

Name:

Student ID:

**Final Exam**  
**Design of Digital Circuits (252-0028-00L)**  
**ETH Zürich, Spring 2018**

Prof. Onur Mutlu

Problem 1 (30 Points):	Potpourri	
Problem 2 (30 Points):	Verilog	
Problem 3 (15 Points):	Boolean Algebra	
Problem 4 (50 Points):	Finite State Machine	
Problem 5 (45 Points):	ISA and Microarchitecture	
Problem 6 (35 Points):	Pipelining	
Problem 7 (45 Points):	Out-of-order Execution	
Problem 8 (40 Points):	Vector Processing	
Problem 9 (45 Points):	GPUs and SIMD	
Problem 10 (40 Points):	Memory Hierarchy	
Problem 11 (35 Points):	Dataflow Meets Logic	
Problem 12 (BONUS: 40 Points):	Branch Prediction	
Total (450 (410 + 40 bonus) Points):		

**Examination Rules:**

1. Written exam, 180 minutes in total.
2. No books, no calculators, no computers or communication devices. 6 pages of handwritten notes are allowed.
3. Write all your answers on this document, space is reserved for your answers after each question. Blank pages are available at the end of the exam.
4. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
5. Put your Student ID card visible on the desk during the exam.
6. If you feel disturbed, immediately call an assistant.
7. Write with a black or blue pen (no pencil, no green or red color).
8. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake.
9. Please write your initials at the top of every page.

**Tips:**

- **Be cognizant of time. Do not spend too much time on one question.**
- **Be concise. You may be penalized for verbosity.**
- **Show work when needed. You will receive partial credit at the instructors' discretion.**
- **Write legibly. Show your final answer.**

Initials: \_\_\_\_\_

Design of Digital Circuits

August 18th, 2018

*This page intentionally left blank*

## 1 Potpourri [30 points]

### 1.1 Microarchitecture or ISA? [10 points]

Based on your knowledge of a basic MIPS design and the computer architecture techniques you learned throughout this course, put an “X” in the box corresponding to whether each of the following design characteristics is *better* classified as “microarchitecture” or “ISA”:

Characteristic	Microarchitecture	ISA
General purpose register \$29 is the stack pointer		
Maximum bandwidth between the L2 and the L3 cache		
Maximum reservation station capacity		
Hardware floating point exception support		
Instruction issue width		
Vector instruction support		
Memory-mapped I/O Port Address		
Arithmetic and Logic Unit (ALU) critical path		
CPU endianness		
Virtual page size		

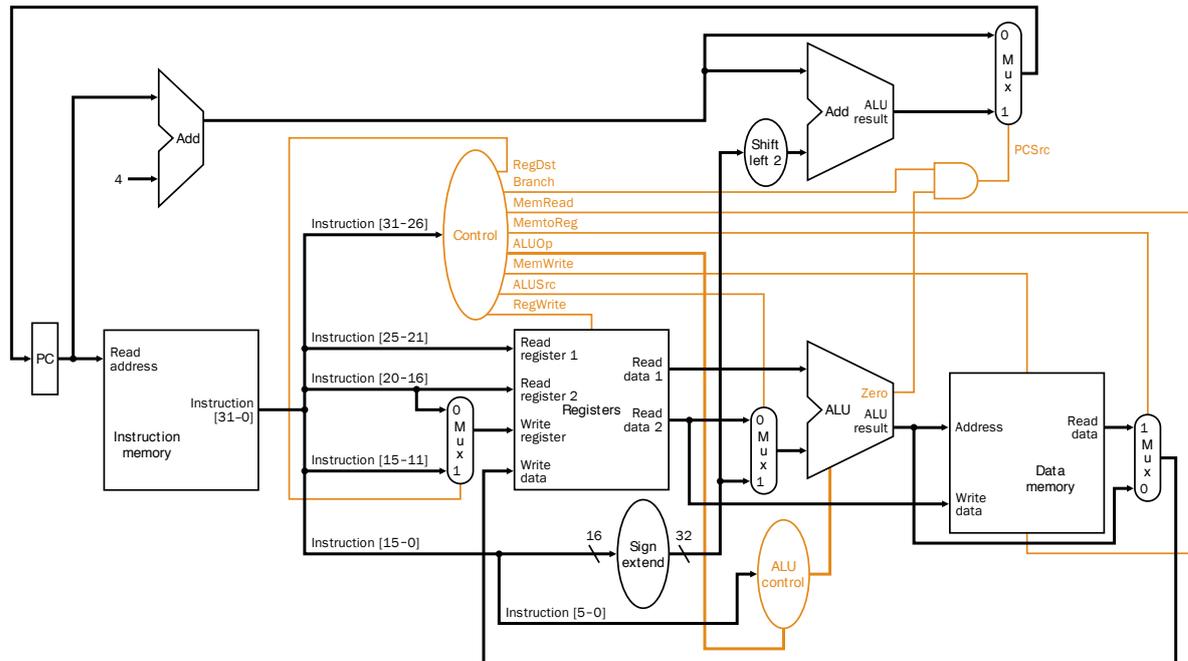
### 1.2 Single-Cycle Processor Datapath [10 points]

Modify the single-cycle processor datapath to include a version of the `lw` instruction, called `lw2`, that adds two registers to obtain the effective address. The datapath that you will modify is provided below. Your job is to implement the necessary data and control signals to support the new `lw2` instruction, which we define to have the following semantics:

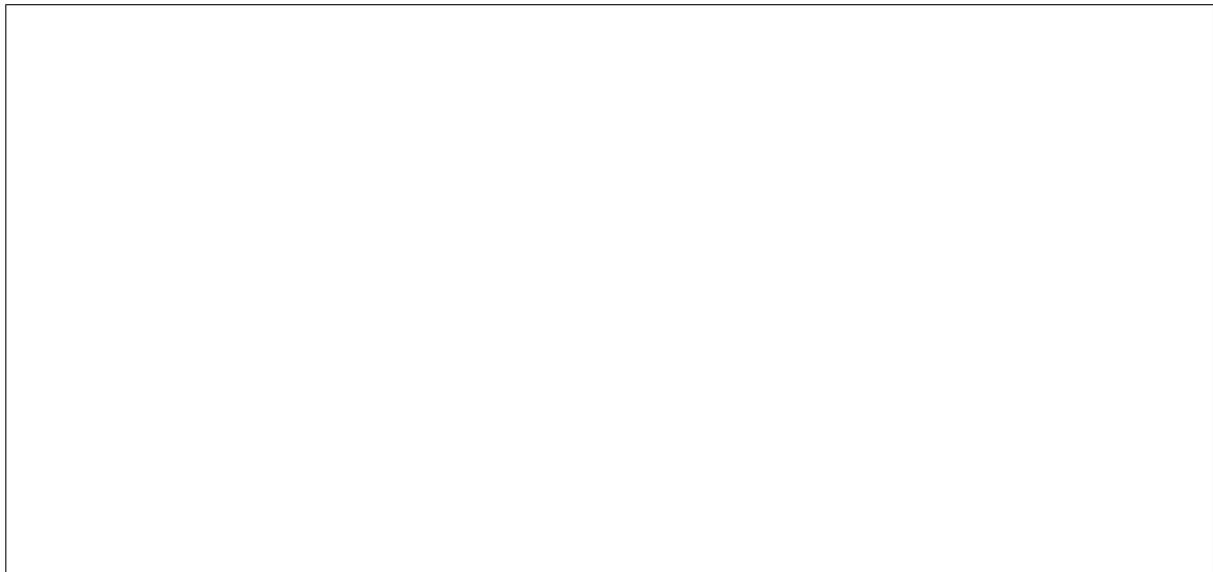
$$lw2: \quad Rd \leftarrow Memory[Rs + Rt]$$

$$PC \leftarrow PC + 4$$

Add to the datapath any necessary data and control signals (if necessary) to implement the `lw2` instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).



ALU opcode	Operation
00	Add
01	Subtract
10	Controlled by funct
11	Not used



**1.3 Performance Evaluation [10 points]**

The execution time of a given benchmark is  $100\text{ ms}$  on a  $500\text{ MHz}$  processor. An ETH alumnus, designing the next generation of the processor, notices that a new implementation enables the processor to run at  $750\text{ MHz}$ . However, the modifications increase the CPI by 20% for the same benchmark.

- (a) [4 points] What is the execution time expressed in terms of the number of cycles taken for the **old** generation of the processor (i.e., before the modifications)?

Assuming that the IPC is 2, what is the number of instructions in the benchmark?

- (b) [3 points] What is the execution time of the benchmark in *milliseconds* for the **new** generation of the processor?

- (c) [3 points] What is the speedup or slowdown of the new generation processor *over* the old generation?

## 2 Verilog [30 points]

Please answer the following four questions about Verilog.

- (a) [6 points] Does the following code result in a sequential circuit or a combinational circuit? Explain why.

```
1 module concat (input clk, input data_in1, input data_in2,
2               output reg [1:0] data_out);
3     always @ (posedge clk, data_in1)
4         if (data_in1)
5             data_out = {data_in1, data_in2};
6         else if (data_in2)
7             data_out = {data_in2, data_in1};
8     endmodule
```

Answer and concise explanation:

- (b) [6 points] In the following code, the input `clk` is a clock signal. What is the hexadecimal value of the output `c` right after the third positive edge of `clk` if initially `c = 8'hE3` and `a = 4'd8` and `b = 4'o2` during the entire time?

```
1 module mod1 (input clk, input [3:0] a, input [3:0] b, output reg [7:0] c);
2     always @ (posedge clk)
3         begin
4             c <= {c, &a, |b};
5             c[0] <= ^c[7:6];
6         end
7     endmodule
```

Please answer below. Show your work.

- (c) [6 points] Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
1 module lnn3r ( input [3:0] d, input op, output [1:0] s );
2   assign s = op ? (d[1:0] - d[3:2]) :
3                 (d[3:2] + d[1:0]);
4 endmodule
5
6 module top ( input wire [6:0] instr, input wire op, output reg z );
7
8   reg [1:0] r1, r2;
9
10  lnn3r i0 (.instr(instr[1:0]), .op(instr[7]), .z(r1) );
11  lnn3r i1 (.instr(instr[3:2]), .op(instr[0]), .z(r2) );
12  assign z = r1 | r2;
13
14 endmodule
```

Answer and concise explanation:

- (d) [6 points] Does the following code correctly implement a counter that counts from 1 to 11 by increments of 2 (e.g., 1, 3, 5, 7, 9, 11, 1, 3, ...)? If so, say "Correct". If not, correct the code with minimal modification.

```
1 module odd_counter (clk, count);
2   wire clk;
3   reg[2:0] count;
4   reg[2:0] count_next;
5
6   always@*
7   begin
8     count_next = count;
9     if(count != 11)
10      count_next = count_next + 2;
11    else
12      count_next <= 1;
13  end
14
15  always@(posedge clk)
16    count <= count_next;
17 endmodule
```

Answer and concise explanation:

- (e) [6 points] Does the following code correctly instantiate a 4-bit adder? If so, say "Correct". If not, correct the code with minimal modification.

```
1 module adder(input a, input b, input c, output sum, output carry);
2   assign sum = a ^ b ^ c;
3   assign carry = (a&b) | (b&c) | (c&a);
4 endmodule
5
6
7 module adder_4bits(input [3:0] a, input [3:0] b, output [3:0] sum, carry);
8   wire [2:0]s;
9
10  adder u0 (a[0],b[0],1'b0,sum[0],s[0]);
11  adder u1 (a[1],s[0],b[1],sum[1],s[1]);
12  adder u2 (a[2],s[1],b[2],sum[2],s[2]);
13  adder u3 (a[3],s[2],b[3],sum[3],carry);
14 endmodule
```

### 3 Boolean Algebra [15 points]

- (a) [5 points] Find the simplest sum-of-products representation of the following Boolean equation. Show your work step-by-step.

$$F = B + (A + \overline{C}).(\overline{A} + \overline{B} + \overline{C})$$

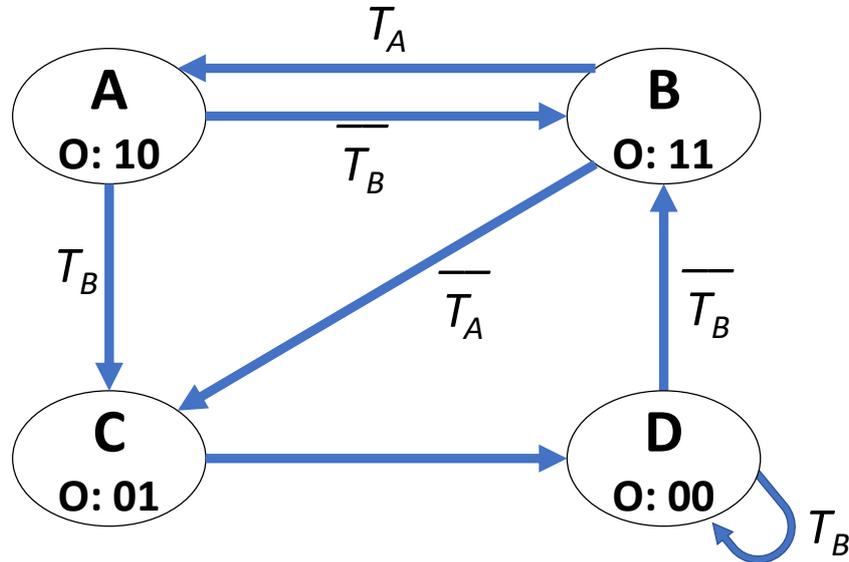
- (b) [5 points] Convert the following Boolean equation so that it only contains NAND operations. Show your work step-by-step.

$$F = \overline{(A + B.C)} + \overline{C}$$

- (c) [5 points] Using Boolean algebra, simplify the following min-terms:  $\sum(3, 5, 7, 11, 13, 15)$   
Show your work step-by-step.

### 4 Finite State Machine [50 points]

You are given the following FSM with two one-bit input signals ( $T_A$  and  $T_B$ ) and one two-bit output signal ( $O$ ). You need to implement this FSM, but you are unsure about how you should encode the states. Answer the following questions to get a better sense of the FSM and how the three different types of state encoding we discussed in the lecture (i.e., one-hot, binary, output) will affect the implementation.



- (a) [3 points] There is one critical component of an FSM that is *missing* in this diagram. Please write what is missing in the answer box below.

- (b) [2 points] Of the two FSM types, what type of an FSM is this?

(c) [5 points] List one major advantage of each type of state encoding below.

One-hot encoding

Binary encoding

Output encoding

(d) [10 points] Fully describe the FSM with equations given that the states are encoded with **one-hot** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with one-hot encoding. Indicate the values you assign to each state *and* simplify all equations:

- (e) [10 points] Fully describe the FSM with equations given that the states are encoded with **binary** encoding. Assign state encodings such that numerical values of states increase monotonically for states A through D while using the **minimum** possible number of bits to represent the states with binary encoding. Indicate the values you assign to each state *and* simplify all equations:

- (f) [10 points] Fully describe the FSM with equations given that the states are encoded with **output** encoding. Use the **minimum** possible number of bits to represent the states with output encoding. Indicate the values you assign to each state *and* simplify all equations:

(g) [10 points] Assume the following conditions:

- We can only implement our FSM with 2-input AND gates, 2-input OR gates, and D flip-flops.
- 2-input AND gates and 2-input OR gates occupy the *same* area.
- D flip-flops occupy 3x the area of 2-input AND gates.

Which state encoding do you choose to implement in order to **minimize the total area** of this FSM?

## 5 ISA and Microarchitecture [45 points]

You are asked to complete the following program written in MIPS assembly with a sequence of MIPS instructions that perform **64-bit integer subtraction (A - B)**. The 64-bit integer to be subtracted *from* (A) is loaded into registers \$4 and \$5. Similarly, the 64-bit integer to subtract (B) is loaded into registers \$6 and \$7. Both numbers are in two's complement form. The upper 32-bit part of each number is stored in the corresponding even-numbered register.

```
Loop:  lw  $4, 0($1)
      lw  $5, 4($1)
      lw  $6, 8($1)
      lw  $7, 12($1)

      # 64-bit subtraction
      # goes here

      addi $1, $1, 16
      j   Loop
```

- (a) [15 points] Complete the above program to perform the 64-bit subtraction explained above **using at most 4 MIPS instructions**. (*Note: A summary of the MIPS ISA is provided at the end of this question.*)

(b) [15 points] Assume that the program executes on a pipelined processor, which does *not* implement interlocking in hardware. The pipeline assumes that all instructions are independent and relies on the compiler to properly 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 `nops`. There is *no* internal register file forwarding (i.e., if an instruction writes into a register, another instruction *cannot* access the new value of the register until the next cycle). The pipeline does *not* implement any data forwarding. The datapath has the following *five pipeline stages*, similarly to the basic pipelined MIPS processor we discussed in lecture. Registers are accessed in the Decode stage. The execution stage contains *one ALU*.

- (a) Fetch (one clock cycle)
- (b) Decode (one clock cycle)
- (c) Execute (one clock cycle)
- (d) Memory (one clock cycle)
- (e) Write-back (one clock cycle).

Reorder the existing instructions and insert as few as possible `nop` instructions to correctly execute the entire program that you completed in part (a) on the given pipelined processor. Show all the instructions necessary to correctly execute the **entire program**.

- (c) [5 points] What is the *Cycles Per Instruction (CPI)* of the program when executed on the pipelined processor provided in part (b)?

- (d) [10 points] Now, assume a processor with a *multi-cycle* datapath. In this multi-cycle datapath, each instruction type is executed in the following number of cycles: 4 cycles for R-type, 5 cycles for load, 4 cycles for store, and 3 cycles for jump. What is the CPI of the program in part (a) when executed on this multi-cycle datapath? Assuming the multi-cycle datapath runs at the same clock frequency as the pipelined datapath in part (b), how much speedup does pipelining provide?

CPI:

Speedup:

## MIPS Instruction Summary

Opcode	Example Assembly	Semantics
add	add \$1, \$2, \$3	$\$1 = \$2 + \$3$
sub	sub \$1, \$2, \$3	$\$1 = \$2 - \$3$
add immediate	addi \$1, \$2, 100	$\$1 = \$2 + 100$
add unsigned	addu \$1, \$2, \$3	$\$1 = \$2 + \$3$
subtract unsigned	subu \$1, \$2, \$3	$\$1 = \$2 - \$3$
add immediate unsigned	addiu \$1, \$2, 100	$\$1 = \$2 + 100$
multiply	mult \$2, \$3	hi, lo = $\$2 * \$3$
multiply unsigned	multu \$2, \$3	hi, lo = $\$2 * \$3$
divide	div \$2, \$3	lo = $\$2/\$3$ , hi = $\$2 \bmod \$3$
divide unsigned	divu \$2, \$3	lo = $\$2/\$3$ , hi = $\$2 \bmod \$3$
move from hi	mfhi \$1	$\$1 = \text{hi}$
move from low	mflo \$1	$\$1 = \text{lo}$
and	and \$1, \$2, \$3	$\$1 = \$2 \& \$3$
or	or \$1, \$2, \$3	$\$1 = \$2   \$3$
and immediate	andi \$1, \$2, 100	$\$1 = \$2 \& 100$
or immediate	ori \$1, \$2, 100	$\$1 = \$2   100$
shift left logical	sll \$1, \$2, 10	$\$1 = \$2 \ll 10$
shift right logical	srl \$1, \$2, 10	$\$1 = \$2 \gg 10$
load word	lw \$1, 100(\$2)	$\$1 = \text{memory}[\$2 + 100]$
store word	sw \$1, 100(\$2)	$\text{memory}[\$2 + 100] = \$1$
load upper immediate	lui \$1, 100	$\$1 = 100 \ll 16$
branch on equal	beq \$1, \$2, label	if ( $\$1 == \$2$ ) goto label
branch on not equal	bne \$1, \$2, label	if ( $\$1 \neq \$2$ ) goto label
set on less than	slt \$1, \$2, \$3	if ( $\$2 < \$3$ ) $\$1 = 1$ else $\$1 = 0$
set on less than immediate	slti \$1, \$2, 100	if ( $\$2 < 100$ ) $\$1 = 1$ else $\$1 = 0$
set on less than unsigned	sltu \$1, \$2, \$3	if ( $\$2 < \$3$ ) $\$1 = 1$ else $\$1 = 0$
set on less than immediate unsigned	sltui \$1, \$2, 100	if ( $\$2 < 100$ ) $\$1 = 1$ else $\$1 = 0$
jump	j label	goto label
jump register	jr \$31	goto \$31
jump and link	jal label	$\$31 = \text{PC} + 4$ ; goto label

## 6 Pipelining [35 points]

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 `nops`. 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 updated value of 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 can forward 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 updated value of 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) [10 points] When the given code segment is executed on Machine I, the compiler has to reorder instructions and insert `nops` if needed. Write the resulting code that has *minimal modifications* from the original.

- (b) [10 points] 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.

- (c) [5 points] Calculate the *machine code size* of the code segments executed on Machine I (part (a)) and Machine II (part (b)).

Machine I:

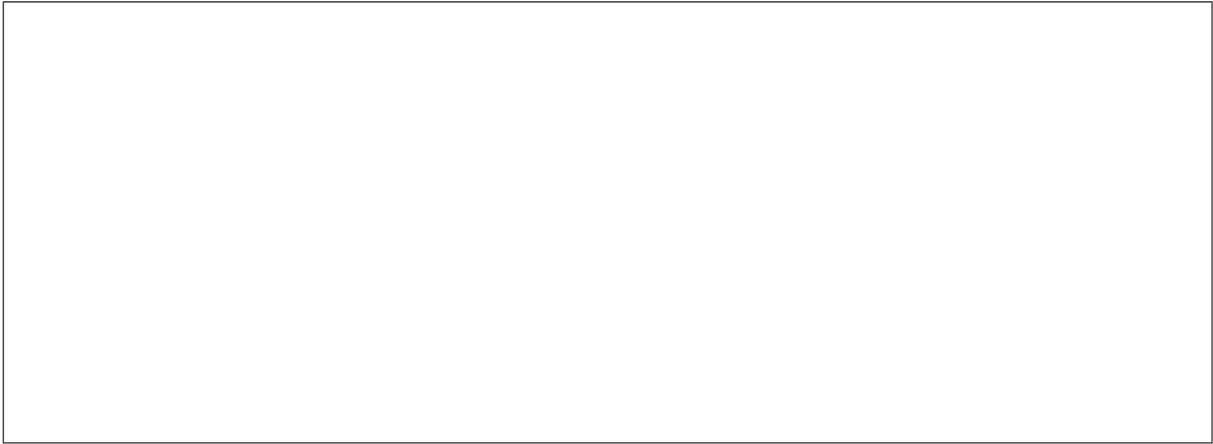
Machine II:

- (d) [7 points] Calculate the number of cycles it takes to execute the code segment on Machine I and Machine II.

Machine I:

Machine II:

(e) [3 points] Which machine is faster for this code segment? Explain.



## 7 Out-of-order Execution [45 points]

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 first task is to determine the original sequence of **four instructions** in program order.

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

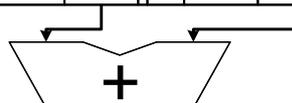
- The frontend of the machine has a one-cycle fetch stage and a one-cycle decode stage. The machine can fetch one instruction per cycle, and can decode one instruction per cycle.
- The machine executes *only* register-type instructions, e.g.,  $OP\ R_{dest} \leftarrow R_{src1}, R_{src2}$ , where  $OP$  can be  $ADD$  or  $MUL$ .
- 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. There is *only* one broadcast bus, and thus multiple instructions *cannot* broadcast in the same cycle.
- 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:

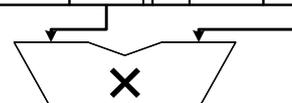
### RAT

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) [15 points] 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  $\leftarrow$  source1, source2.”

		$\leftarrow$		,	
		$\leftarrow$		,	
		$\leftarrow$		,	
		$\leftarrow$		,	

- (b) [15 points] 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. Fill in each instruction as well. You may or may not need all columns shown.

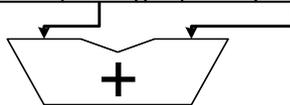
Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Inst.:																		
Inst.:																		
Inst.:																		
Inst.:																		
Inst.:																		

(c) [15 points] 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.

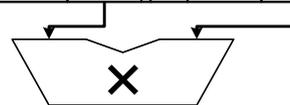
### RAT

Reg	V	Tag	Value
R0			
R1			
R2			
R3			
R4			
R5			

ID	V	Tag	Value	V	Tag	Value
A						
B						
C						



ID	V	Tag	Value	V	Tag	Value
D						
E						
F						



## 8 Vector Processing [40 points]

Assume a vector processor that implements the following ISA:

Opcode	Operands	Latency (cycles)	Description
SET	$V_{st}, \#n$	1	$V_{st} \leftarrow n$ ( $V_{st}$ = Vector Stride Register)
SET	$V_{ln}, \#n$	1	$V_{ln} \leftarrow n$ ( $V_{ln}$ = Vector Length Register)
VLD	$V_i, \#A$	100, pipelined	$V_i \leftarrow Mem[Address]$
VST	$V_i, \#A$	100, pipelined	$Mem[Address] \leftarrow V_i$
VMUL	$V_i, V_j, V_k$	10, pipelined	$V_i \leftarrow V_j * V_k$
VADD	$V_i, V_j, V_k$	5, pipelined	$V_i \leftarrow V_j + V_k$
VDIV	$V_i, V_j, V_k$	20, pipelined	$V_i \leftarrow V_j / V_k$

Assume the following:

- The processor has an in-order pipeline.
- The size of a vector element is 4 bytes.
- $V_{st}$  and  $V_{ln}$  are 10-bit registers.
- The processor *does not* support chaining between vector functional units.
- The main memory has  $N$  banks.
- Vector elements stored in consecutive memory addresses are interleaved between the memory banks. E.g., if a vector element at address  $A$  maps to bank  $B$ , a vector element at address  $A + 4$  maps to bank  $(B + 1) \% N$ , where  $\%$  is the modulo operator and  $N$  is the number of banks.  $N$  is *not necessarily* a power of two.
- The memory is byte addressable and the address space is represented using 32 bits.
- Vector elements are stored in memory in 4-byte-aligned manner.
- Each memory bank has a 4KB row buffer.
- Each memory bank has a single read and a single write port so that a load and a store operation can be performed simultaneously.
- There are separate functional units for executing VLD and VST instructions.

(a) [5 points] What should the minimum value of  $N$  be to avoid stalls while executing a VLD or VST instruction, assuming a vector stride of 1? Explain.

- (b) [8 points] What should the minimum value of  $N$  be to avoid stalls while executing a VLD or VST instruction, assuming a vector stride of 2? Explain.

- (c) [12 points] Assume:

- A machine that has a memory with as many banks as you found is part (a).
- The vector stride is set to 1.
- The value of the vector length is set to  $M$  (but we do *not* know  $M$ )

The machine executes the following program:

```
VLD V1 ← A
VLD V2 ← (A + 32768)
VADD V3 ← V1, V1
VMUL V4 ← V2, V3
VST (A + 32768*2) ← V4
```

The total number of cycles needed to complete the execution of the above program is 4306. What is  $M$ ?

- (d) [15 points] If we modify the vector processor to *support chaining*, how many cycles would be required to execute the same program in part (c)? Explain.

## 9 GPUs and SIMD [45 points]

We define the *SIMD utilization* of a program that runs on a GPU as the fraction of SIMD lanes that are kept busy with *active threads* during the run of the program. As we saw in lecture and practice exercises, the SIMD utilization of a program is computed across the *complete run* of the program.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays A and B are already in vector registers, so there are no loads and stores in this program. (Hint: Notice that there are 3 instructions in each iteration.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```
for (i = 0; i < 1025; i++) {
    if (A[i] < 33) {           // Instruction 1
        B[i] = A[i] << 1;    // Instruction 2
    }
    else {
        B[i] = A[i] >> 1;    // Instruction 3
    }
}
```

Please answer the following six questions.

- (a) [2 points] How many warps does it take to execute this program?

- (b) [10 points] What is the *maximum* possible SIMD utilization of this program? (Hint: The warp scheduler does *not* issue instructions when *no* threads are active).

- (c) [5 points] Please describe what needs to be true about array A to reach the maximum possible SIMD utilization asked in part (b). (Please cover all cases in your answer.)

- (d) [13 points] What is the *minimum* possible SIMD utilization of this program?

- (e) [5 points] Please describe what needs to be true about array A to reach the minimum possible SIMD utilization asked in part (d). (Please cover all cases in your answer.)

- (f) [10 points] What is the SIMD utilization of this program if  $A[i] = i$ ? Show your work.

## 10 Memory Hierarchy [40 points]

An enterprising computer architect is building a new machine for high-frequency stock trading and needs to choose a CPU. She will need to optimize her setup for *memory access latency* in order to gain a competitive edge in the market. She is considering two different prototype enthusiast CPUs that advertise high memory performance:

- (A) Dragonfire-980 Hyper-Z
- (B) Peregrine G-Class XTreme

She needs to characterize these CPUs to select the best one, and she knows from Prof. Mutlu's course that she is capable of reverse-engineering everything she needs to know. Unfortunately, these CPUs are not yet publicly available, and their exact specifications are unavailable. Luckily, important documents were recently leaked, claiming that the two CPUs have:

- Exactly 1 high-performance core
- LRU replacement policies (for any set-associative caches)
- Inclusive caching (i.e., data in a given cache level is present upward throughout the memory hierarchy. For example, if a cache line is present in L1, the cache line is also present in L2 and L3 if available.)
- Constant-latency memory structures (i.e., an access to any part of a given memory structure takes the same amount of time)
- Cache line, size, and associativity are all size aligned to powers of two

Being an ingenious engineer, she devises the following simple application in order to extract all of the information she needs to know. The application uses a high-resolution timer to measure the amount of time it takes to read data from memory with a specific pattern parameterized by *STRIDE* and *MAX\_ADDRESS*:

```
start_timer()
repeat N times:
    memory_address <- random_data()
    READ[(memory_address * STRIDE) % MAX_ADDRESS]
end_timer()
```

*Assume 1) this code runs for a long time, so all memory structures are fully warmed up, i.e., repeatedly accessed data is already cached, and 2) N is large enough such that the timer captures **only** steady-state information.*

By sweeping *STRIDE* and *MAX\_ADDRESS*, the computer architect can glean information about the various memory structures in each CPU.

She produces Figure 1 for CPU A and Figure 2 for CPU B.

**Your task:** Using the data from the graphs, reverse-engineer the following system parameters. If the parameter does *not make sense* (e.g., L3 cache in a 2-cache system), mark the box with an "X". If the graphs provide *insufficient information* to ascertain a desired parameter, simply mark it as "N/A".

(a) [15 points] Fill in the blanks for Dragonfire-980 Hyper-Z.

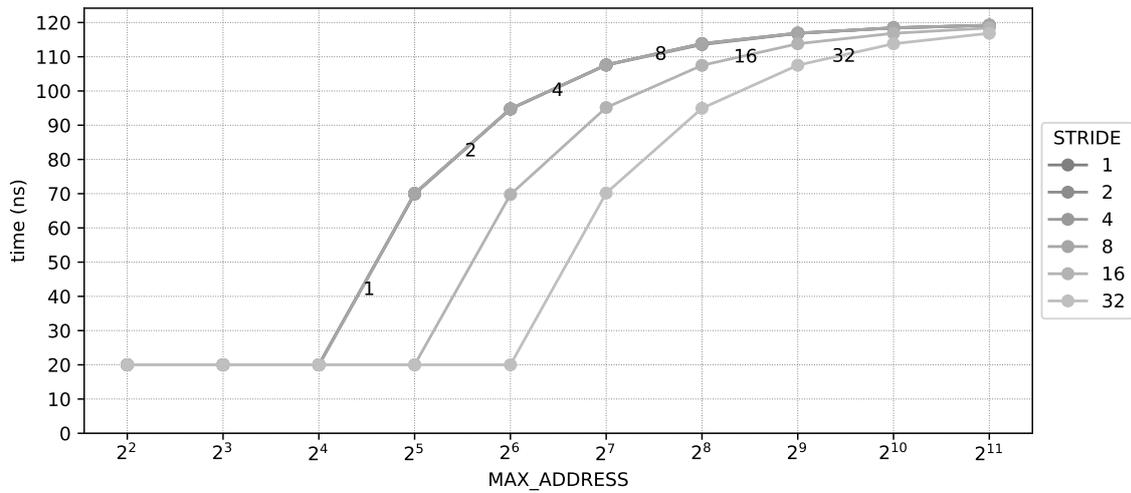


Figure 1: Execution time of the test code on CPU A for various values of *STRIDE* and *MAX\_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, and 8 overlap in the figure.

Table 1: Fill in the following table for CPU A (Dragonfire-980 Hyper-Z)

System Parameter	CPU A: Dragonfire-980 Hyper-Z			
	L1	L2	L3	DRAM
Cache Line Size (B)				
Cache Associativity				
Total Cache Size (B)				
Access Latency from (ns) <sup>1</sup>				

<sup>1</sup> DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

(b) [25 points] Fill in the blanks for Peregrine G-Class XTreme.

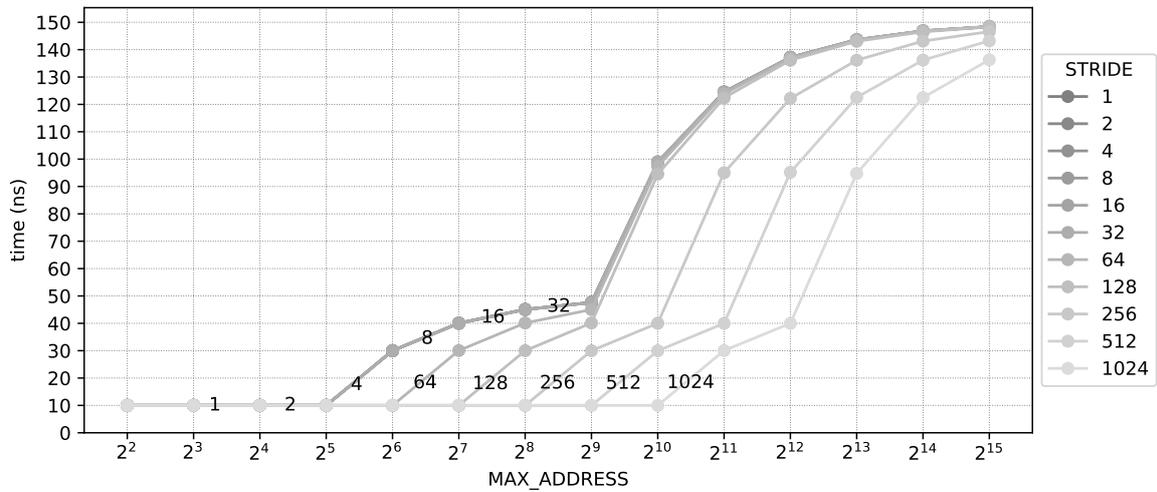


Figure 2: Execution time of the test code on CPU B for various values of *STRIDE* and *MAX\_ADDRESS*. *STRIDE* values are labeled on curves themselves for clarity. Note that the curves for strides 1, 2, 4, 8, 16, and 32 overlap in the figure.

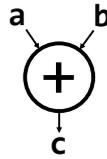
Table 2: Fill in the following table for CPU B (Peregrine G-Class XTreme)

System Parameter	CPU B: Peregrine G-Class XTreme			
	L1	L2	L3	DRAM
Cache Line Size (B)				
Cache Associativity				
Total Cache Size (B)				
Access Latency from (ns) <sup>1</sup>				

<sup>1</sup> DRAM access latency means the latency of fetching the data from DRAM to L3, *not* the latency of bringing the data from the DRAM all the way down to the CPU. Similarly, L3 access latency means the latency of fetching the data from L3 to L2. L1 access latency is the latency to bring the data to the CPU from the L1 cache.

## 11 Dataflow Meets Logic [35 points]

We often use the “addition node”:



to represent the addition of two input tokens. If we think of the tokens as binary numbers, we can model a simple logic circuit using dataflow graphs.<sup>1</sup> Note that a token can be used as an input to only *one* node. If the same value is needed by more than one node, it first should be replicated using one or more copy nodes, and then each copied token can be supplied to one node only.

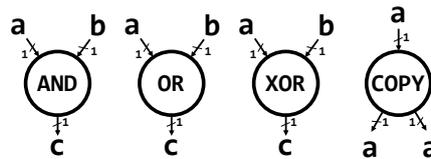
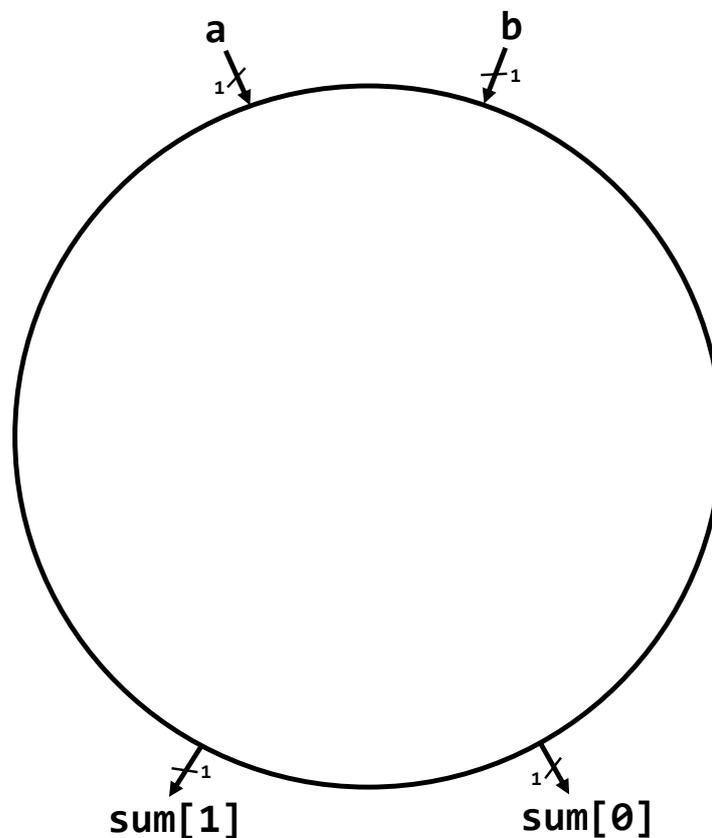


Figure 3: Dataflow nodes of basic bitwise operations allowed in Part (a).

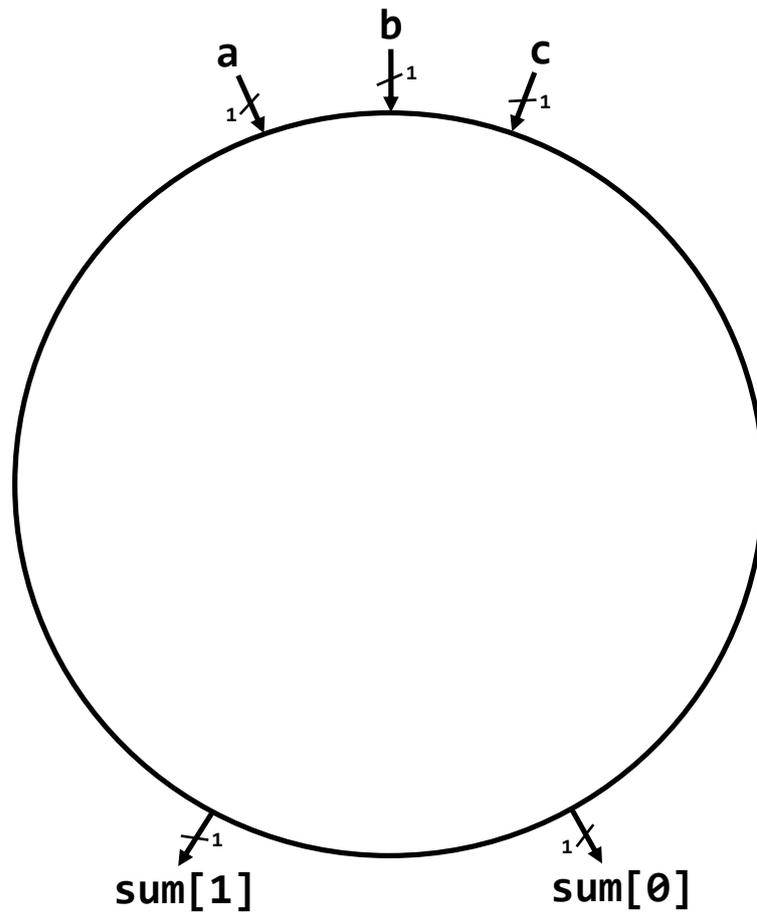
- (a) [5 points] Implement the single-bit binary addition of two “1-bit” input tokens  $a$  and  $b$  as a dataflow graph using *only* 2-input {AND, OR, XOR} nodes and COPY nodes if necessary (illustrated in Figure 3). Fill in the internal implementation below, where inputs and outputs (labeled with their corresponding bit-widths) have been provided:



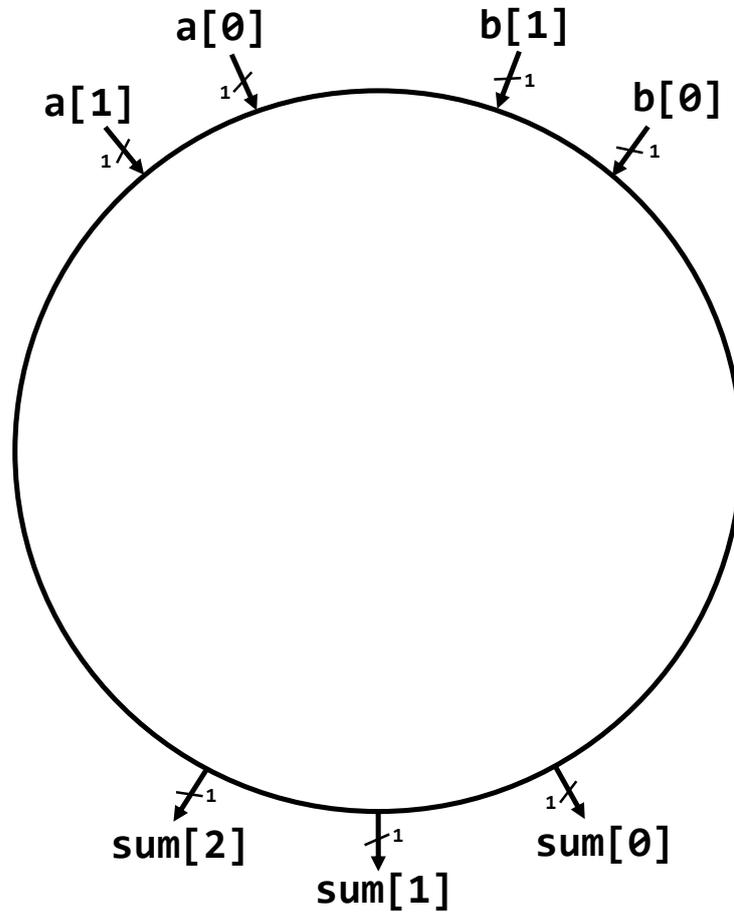
<sup>1</sup>Note: this is not an accurate electrical model of a circuit. Instead, the dataflow analogy is best thought of in terms of the desired flow of *information* rather than physical phenomena.

- (b) [5 points] You may recognize the node we designed in part (a) as a model for a so-called “half-adder (HA)”, which is not very useful by itself since it is only useful for adding 1-bit input tokens. In order to extend this design to perform binary addition of 2-bit input tokens  $a[1:0]$  and  $b[1:0]$ , the  $sum[1]$  token from half-adding  $a[0]$  and  $b[0]$  will have to act as an input token for *another* half-adder node used for adding  $a[1]$  and  $b[1]$ . This results in a 3-input adder called a “full-adder (FA)”.

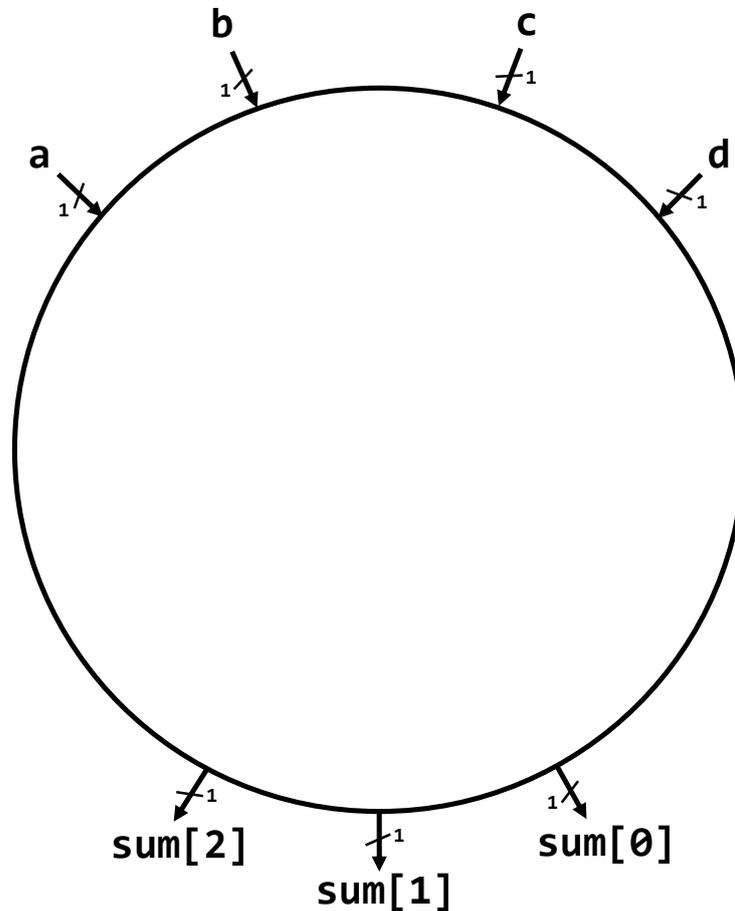
Fortunately, we can implement a full-adder (FA) using half-adders (HA) (i.e., the node we designed in part (a)). Implement the full-adder using a *minimum* number of half-adders and *at most* 1 additional 2-input {AND, OR, XOR} node.



- (c) [5 points] The full-adder (FA) is a versatile design that can be used to implement  $n$ -bit addition. Show how we might use it to implement 2-bit binary addition of two input tokens  $a[1:0]$  and  $b[1:0]$ . Use only a *minimum* number of full-adders (i.e., the dataflow node you designed in Part 2). *Hint: you may use constant input tokens if necessary.*



- (d) [5 points] Interestingly, the full-adder can also be used to add four 1-bit input tokens. This is a natural extension of the full-adder in the same way we extended the half-adder to create the full-adder itself (in part (b)). Implement the 4-input node below using only a *minimum* number of full-adders (FA) (i.e., the dataflow node you designed in part (b)). *Hint: you may use constant input tokens if necessary.*



- (e) [15 points] As it turns out, any  $n \geq 3$  1-bit input binary adders can be implemented purely using full-adders. Fill in the table below for the *minimum* number of required full adders to implement an  $n$ -input 1-bit adder.

$n$	# required full-adders
3	
4	
5	
6	
7	
8	

## 12 BONUS: Branch Prediction [40 points]

Assume a processor that implements an ISA with eight registers (R0-R7). In this ISA, the main memory is byte-addressable and each word contains 4 bytes. The processor employs a branch predictor. The ISA implements the instructions given in the following table:

Instructions	Description
la $R_i$ , Address	load the <i>Address</i> into $R_i$
move $R_i$ , $R_j$	$R_i \leftarrow R_j$
move $R_i$ , ( $R_j$ )	$R_i \leftarrow \text{Memory}[R_j]$
move ( $R_i$ ), $R_j$	$\text{Memory}[R_i] \leftarrow R_j$
li $R_i$ , Imm	$R_i \leftarrow \text{Imm}$
add $R_i$ , $R_j$ , $R_k$	$R_i \leftarrow R_j + R_k$
addi $R_i$ , $R_j$ , Imm	$R_i \leftarrow R_j + \text{Imm}$
cmp $R_i$ , $R_j$	Compare: Set sign flag, if $R_i < R_j$ ; set zero flag, if $R_i = R_j$
cmp $R_i$ , ( $R_j$ )	Compare: Set sign flag, if $R_i < \text{Memory}[R_j]$ ; set zero flag, if $R_i = \text{Memory}[R_j]$
cmpi $R_i$ , Imm	Compare: Set sign flag, if $R_i < \text{Imm}$ ; set zero flag, if $R_i = \text{Imm}$ .
jg label	Jump to the target address if <b>both</b> of sign and zero flags are zero.
jnz label	Jump to the target address if zero flag is zero.
halt	Stop executing instructions.

The processor executes the following program. Answer the questions below related to the accuracy of the branch predictors that the processor can potentially implement.

```

1      la R0, Array
2      move R6, R0
3      li R1, 4
4      move R5, R1
5      move R7, R1
6      move R2, R0
7      addi R2, R2, 4
8  Loop:
9      move R3, (R2)
10     cmp R3, (R0)
11     jg Next_Iteration
12     move R4, (R0)
13     move (R0), R3
14     move (R2), R4
15 Next_Iteration:
16     addi R0, R0, 4
17     addi R2, R2, 4
18     addi R1, R1, -1
19     cmpi R1, 0
20     jnz Loop
21     move R1, R7
22     addi R5, R5, -1
23     move R0, R6
24     move R2, R0
25     addi R2, R2, 4
26     cmpi R5, 0
27     jnz Loop
28     halt
29 .data
30 Array: word 5, 20, 1, -5, 34

```

- (a) [15 points] What would be the prediction accuracy using a global one-bit-history (last-time) branch predictor shared between *all* the branches? The initial state of the predictor is "taken".

- (b) [15 points] What would be the prediction accuracy using a global two-bit-history (two-bit counter) branch predictor shared between *all* the branches? Assume that the initial state of the two-bit counter is "weakly taken". The "weakly taken" state transitions to the "weakly not-taken" state on misprediction. Similarly, the "weakly not-taken" state transitions to the "weakly taken" state on misprediction. A correct prediction in one of the "weak" states transitions the state to the corresponding "strong" state.

- (c) [10 points] What would be the prediction accuracy using a local two-bit-history (two-bit counter) branch predictor that is separate for *each* branch? The initial state is "weakly taken" and the state transitions are the same as in part (b).