executes an arbitrary program. From the beginning of the program until the program execution finishes, *seven* dynamic instructions enter the processor pipeline. Table 3 shows the seven instructions and their execution diagram.

Instruction semantics:

- MV R0 ← #0x1000: moves the hexademical number 0x1000 to register $R0$.
- LD R1 ← [R0]: loads the value stored at memory address $R0$ to register $R1$.
- BL R1, #100, #LB1: a branch instruction that conditionally takes the path specified by label "#LB1" if the content of register $R1$ is smaller than integer value 100.
- MUL R1 ← R1, #5: multiplies $R1$ and 5 and writes the result to $R1$.
- ST [R0] ← R1: stores $R1$ to memory address specified by $R0$.
- ADD R1 ← R1, R0: adds $R1$ and $R0$ and writes the result to $R1$.

Instruction/Cycle:	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	26
1: MV R0 ← #0x1000	F	D	E1	E2	E3	E4	W																			
2: LD R1 ← [R0]		F	D	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	W											
3: BL R1 #100, #LB1						F	D	-	-	-	-	-	-	-	E1	E2	E3	E4	W							
4: MUL R1 ← R1, #5														F	D	E1	E2	E3	//squashed (i.e., killed)							
5: ST [R0] ← R1															F	D	-	-	//squashed (i.e., killed)							
6: ADD R1 ← R1, R0																			F	D	E1	E2	E3	E4	W	
7: ST [R0] ← R1																				F	D	-	-	-	E1	W

Table 2: Execution diagram of the seven instructions.

(a) Using the information provided above, answer the following questions regarding the processor design. If a question has more than one correct answer or a correct answer cannot be determined using the information provided in the question, answer the question as specifically as possible. For example, use phrases such as "at least/at most" and try to narrow down the answer using the information that is provided in the question and can be inferred from Table 3. If nothing can be inferred, write "Unknown" as an answer. Explain your reasoning briefly.

What is the cache hit latency?

1 cycle. The last ST instruction that writes to the same memory location that is previously loaded by LD takes only 1 cycle in E stage.

What is the cache miss latency?

8 cycles. The first LD instruction spends 8 cycles in the E stage.

What is the cache line size?

Unknown. We cannot infer the cache line size from one LD and ST instructions and also we cannot determine the minimum cache line size since the register width is not given in the question.

What is the number of entries in each reservation station (R)?

ALU → at least 2, MU → unknown

How many ALUs does the processor have?

At least 2 ALUs if not pipelined, otherwise at least 1 ALU. This is because we see two arithmetic instructions simultaneously in E stage in the pipeline diagram.

Is the integer ALU pipelined?

Yes if the processor has 1 ALU, otherwise no. The explanation is similar to the above question.

Does the processor perform branch prediction?

Yes because there are squashed (as a result of branch misprediction) instructions in the pipeline.

At which pipeline stage is the correct outcome of a branch evaluated?

At the end of stage E4. This is because in the next cycle after the branch instruction completes E4 previously fetched instructions are killed and an instruction from the correct path is fetched.

(b) What is the program (i.e., static instructions) that leads to the execution diagram shown in Table 3? Fill in the blanks below with the known instructions of the program and also (if applicable) show where and how many unknown instructions there are in the program.

Program:

MV R0 ← #0x1000

LD R1 ← [R0]

BL R1 #100, #LB1

MUL R1 ← R1, #5

ST [R0] ← R1

Any number of unknown instructions can be here

LB1: ADD R1 ← R1, R0

ST [R0] ← R1

6 Tomasulo's Algorithm - Reverse Engineering

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for an out-of-order execution engine that employs Tomasulo's algorithm, as we discussed in lectures. Your job is to determine the original sequence of **four instructions** in program order.

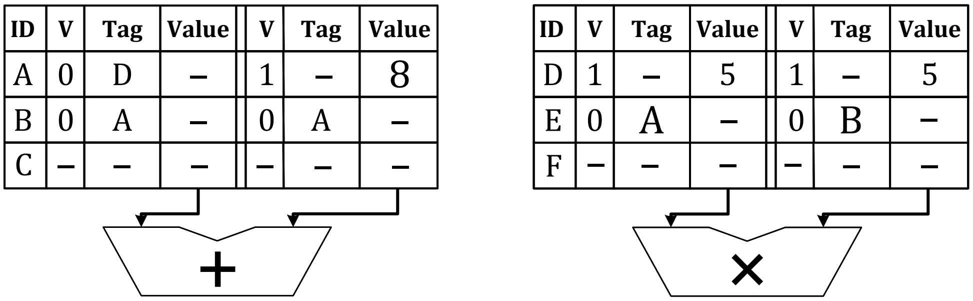
The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine executes *only* register-type instructions, e.g., $OP\ R_{dest} \leftarrow R_{src1},\ R_{src2}$.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- The adder and multiplier **are not** pipelined. An add operation takes 2 cycles. A multiply operation takes 3 cycles.
- The result of an addition and multiplication is broadcast to the reservation station entries and the RAT in the writeback stage. A dependent instruction can begin execution in the next cycle after the writeback if it has all of its operands available in the reservation station entry.
- When multiple instructions are ready to execute at a functional unit at the same cycle, the oldest ready instruction is chosen to be executed first.

Initially, the machine is empty. Four instructions then are fetched, decoded, and dispatched into reservation stations. Pictured below is the state of the machine when the final instruction has been dispatched into a reservation station:

RAT

Reg	V	Tag	Value
R0	–	–	–
R1	0	A	5
R2	1	–	8
R3	0	E	–
R4	0	B	–
R5	–	–	–



ID	V	Tag	Value	V	Tag	Value
A	0	D	–	1	–	8
B	0	A	–	0	A	–
C	–	–	–	–	–	–

ID	V	Tag	Value	V	Tag	Value
D	1	–	5	1	–	5
E	0	A	–	0	B	–
F	–	–	–	–	–	–

(a) Give the four instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: "opcode destination ⇐ source1, source2."

MUL R1 ⇐ R1 , R1

ADD R1 ⇐ R1 , R2

ADD R4 ⇐ R1 , R1

MUL R3 ⇐ R1 , R4

(b) Now assume that the machine flushes all instructions out of the pipeline and restarts fetch from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of four instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fourth instruction.

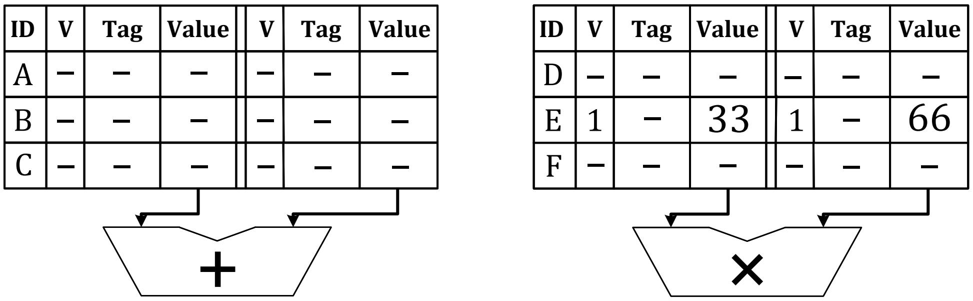
As we saw in lectures, use "F" for fetch, "D" for decode, "En" to signify the nth cycle of execution for an instruction, and "W" to signify writeback. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MUL R1 ← R1, R1	F	D	E1	E2	E3	W										
ADD R1 ← R1, R2		F	D				E1	E2	W							
ADD R4 ← R1, R1			F	D						E1	E2	W				
MUL R3 ← R1, R4				F	D								E1	E2	E3	W

(c) Finally, show the state of the RAT and reservation stations at the end of the **12th cycle** of execution in the figure below. Complete all blank parts.

RAT

Reg	V	Tag	Value
R0	–	–	–
R1	1	–	33
R2	1	–	8
R3	0	E	–
R4	1	–	66
R5	–	–	–



ID	V	Tag	Value	V	Tag	Value
A	–	–	–	–	–	–
B	–	–	–	–	–	–
C	–	–	–	–	–	–

ID	V	Tag	Value	V	Tag	Value
D	–	–	–	–	–	–
E	1	–	33	1	–	66
F	–	–	–	–	–	–

7 Out-of-Order Execution

In this problem, we consider an in-order fetch, out-of-order dispatch, and in-order retirement execution engine that employs Tomasulo's algorithm. This engine behaves as follows:

- The engine has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The engine can fetch one instruction per cycle, decode one instruction per cycle, and write back the result of one instruction per cycle.
- The engine has two execution units: 1) an adder for executing ADD instructions and 2) a multiplier for executing MUL instructions.
- The execution units are fully pipelined. The adder has two stages (E1-E2) and the multiplier has four stages (E1-E2-E3-E4). Execution of each stage takes one cycle.
- The adder has a two-entry reservation station and the multiplier has a four-entry reservation station.
- An instruction always allocates the first available entry of the reservation station (in top-to-bottom order) of the corresponding execution unit.
- Full data forwarding is available, i.e., during the last cycle of the E stage, the tags and data are broadcast to the reservation station and the Register Alias Table (RAT). For example, an ADD instruction updates the reservation station entries of the dependent instructions in E2 stage. So, the updated value can be read from the reservation station entry in the next cycle. Therefore, a dependent instruction can potentially begin its execution in the next cycle (after E2).
- The multiplier and adder have separate output data buses, which allow both the adder and the multiplier to update the reservation station and the RAT in the same cycle.
- An instruction continues to occupy a reservation station slot until it finishes the Write-back (W) stage. The reservation station entry is deallocated after the Write-back (W) stage.

7.1 Problem Definition

The processor is about to fetch and execute *six* instructions. Assume the *reservation stations (RS)* are all initially empty and the initial state of the *register alias table (RAT)* is given below in Figure (a). Instructions are fetched, decoded and executed as discussed in class. At some point during the execution of the six instructions, a snapshot of the state of the RS and the RAT is taken. Figures (b) and (c) show the state of the RS and the RAT at the snapshot time. A dash (–) indicates that a value has been cleared. A question mark (?) indicates that a value is unknown.

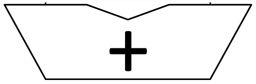
Reg	Valid	Tag	Value
R0	1	–	1900
R1	1	–	82
R2	1	–	1
R3	1	–	3
R4	1	–	10
R5	1	–	5
R6	1	–	23
R7	1	–	35
R8	1	–	61
R9	1	–	4

Initial state of the RAT

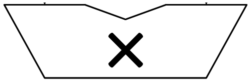
Reg	Valid	Tag	Value
R0	1	?	1900
R1	0	Z	?
R2	1	?	12
R3	1	?	3
R4	1	?	10
R5	0	B	?
R6	1	?	23
R7	0	H	?
R8	1	?	350
R9	0	A	?

Snapshot state of the RAT

ID	V	Tag	Value	V	Tag	Value
A	1	?	350	1	?	12
B	0	A	?	0	Z	?

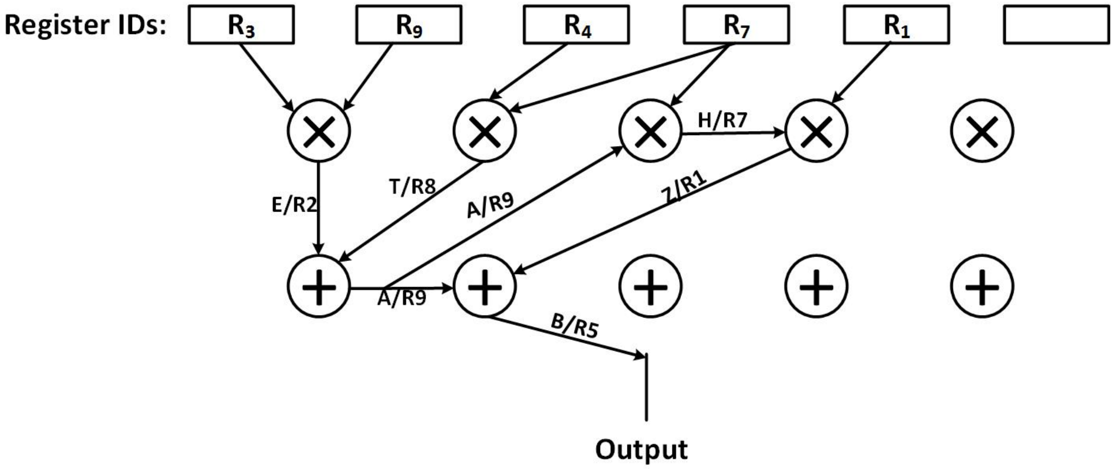


ID	V	Tag	Value	V	Tag	Value
–	–	–	–	–	–	–
T	1	?	10	1	?	35
H	1	?	35	0	A	?
Z	1	?	82	0	H	?



7.2 (a) Data Flow Graph

Based on the information provided above, identify the instructions and complete the dataflow graph below for the six instructions that have been fetched. Please appropriately connect the nodes using edges and specify the direction of each edge. Label each edge with the destination architectural register and the corresponding Tag. *Note that you may **not** need to use all registers and/or nodes provided below.*



7.3 (b) Program Instructions

Fill in the blanks below with the six-instruction sequence in program order. When referring to registers, please use their architectural names (R0 through R9). Place the register with the smaller architectural name on the left source register box. For example, `ADD R8 ⇐ R1, R5`.

MUL	R2	←	R3	,	R9
MUL	R8	←	R4	,	R7
ADD	R9	←	R2	,	R8
MUL	R7	←	R7	,	R9
MUL	R1	←	R1	,	R7
ADD	R5	←	R1	,	R9

8 Out-of-Order Execution - Reverse Engineering

A five instruction sequence executes according to Tomasulo's algorithm. Each instruction is of the form ADD DR,SR1,SR2 or MUL DR,SR1,SR2. ADDs are pipelined and take 9 cycles (F-D-E1-E2-E3-E4-E5-E6-WB). MULs are also pipelined and take 11 cycles (two extra execute stages). An instruction must wait until a result is in a register before it sources it (reads it as a source operand). For instance, if instruction 2 has a read-after-write dependence on instruction 1, instruction 2 can start executing in the next cycle after instruction 1 writes back (shown below).

```
instruction 1     |F|D|E1|E2|E3|..... |WB|
instruction 2       |F|D |- |- |..... |- |E1|
```

The machine can fetch one instruction per cycle, and can decode one instruction per cycle.

The register file before and after the sequence are shown below.

	Valid	Tag	Value
R0	1		4
R1	1		5
R2	1		6
R3	1		7
R4	1		8
R5	1		9
R6	1		10
R7	1		11

	Valid	Tag	Value
R0	1		310
R1	1		5
R2	1		410
R3	1		31
R4	1		8
R5	1		9
R6	1		10
R7	1		21

(a) Complete the five instruction sequence in program order in the space below. Note that we have helped you by giving you the opcode and two source operand addresses for the fourth instruction. (The program sequence is unique.)

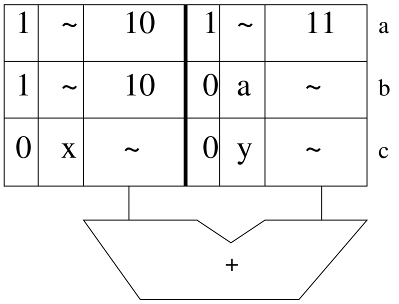
Give instructions in the following format: "opcode destination ⇐ source1, source2."

ADD	R7	⇐	R6	,	R7
ADD	R3	⇐	R6	,	R7
MUL	R0	⇐	R3	,	R6
MUL	R2	⇐	R6	,	R6
ADD	R2	⇐	R0	,	R2

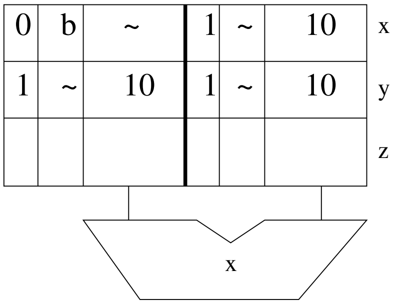
(b) In each cycle, a single instruction is fetched and a single instruction is decoded.

Assume the reservation stations are all initially empty. Put each instruction into the next available reservation station. For example, the first ADD goes into "a". The first MUL goes into "x". Instructions remain in the reservation stations until they are completed. Show the state of the reservation stations at the end of cycle 8.

Note: to make it easier for the grader, when allocating source registers to reservation stations, please always have the higher numbered register be assigned to source2.



1	~	10	1	~	11	a
1	~	10	0	a	~	b
0	x	~	0	y	~	c



0	b	~	1	~	10	x
1	~	10	1	~	10	y
						z

(c) Show the state of the Register Alias Table (Valid, Tag, Value) at the end of cycle 8.

	Valid	Tag	Value
R0	0	x	4
R1	1	~	5
R2	0	c	6
R3	0	b	7
R4	1	~	8
R5	1	~	9
R6	1	~	10
R7	0	a	11

Extra Exercises for Practicing

The following exercises are old exam questions that are conceptually similar to the ones above, but with slight alterations. We do not expect or recommend you to solve all of them, unless you think you are struggling with a particular concept, or would like to do practice runs on these old exam questions.

9 Pipelining (Extra)

Consider two pipelined machines implementing MIPS ISA, Machine I and Machine II:

Both machines have the following *five pipeline stages*, very similarly to the basic 5-stage pipelined MIPS processor we discussed in lectures, and *one ALU*:

1. Fetch (one clock cycle)
2. Decode (one clock cycle)
3. Execute (one clock cycle)
4. Memory (one clock cycle)
5. Write-back (one clock cycle).

Machine I does *not* implement interlocking in hardware. It assumes all instructions are independent and relies on the compiler to order instructions such that there is sufficient distance between dependent instructions. The compiler either moves other independent instructions between two dependent instructions, if it can find such instructions, or otherwise, inserts `nop`s. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can correctly access the same register in the next half of the cycle). Assume that the processor predicts all branches as always-taken.

Machine II implements data forwarding in hardware. On detection of a flow dependence, it forwards an operand from the memory stage or from the write-back stage to the execute stage. The load instruction (`lw`) can *only* be forwarded from the write-back stage because data becomes available in the memory stage but not in the execute stage like for the other instructions. Assume internal register file forwarding (an instruction writes into a register in the first half of a cycle and another instruction can access the same register in the next half of the cycle). The compiler does *not* reorder instructions. Assume that the processor predicts all branches as always-taken.

Consider the following code segment:

```
Copy: lw   $2, 100($5)
      sw   $2, 200($6)
      addi $1, $1, 1
      bne  $1, $25, Copy
```

Initially, `$5` = 0, `$6` = 0, `$1` = 0, and `$25` = 25.

(a) When the given code segment is executed on Machine I, the compiler has to reorder instructions and insert `nop`s if needed. Write the resulting code that has minimal modifications from the original.

```
Copy:  lw $2, 100($5)
       addi $1, $1, 1
       nop
       sw $2, 200($6)
       bne $1, $25, Copy
```

(b) When the given code segment is executed on Machine II, dependencies between instructions are resolved in hardware. Explain when data is forwarded and which instructions are stalled and when they are stalled.

In every iteration, data are forwarded for `sw` and for `bne`. The instruction `sw` is dependent on `lw`, so it is stalled one cycle in every iteration

(c) Calculate the *machine code size* of the code segments executed on Machine I (part (a)) and Machine II (part (b)).

Machine I - 20 bytes (because of the additional `nop`)
Machine II - 16 bytes

(d) Calculate the number of cycles it takes to execute the code segment on Machine I and Machine II.

Machine I: The compiler reorders instructions and places one `nop`. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	E	M	W			
		N	N	N	N	N		
			F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 5 instructions in each iteration and 25 iterations, the total number of cycles is 129 cycles.

Machine II: The machine stalls `sw` one cycle in the decode stage. This is the execution timeline of the first iteration:

1	2	3	4	5	6	7	8	9
F	D	E	M	W				
	F	D	D	E	M	W		
		F	F	D	E	M	W	
				F	D	E	M	W

9 cycles for one iteration. As there are 4 instructions in each iteration and 25 iterations, and one stall cycle in each iteration, the total number of cycles is 129 cycles.

(e) Which machine is faster for this code segment? Explain.

For this code segment, both machines take the same number of cycles. We cannot say which one is faster, since we do not know the clock frequency.

10 Pipeline - Reverse Engineering (Extra)

The following piece of code runs on a pipelined microprocessor as shown in the table (F: Fetch, D: Decode, E: Execute, M: Memory, W: Write back). Instructions are in the form “Instruction Destination,Source1,Source2.” For example, “ADD A, B, C” means A ← B + C.

	Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	MUL R5, R6, R7	F	D	E1	E2	E3	E4	M	W										
1	ADD R4, R6, R7		F	D	E1	E2	E3	-	M	W									
2	ADD R5, R5, R6			F	D	-	-	E1	E2	E3	M	W							
3	MUL R4, R7, R7				F	-	-	D	E1	E2	E3	E4	M	W					
4	ADD R6, R7, R5							F	D	-	E1	E2	E3	M	W				
5	ADD R3, R0, R6								F	-	D	-	-	E1	E2	E3	M	W	
6	ADD R7, R1, R4										F	-	-	D	E1	E2	E3	M	W

Use this information to reverse engineer the architecture of this microprocessor to answer the following questions. Answer the questions as precise as possible with the provided information. If the provided information is not sufficient to answer a question, answer “Unknown” and explain your reasoning clearly.

(a) How many cycles does it take for an adder and for a multiplier to calculate a result?

3 cycles for adder (E1, E2, E3) and 4 cycles for multiplier (E1, E2, E3, E4).

(b) What is the minimum number of register file read/write ports that this architecture implements? Explain.

The register file has two read ports and one write port.
Decode and Writeback stages can be performed simultaneously as seen at cycle 8. Decode reads from two registers, Writeback writes to one register.

(c) Can we reduce the execution time of this code by enabling more read/write ports in the register file? Explain.

It is not possible to reduce stall cycles of the given code by only enabling more register file ports, as the pipeline would be stalled due to other limited resources.

(d) Does this architecture implement any data forwarding? If so, how is data forwarding done between pipeline stages? Explain.

There is data forwarding from the M stage to E1, as we observe that the instruction 2 starts using R5 at the clk cycle 7, which is one clk cycle after the instruction 0 finishes calculating its result in the execution unit.
Similarly, as another proof of this data forwarding, we observe that the instruction 4 starts using R5 at the clk cycle 10, which is one clk cycle after the instruction 2 finishes calculating its result in the execution unit.

Any other data forwarding is *unknown* with the given information.

(e) Is it possible to run this code faster by adding more data forwarding paths? If it is, how? Explain.

Not possible.

All instructions that stall due to data dependency are already using the best possible data forwarding. There is no stall cycles that can be eliminated by enabling another form of data forwarding.

(f) Is there internal forwarding in the register file? If there is not, how would the execution time of the same program change by enabling internal forwarding in the register file? Explain.

The register file already implements internal forwarding, as instruction 6 can finish the decode stage by fetching the value of R4 from the register file in the same cycle that R4 is written (cycle 13).

(g) Optimize the assembly code in order to reduce the number of stall cycles. You are allowed to *reorder*, *add*, or *remove* ADD and MUL instructions. You are expected to achieve the minimum possible execution time. Make sure that the register values that the optimized code generates at the end of its execution are identical to the register values that the original code generates at the end of its execution. Justify each individual change you make. Show the execution timeline of each instruction and what stage it is in the table below. (*Notice that the table below consists of two parts: the first ten cycles at the top, and the next ten cycles at the bottom.*)

- Instruction 1 is useless due to write-after-write, remove it.
- Instruction 3 stalls for decode logic, move it up.
- Instruction 6 does not have read-after-write dependency and can be executed before instr. 5. However, it cannot execute before instruction 4 as it would change the value of R7.

New total execution time is 17 cycles instead of 18.

Instr.	Instructions	Cycles																			
No		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	MUL R5, R6, R7	F	D	E1	E2	E3	E4	M	W												
3	MUL R4, R7, R7		F	D	E1	E2	E3	E4	M	W											
2	ADD R5, R5, R6			F	D	-	-	E1	E2	E3	M	W									
4	ADD R6, R7, R5				F	-	-	D	-	-	E1	E2	E3	M	W						
6	ADD R7, R1, R4							F	-	-	D	E1	E2	E3	M	W					
5	ADD R3, R0, R6										F	D	-	E1	E2	E3	M	W			

11 Tomasulo's Algorithm (Extra)

In this problem, we consider an in-order fetch, out-of-order dispatch, and out-of-order retirement execution engine that employs Tomasulo's algorithm. This engine behaves as follows:

- The engine has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The engine can fetch **FW** instructions per cycle, decode **DW** instructions per cycle, and write back the result of **RW** instructions per cycle.
- The engine has two execution units: 1) an *integer ALU* for executing integer instructions (i.e., addition and multiplication) and 2) a *memory unit* for executing load/store instructions.
- Each execution unit has an **R**-entry reservation station.
- An instruction always allocates the first available entry of the reservation station (in top-to-bottom order) of the corresponding execution unit.

The reservation stations are all initally empty. The processor fetches and executes *six* instructions. Table 3 shows the six instructions and their execution diagram.

Using the information provided above and in Table 3 (see the next page), fill in the blanks below with the configuration of the out-of-order microarchitecture. Write "Unknown" if the corresponding configuration cannot be determined using the information provided in the question.

The latency of the ALU and memory unit instructions:	ALU - 2 cycles, MU - 10 cycles
In which pipeline stage is an intruction dispatched?	Decode (D) stage
Number of entries of each reservation station (R):	Two entries each
Fetch width (FW):	2
Decode width (DW):	2
Retire width (RW):	Unknown
Is the integer ALU pipelined?	Unknown
Is the memory unit pipelined?	Yes
If applicable, between which stages is data forwarding implemented?	No data forwarding

Instruction/Cycle:	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	26	27	28	29	30	31	32	33
1: ADD R1 ← R0, R1	F	D	E1	E2	W																												
2: LD R2 ← [R1]	F	D	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W																	
3: ADDI R1 ← R1, #4		F	D	-	-	E1	E2	W																									
4: LD R3 ← [R1]		F	D	-	-	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W														
5: MUL R4 ← R2, R3			F	-	-	D	-	-	-	-	-	-	-	-	-	-	-	-	-	E1	E2	W											
6: ST [R0] ← R4			F	-	-	-	-	-	-	-	-	-	-	-	-	-	D	-	-	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W

Table 3: Execution diagram of the six instructions.

12 Tomasulo's Algorithm - Reverse Engineering (Extra)

Consider an in-order fetch, out-of-order dispatch, and in-order retirement execution engine that employs Tomasulo's algorithm. This engine has the following characteristics:

- The engine has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The engine can fetch one instruction per cycle, decode one instruction per cycle, and write back the result of one instruction per cycle.
- The engine has two execution units: 1) an adder to execute ADD instructions and 2) a multiplier to execute MUL instructions.
- The execution units are fully pipelined. The adder has two stages (E1-E2), and the multiplier has four stages (E1-E2-E3-E4). Execution of each stage takes one cycle.
- The adder has a two-entry reservation station, and the multiplier has a three-entry reservation station.
- An instruction always allocates the first available entry of the reservation station (in top-to-bottom order) of the corresponding execution unit.
- Full data forwarding is available, i.e., during the last cycle of the E stage, the tags and data are broadcast to the reservation station and the Register Alias Table (RAT). For example, an ADD instruction updates the reservation station entries of the dependent instructions in the E2 stage. So, the updated value can be read from the reservation station entry in the next cycle. Therefore, a dependent instruction can potentially begin its execution in the next cycle (after E2).
- The multiplier and adder have separate output data buses, which allow both the adder and the multiplier to update the reservation station and the RAT in the same cycle.
- An instruction continues to occupy a reservation station slot until it finishes the Write-back (W) stage. The reservation station entry is deallocated after the Write-back (W) stage.

12.1 Problem Definition

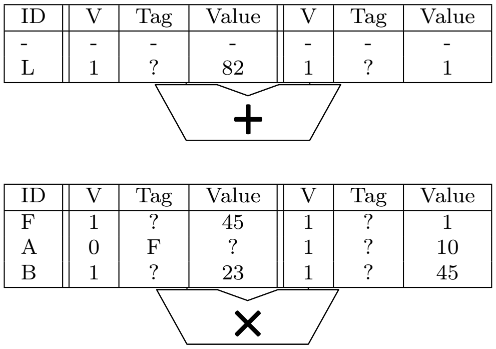
The processor is about to fetch and execute *five* instructions. Assume the *reservation stations (RS)* are all initially empty, and the initial state of the *register alias table (RAT)* is given below in Figure (a). Instructions are fetched, decoded, and executed as discussed in class. At some point during the execution of the five instructions, a snapshot of the state of the RS and the RAT is taken. Figures (b) and (c) show the state of the RS and the RAT at the snapshot time. A dash (–) indicates that a value has been cleared. A question mark (?) indicates that a value is unknown to you.

Reg	Valid	Tag	Value
R0	1	–	1900
R1	1	–	82
R2	1	–	1
R3	1	–	3
R4	1	–	10
R5	1	–	5
R6	1	–	23
R7	1	–	35
R8	1	–	61
R9	1	–	4

(a) Initial state of the RAT

Reg	Valid	Tag	Value
R0	1	?	1900
R1	1	?	82
R2	1	?	1
R3	1	?	45
R4	0	A	?
R5	0	F	?
R6	1	?	23
R7	1	?	35
R8	0	L	?
R9	0	B	?

(b) State of the RAT at the snapshot time



ID	V	Tag	Value	V	Tag	Value
-	-	-	-	-	-	-
L	1	?	82	1	?	1

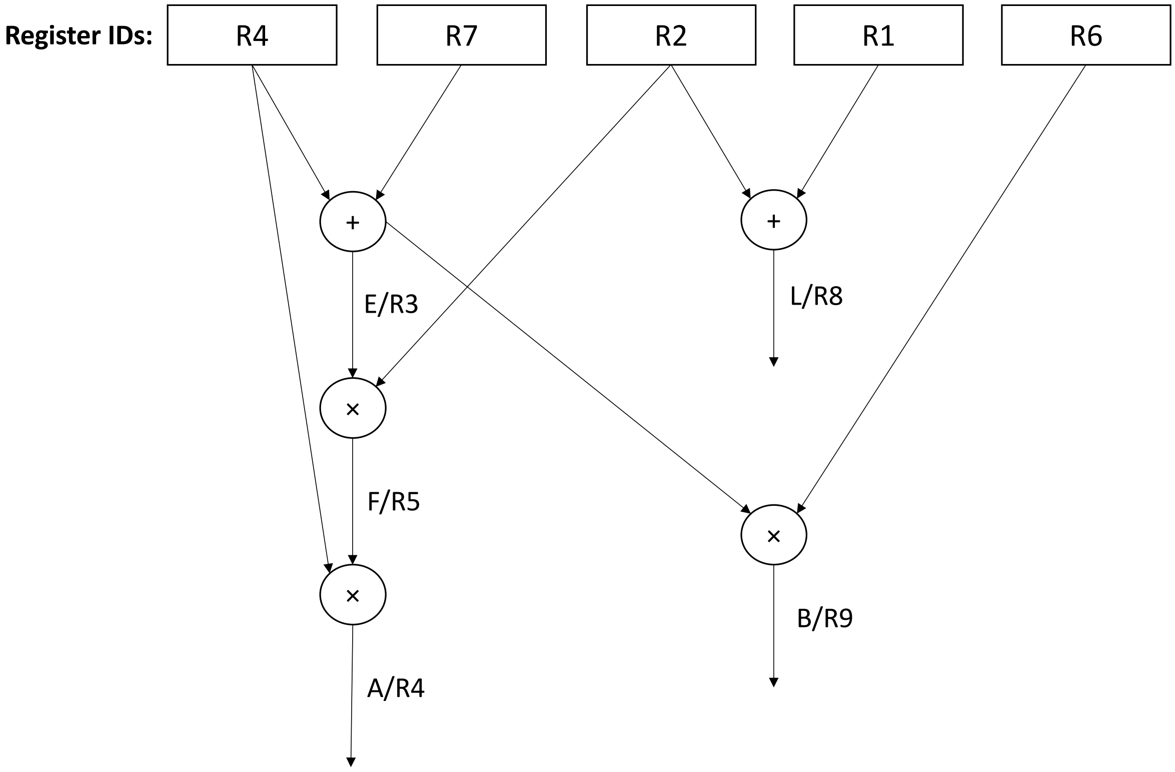
ID	V	Tag	Value	V	Tag	Value
F	1	?	45	1	?	1
A	0	F	?	1	?	10
B	1	?	23	1	?	45

(c) State of the RS at the snapshot time

12.2 Questions

12.2.1 Dataflow Graph

Based on the information provided above, identify the instructions and provide the dataflow graph below for the instructions that have been fetched. Please appropriately connect the nodes using edges and specify the direction of each edge. Label each edge with the destination architectural register and the corresponding Tag.



12.2.2 Program Instructions

Fill in the blanks below with the five-instruction sequence in program order. There can be more than one correct ordering. Please provide *only one* correct ordering. When referring to registers, please use their architectural names (R0 through R9). Place the register with the smaller architectural name on the left source register box.
For example, `ADD R8` ⇐ `R1, R5`.

ADD	R3	⇐	R4	,	R7
MUL	R5	⇐	R3	,	R2
MUL	R4	⇐	R5	,	R4
ADD	R8	⇐	R1	,	R2
MUL	R9	⇐	R6	,	R3

13 Out-of-Order Execution - Reverse Engineering (Extra)

In this problem, we will give you the state of the Register Alias Table (RAT) and Reservation Stations (RS) for an out-of-order execution engine that employs Tomasulo's algorithm. Your job is to determine the original sequence of **five instructions** in program order.

The out-of-order machine in this problem behaves as follows:

- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine dispatches one instruction per cycle into the reservation stations, in program order. Dispatch occurs during the decode stage.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.
- When a value is captured (at a reservation station) or written back (to a register) in this machine, the old tag that was previously at that location is *not cleared*; only the valid bit is set.
- When an instruction in a reservation station finishes executing, the reservation station is cleared.
- Both the adder and multiplier are fully pipelined. An add instruction takes 2 cycles. A multiply instruction takes 4 cycles.
- When an instruction completes execution, it broadcasts its result. A dependent instructions can begin execution in the next cycle if it has all its operands available.
- When multiple instructions are ready to execute at a functional unit, the oldest ready instruction is chosen.

Initially, the machine is empty. Five instructions then are fetched, decoded, and dispatched into reservation stations. When the final instruction has been fetched and decoded, one instruction has already been written back. Pictured below is the state of the machine at this point, after the fifth instruction has been fetched and decoded:

RAT

Reg	V	Tag	Value
R0	1		13
R1	0	A	8
R2	1		3
R3	1		5
R4	0	X	255
R5	0	Y	12
R6	0	Z	74
R7	1		7

ADD

	Src 1			Src2		
	Tag	V	Value	Tag	V	Value
A	-	1	5	Z	0	-
B						
C						

MUL

	Src 1			Src2		
	Tag	V	Value	Tag	V	Value
X	A	1	8	-	1	7
Y	X	0	-	-	1	13
Z	-	1	3	-	1	8

(a) Give the five instructions that have been dispatched into the machine, in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: "opcode destination ⇐ source1, source2."

```
ADD R1 ← R2, R3

MUL R4 ← R1, R7

MUL R5 ← R4, R0

MUL R6 ← R2, R1

ADD R1 ← R3, R6
```

(b) Now assume that the machine flushes all instructions out of the pipeline and restarts fetch from the first instruction in the sequence above. Show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use "F" for fetch, "D" for decode, "En" to signify the nth cycle of execution for an instruction, and "W" to signify writeback. You may or may not need all columns shown.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction:	F	D	E1	E2	W									
Instruction:		F	D		E1	E2	E3	E4	W					
Instruction:			F	D					E1	E2	E3	E4	W	
Instruction:				F	D	E1	E2	E3	E4	W				
Instruction:					F	D				E1	E2	W		

Finally, show the state of the RAT and reservation stations after **10 cycles** in the blank figures below.

RAT

Reg	V	Tag	Value
R0	1		13
R1	0	A	8
R2	1		3
R3	1		5
R4	1	X	56
R5	0	Y	12
R6	1	Z	24
R7	1		7

	Src 1			Src2		
	Tag	V	Value	Tag	V	Value
A	-	1	5	Z	1	24
B						
C						

	Src 1			Src2		
	Tag	V	Value	Tag	V	Value
X						
Y	X	1	56	-	1	13
Z						