Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Digital Design and Computer Architecture (252-0028-00L)

ETH Zürich, Spring 2020

Prof. Onur Mutlu

Problem 1 (20 Points):	Boolean Circuit Minimization	
Problem 2 (40 Points):	Verilog	
Problem 3 (40 Points):	Finite State Machines	
Problem 4 (30 Points):	ISA vs. Microarchitecture	
Problem 5 (35 Points):	Performance Evaluation	
Problem 6 (45 Points):	Pipelining (Reverse Engineering)	
Problem 7 (50 Points):	Tomasulo's Algorithm	
Problem 8 (40 Points):	GPUs and SIMD	
Problem 9 (40 Points):	Caches (Reverse Engineering)	
Problem 10 (60 Points):	Branch Prediction	
Problem 11 (BONUS: 50 Points):	VLIW	
Total (450 (400 + 50 bonus) Points):		

Examination Rules:

1. Written exam, 180 minutes in total.
2. **No books, no calculators, no computers or communication devices**. 3 double-sided (or 6 one-sided) A4 sheets of handwritten notes are allowed.
3. Write all your answers on this document; space is reserved for your answers after each question.
4. You are provided with scratchpad sheets. Do not answer questions on them. **We will not collect them**.
5. Clearly indicate your final answer for each problem. Answers will only be evaluated if they are readable.
6. Put your Student ID card visible on the desk during the exam.
7. If you feel disturbed, immediately call an assistant.
8. Write with a black or blue pen (no pencil, no green, red or any other color).
9. Show all your work. For some questions, you may get partial credit even if the end result is wrong due to a calculation mistake. If you make assumptions, state your assumptions clearly and precisely.
10. 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.

This page intentionally left blank

1 Boolean Circuit Minimization [20 points]

(a) [10 points] Convert the following Boolean equation so that it only contains NAND operations. Show your work step-by-step.

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

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

Explanation:

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

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

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

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

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

(b) [10 points] Using Boolean algebra, find the simplest Boolean algebra equation for the following min-terms. Show your work step-by-step. You may label the order of variables as ABCD (e.g., $\overline{A} \cdot B \cdot \overline{C} \cdot \overline{D}$ denotes 0100).

$$\sum(0000, 0100, 0101, 1000, 1100, 1101)$$

$F = \overline{C} \cdot (B + \overline{D})$

Explanation:

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

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

$F = B \cdot \overline{C} + \overline{C} \cdot \overline{D}$

$F = \overline{C} \cdot (B + \overline{D})$

2 Verilog [40 points]

Please answer the following four questions about Verilog.

(a) [10 points] Does the following code result in a sequential circuit or a combinational circuit? Please explain your answer.

```

module sevensegment (input [3:0] data, output reg [6:0] segments);
    always @ ( * )
        case (data)
            4'd0: segments = 7'b111_1110;
            4'd1: segments = 7'b011_0000;
            4'd2: segments = 7'b110_1101;
            4'd3: segments = 7'b111_1001;
            4'd4: segments = 7'b011_0011;
        endcase
endmodule
```

Sequential circuit.

Explanation:
This code results in a sequential circuit, as all the left-hand side signals are not assigned in every possible condition. For example, for values of `data` that are more than 4, segment is not assigned to a specific value.

(b) [10 points] Does the following code result in an output signal which is zero except for one clock cycle in every three clock cycles (0-0-1-0-0-1...)? If not, please enable this functionality by adding minimal changes. Explain your answer.

```
module divideby3 (input clk, input reset, output q);
    reg [1:0] curVal, nextVal;
    parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;
    always @ (*)
        case (curVal)
            S0: nextVal = S1;
            S1: nextVal = S2;
            S2: nextVal = S0;
            default: nextVal = S0;
        endcase
    assign q = (curVal == S0);
endmodule
```

No.

Explanation:
The FSM misses state register. This leads to curVal not changing to nextVal. To fix the problem, the state register should be added:

always @ (posedge clk, posedge reset)
if (reset) curVal <= S0;
else curVal <= nextVal;

(c) [10 points] The following code implements a circuit and we initialize all inputs and registers of the circuit to zero. We apply the following changes to the input signals in two subsequent steps. What are the values of `out` and `tmp` after each step? Please show your work.

- **Step 1:** `sel` changes to 1.
- **Step 2:** While `sel` is still 1, `b` changes to 1.

```verilog
module mod1 (input sel, input a, input b, input c, output out);
    reg tmp = 1'b0;
    always @ (sel)
        if (sel)
            tmp <= ~(a & b);
            out <= tmp ^ c;
        else
            tmp <= 0;
            out <= 0;
endmodule
```

After step 1, tmp and out are respectively 1 and 0 . After step 2, tmp and out are respectively 1 and 0

Explanation:
After step 1: sel is in the sensitivity list of the always block. Therefore, the if statement executes. Since tmp and out are in the left hand side of non-blocking assignments, they execute concurrentlyi, and the second assignment in the if block does not see the new value of tmp when executing.
After step 2: The values of tmp and out do not change because b is not in the sensitivity list of the always block.

(d) [10 points] Is the following code syntactically correct and result in deterministic values for all signals? If not, please explain the mistake(s).

```verilog
module top (input [1:0] in1, in2 , input op, output reg [1:0] z, output reg s);

wire tmp;
always@(*) begin
    tmp = in1[0] & in2[0];
    z[0] = tmp & op;
end
always@(*) begin
    tmp = in1[1] | in2[1];
    z[1] = tmp & (~p)
end
assign s = (z[1] > z[0])
endmodule
```

The code is *not* syntactically correct and does not result in deterministic values.

Explanation:

- 'tmp' has to be declared as a *reg* since it is used in the always blocks.
- 's' should not to be declared as a *reg* since it is driven by the *assign* statement.
- 'tmp' signal is connected to multiple drivers and leads to non-deterministic values.

3 Finite State Machines [40 points]

3.1 Designing an FSM [20 points]

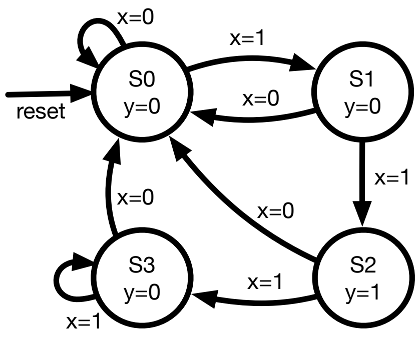
Draw a Moore finite state machine for a digital circuit that has a one-bit input x and a one-bit output y. The circuit detects the bit pattern 0-1-1 on the input x. The output bit is set (i.e., $y = 1$) during clock cycle t only if the three following values of x happen.

- $x = 0$ at clock cycle $t - 3$,
- $x = 1$ at clock cycle $t - 2$,
- $x = 1$ at clock cycle $t - 1$

Your state machine should use as few states as possible. Assume that the initial bit value of x is zero. Please clearly and comprehensively define each state and state transition. Note that you can lose points for ambiguity in your state machine.

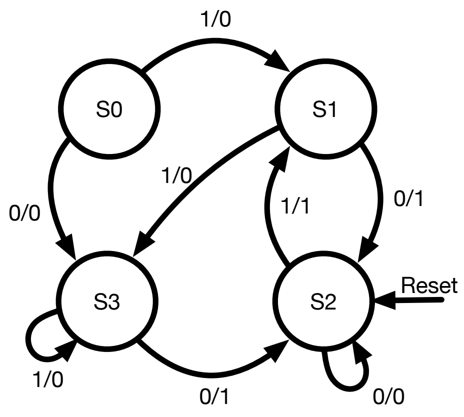
We need four states to keep track of the recent input values.

- S0: Input was zero in the last clock cycle (the initial state). The output should be zero.
- S1: Input was one in the last clock cycle, but it was zero before. The output should be zero.
- S2: Input was one in the last two clock cycles, but it was zero before. The output should be one.
- S3: Input was one in the last three clock cycles. The output should be zero.



3.2 Simplifying an FSM [20 points]

You are given the state machine of a *one-bit input / one-bit output* digital circuit design. Answer the following questions for the given state diagram.



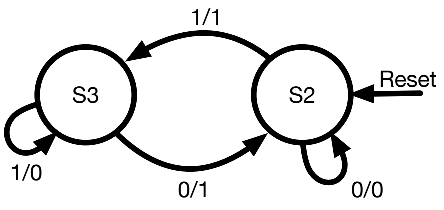
(a) [5 points] Is this a Mealy or a Moore machine? Explain why.

This is a Mealy machine because the output depends on both the state and the input.

(b) [10 points] Is it possible to simplify this state diagram to reduce the number of states? If so, simplify it to the minimum number of states. Explain each step of your simplification. Draw the simplified state diagram.

Yes, it is possible.

- There is no way the state goes to S0, so it is a non-used state. [2 points]
- S1 and S3 exhibit identical behavior, so they are redundant states. [3 points]
- We can simplify the state diagram as following.



(c) [5 points] What does this state machine do? For what purpose can it be useful? Explain.

This state machine outputs 1 when the input signal changes zero-to-one or one-to-zero. It outputs zero otherwise. (This is an edge detector. A student does not need to specify the name “edge detector” for getting full grade.)

4 ISA vs. Microarchitecture [30 points]

A new CPU has two comprehensive user manuals available for purchase as shown in Table 1.

Manual Title	**Cost**	**Description**
`the_isa.pdf`	CHF 1 million	describes the ISA in detail
`the_microarchitecture.pdf`	CHF 10 million	describes the microarchitecture in detail

Table 1: Manual Costs

Unfortunately, the manuals are extremely expensive, and you can only afford one of the two. If both manuals might be useful, you would prefer the cheaper one.
For each of the following questions that you would like to answer, decide which manual is more likely to help. *Note: we will subtract 1 point for each* ***incorrect*** *answer, and award 0 points for unanswered questions.*

1. [2 points] The integer multiplication algorithm used by the ALU.

   1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

2. [2 points] The program counter width.

   [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

3. [2 points] Branch misprediction penalty.

   1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

4. [2 points] The ability to flush the TLB from the OS.

   [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

5. [2 points] The size of the Reorder Buffer in an Out-of-Order CPU.

   1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

6. [2 points] The fetch width of a superscalar CPU.

   1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

7. [2 points] SIMD instruction support.

   [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

8. [2 points] The memory addresses of the memory-mapped devices of the CPU (e.g., keyboard).

   [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

9. [2 points] The number of non-programmable registers in the CPU.

   1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

10. [2 points] The replacement policy of the L1 data cache.

    1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

11. [2 points] The memory controller's scheduling algorithm.

    1. `the_isa.pdf` [✓ 2. `the_microarchitecture.pdf`]

12. [2 points] The number of bits required for the destination register of a load instruction.

    [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

13. [2 points] Description of the support for division and multiplication between integers.

    [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

14. [2 points] The mechanism to enter in a system call in the OS.

    [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

15. [2 points] The size of the addressable memory.

    [✓ 1. `the_isa.pdf`] 2. `the_microarchitecture.pdf`

5 Performance Evaluation [35 points]

A multi-cycle processor $P1$ executes *load instructions* in **10 cycles**, *store instructions* in **8 cycles**, *arithmetic instructions* in **4 cycles**, and *branch instructions* in **4 cycles**. Consider an application A where 20% of all instructions are load instructions, 20% of all instructions are store instructions, 50% of all instructions are arithmetic instructions, and 10% of all instructions are branch instructions.

(a) [5 points] What is the CPI of application A when executing on processor $P1$? Show your work.

$$CPI = 0.2 \times 10 + 0.2 \times 8 + 0.5 \times 4 + 0.1 \times 4$$
$$CPI = 6$$

(b) [10 points] A new design of the processor doubles the clock frequency of $P1$. However, the latencies of the load, store, arithmetic, and branch instructions increase by 2, 2, 2, and 1 cycles, respectively. We call this new processor $P2$. The compiler used to generate instructions for $P2$ is the same as for $P1$. Thus, it produces the same number of instructions for program A. What is the CPI of application A when executing on processor $P2$? Show your work.

$$CPI = 0.2 \times 12 + 0.2 \times 10 + 0.5 \times 6 + 0.1 \times 5$$
$$CPI = 7.9$$

(c) [10 points] Which processor is faster ($P1$ or $P2$)? By how much? Show your work.

$P2$ is 1.52× faster than $P1$.

Explanation.

$$Execution\_Time\_P1 = instructions \times CPI_{P1} \times clock\_rate$$
$$Execution\_Time\_P2 = instructions \times CPI_{P2} \times \frac{clock\_rate}{2}$$
$$clock\_rate = \frac{1}{clock\_frequency}$$

Assuming that $Execution\_Time\_P2 < Execution\_Time\_P1 \implies \frac{Execution\_Time\_P1}{Execution\_Time\_P2} > 1$. Thus:

$$\implies \frac{instructions \times CPI_{P1} \times clock\_rate}{instructions \times CPI_{P2} \times \frac{clock\_rate}{2}}$$
$$\implies \frac{6 \times clock\_rate}{7.9 \times \frac{clock\_rate}{2}}$$
$$\implies \frac{6}{3.95}$$
$$\implies 1.52$$

(d) [10 points] There is some extra area available in the chip of processor $P1$, where extra hardware can fit. You can decide to include in your processor a faster branch execution unit or a faster memory device. The faster branch execution unit reduces the latency of branch instructions by a factor of 4. The memory device reduces the latency of the memory operations by a factor of 2. Which design do you choose? Show your work.

A faster memory device.

Explanation.
Application A executes 10% of branch operations and 40% of memory operations (load and stores).
By Amdahl's Law, we have:

$Speedup_{branch} = \frac{1}{(1-0.1)+\frac{0.1}{4}} = 1.08$
$Speedup_{memory} = \frac{1}{(1-0.4)+\frac{0.4}{2}} = 1.25$

Therefore, the new memory device provides more speedup than the faster branch execution unit, for this particular application.

Alternative Solution.
In case we decide to reduce the latency of the branch operations, the new CPI of processor $P1$ will be:
$CPI_{branch} = 0.2 \times 10 + 0.2 \times 8 + 0.5 \times 4 + 0.1 \times \frac{4}{4}$
$CPI_{branch} = 5.7$

In case we decide to reduce the latency of the memory operations, the new CPI of processor $P1$ will be:
$CPI_{memory} = 0.2 \times \frac{10}{2} + 0.2 \times \frac{8}{2} + 0.5 \times 4 + 0.1 \times 4$
$CPI_{memory} = 4.2$

Since $CPI_{memory} < CPI_{branch}$, improving the memory device will provide shorter cycles-per-instructions.

6 Pipelining (Reverse Engineering) [45 points]

Algorithm 1 contains a piece of assembly code. Table 2 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
    JZ   L1            # Jump to L1 if zero flag is 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. Number	Instructions	Cycles 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	JZ L1								F	-	D	-	-	E1	...
7	...														

Table 2: 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) [10 points] 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 JZ at clock cycle 13).
There is no other information for any other data forwarding. Therefore, other data forwardings are unknown.

(b) [5 points] 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) [15 points] 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 2 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
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
4	NOP				F	D	E1	E2
5	ADD R1, R1, R2					F	D	E1
6	MUL R4, R2, R3						F	D
7	NOP							F
8	SUB R3, R1, 100							
9	NOP							
10	NOP							
11	JZ L1							
		8	9	10	11	12	13	14
3	NOP	M	W					
4	NOP	E3	M	W				
5	ADD R1, R1, R2	E2	E3	M	W			
6	MUL R4, R2, R3	E1	E2	E3	M	W		
7	NOP	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	JZ 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) [10 points] Calculate the value of T. Show your work.

T = 696.

Explanation.
Steady state throughput of an iteration is 4 instructions in 7 cycles.

If R1 = 98, this iteration is executed for 99 times so far.

At clock cycle T, there are still 7 cycles to finish iteration 99.

Initialization and filling the pipeline takes 10 cycles as shown below.

Then, $T = 10 + 99 \times 7 - 7 = 696$

(e) [5 points] Calculate the value of N. Show your work.

N = 398.

Explanation.
Loop iterates for 99 times before reaching to clock cycle T.

There are two instructions before the loop starts.

Then, $N = 2 + 99 \times 4 = 398$, assuming that the instruction indices start from 0.

7 Tomasulo's Algorithm [50 points]

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 3: Execution diagram of the seven instructions.

(a) [32 points] 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 $\rightarrow$ at least 2, MU $\rightarrow$ 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) [18 points] 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 $\leftarrow$ #0x1000
LD R1 $\leftarrow$ [R0]
BL R1 #100, #LB1
MUL R1 $\leftarrow$ R1, #5
ST [R0] $\leftarrow$ R1
Any number of unknown instructions can be here
LB1: ADD R1 $\leftarrow$ R1, R0
ST [R0] $\leftarrow$ R1

8 GPUs and SIMD [40 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 a 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. A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the 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. Both A and B are arrays of integers. (Hint Notice that there are 6 instructions in each thread.)

```
for (i = 0; i < 4096; i++) {
    if (B[i] < 8888) {      // Instruction 1
        A[i] = A[i] * C[i]; // Instruction 2
        A[i] = A[i] + B[i]  // Instruction 3
        C[i] = B[i] + 1;    // Instruction 4
    }
    if (B[i] > 8888) {      // Instruction 5
        A[i] = A[i] * B[i]; // Instruction 6
    }
}
```

Please answer the following four questions.

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

Warps = (Number of threads) / (Number of threads per warp) Number of threads = 2^{12} (i.e., one thread per loop iteration) Number of threads per warp = 64 = 2^6 (given) Warps = $2^{12}/2^6 = 2^6$

(b) [10 points] When we measure the SIMD utilization for this program with one input set, we find that it is 134/320. What can you say about arrays A,B, and C? Be precise. (Hint: Look at the "if" branch).

A. Nothing.
B. 2 in every 64 consecutive elements of B are less than 8888, the rest are exactly 8888.
C. Nothing.

(c) [10 points] What needs to be true about array `B` to achieve 100% utilization? Show your work. Be precise and complete. (Hint: The warp scheduler does not issue instructions where no threads are active).

Every 64 consecutive elements of B are either:
(1) equal to 8888,
(2) less than 8888,
(3) greater than 8888.

(d) [8 points] What is the minimum possible SIMD utilization of this program?

132/384.

(e) [10 points] What needs to be true about array `B` to achieve the minimum possible SIMD utilization? Show your work. (Please cover all cases in your answer.)

1 in every 64 of B's elements are greater than 8888, and 1 in every 64 of B's elements are less than 8888, and the rest of the elements are 8888.

9 Caches (Reverse Engineering) [40 points]

You are trying to reverse-engineer the characteristics of a cache in a system, so that you can design a more efficient, machine-specific implementation of an algorithm you are working on. To do so, you have come up with two sequences of memory accesses to various *bytes* in the system in an attempt to determine the following four cache characteristics:

- Cache block size (8, 16, 32, 64, or 128 B).
- Cache associativity (1-, 2-, 4-, or 8-way).
- Cache size (4 or 8 KiB).
- Cache replacement policy (LRU or FIFO).

The only statistic that you can collect on this system is *cache hit rate* after performing each sequence of memory accesses. Here is what you observe:

Sequence	Addresses Accessed (Oldest $\rightarrow$ Youngest)								Hit Rate
1.	0	32	128	73	8192	255	16384	196	1/2
2.	127	4096	8192	32768	196	16384	0	512	3/8

Assume that the cache is initially empty at the beginning of the first sequence, but *not* at the beginning of the second sequence. The sequences are executed back-to-back, i.e., no other accesses take place in between the two sequences. Thus, **at the beginning of the second sequence, the contents are the same as at the end of the first sequence**.

Based on what you observe, what are the following characteristics of the cache? Explain to get points. If a characteristic cannot be known, then write "Unknown" and explain.

(a) [10 points] Cache block size (8, 16, 32, 64, or 128 B)?

128 B.

Explanation:
Cache hit rate is 1/2 in sequence 1. This means that there are 4 hits. Depending on the cache block size, we can group addresses that belong to the same cache block as follows:

- **8–32 B:** {0}, {32}, {128}, {73}, {8192}, {255}, {16384}, {196}. $\therefore$ Number of possible hits = 0.
- **64 B:** {0, 32}, {128}, {73}, {8192}, {255, 196}, {16484}. $\therefore$ Number of possible hits = 2.
- **128 B:** {0, 32, 73}, {128, 255, 196}, {8192}, {16384}. $\therefore$ Number of possible hits = 4.

Therefore, we can know that the cache block size is 128 B.

(b) [10 points] Cache associativity (1-, 2-, 4-, or 8-way)?

4-way.

Explanation:

Cache hit rate is 3/8 in sequence 2, which means that there are 3 hits.
We already know that the cache block size is 128 B. Thus, there are 7 offset bits.

The access to address 196 in sequence 2 would hit because the cache block would not be replaced.
The access to address 512 in sequence 2 would miss because address 512 does not belong to any cache block previously accessed.
Therefore, the accesses to addresses 0, 127, 4096, 8192, 16834 and 32768 in sequence 2 would hit 2 times.
Regardless of cache size, those addresses will never hit when the cache were 1-way or 2-way.
If the cache were 8-way, those addresses would all map to set 0. With 8 ways, addresses 127, 9182, 16384 would not be replaced, so the three addressess would hit.
Therefore, the cache is 4-way associative.

(c) [10 points] Cache replacement policy (LRU or FIFO)?

LRU.

Explanation:
From questions (a) and (b), we already know the following facts:

- The cache block size is 128 B.
- The cache is 4-way.
- The accesses to addresses 0, 127, 4096, 8192, 16384, and 32768 in sequence 2 would hit 2 times.

Regardless of the cache size, 0, 127, 4096, 8192, 16834, and 32768 in sequence 2 would all map to set 0.
With the FIFO policy, accesses to addresses 127, 8192, and 16484 in sequence 2 would hit.
With the LRU policy, accesses to addresses 127 and 16384 would hit.
Therefore, the cache adopts the LRU policy.

(d) [10 points] To identify the cache size, you execute the following sequence right after sequence 2 (i.e., the contents are the same as at the end of the second sequence) and measure the cache hit rate:

Addresses Accessed (Oldest $\rightarrow$ Youngest): 8192 $\rightarrow$ X $\rightarrow$ Y

Which addresses should you use for X and Y?

$X = 1024 \times (2k - 1)$ where k is a positive integer.
$Y = 32768$

Explanation:
If the cache is 4-KiB, all addresses that are multiples of 1024 would map to set 0. If the cache is 8-KiB, all addresses that are multiples of 2048 would map to set 0.
After the access to 8192 in sequence 3, the LRU address in set 0 is 32768.
If the cache is 4-KiB, access to $X = 1024 \times (2k - 1)$ would replace 32768, so access to 32768 would miss. If the cache is 8-KiB, such access would not replace 32768, so access to 32768 would hit.

10 Branch Prediction [60 points]

A processor implements an *in-order* pipeline with *15 stages*. Each stage completes in a single cycle. The pipeline stalls on a conditional branch instruction until the condition of the branch is evaluated. However, you *do not* know at which stage the branch condition is evaluated. Please answer the following questions.

(a) [10 points] A program with 2500 dynamic instructions completes in 4514 cycles. If 500 of those instructions are conditional branches, at the end of which pipeline stage are the branch instructions resolved? (Assume that the pipeline does not stall for any other reason than conditional branches, e.g., data dependencies, during the execution of that program.)

At the end of the 5th stage.

Explanation: $Total\ cycles = 15 + 2500 + 500 * X - 1$
$4514 = 2514 + 500 * X$
$2000 = 500 * X$
$X = 4$
Each branch causes 4 idle cycles (bubbles), thus branches are resolved at the end of 5th stage.

(b) [2+3 points] In a new, higher-performance version of the previous processor, the architects implement a *mysterious* branch prediction mechanism to improve the performance of the processor. They keep the rest of the design exactly the same as before. The new design with the mysterious branch predictor completes the execution of the following piece of code in 136 cycles.

Please note that the number of pipeline stages and the stage at which the branch condition is evaluated are same as the previous question. Also, assume that the pipeline never stalls due to any other reasons than conditional branches.

```
MOV R1, #0 // R1 = 0

LOOP_1:
    BEQ R1, #5, LAST // Branch to LAST if R1 == 5
    ADD R1, R1, #1   // R1 = R1 + 1
    MOV R2, #0       // R2 = 0
LOOP_2:
    BEQ R2, #5, LOOP_1 // Branch to LOOP_1 if R2 == 5.
    ADD R2, R2, #1     // R2 = R2 + 1
    B LOOP_2           // Unconditional branch to LOOP_2

LAST:
    MOV R1, #1         // R1 = 0
```

How many instructions will be executed when running this piece of code? Show your work.

Total instructions executed = 98;

How many of them are CONDITIONAL branch instructions? Show your work.

Conditional branch instructions = 36;

(c) Based on the given information, determine which of the following branch prediction mechanisms could be the *mysterious* branch predictor implemented in the new version of the processor. For each branch prediction mechanism below, you should circle the configuration parameters that makes it match the performance of the mysterious branch predictor.

(I) [10 points] **Static Branch Predictor**

Could this be the mysterious branch predictor?

[✓ YES] NO

If YES, for which configuration below is the answer *YES*? Pick an option for each configuration parameter.

i. Static Prediction Direction

Always taken [✓ Always not taken]

Explain clearly to receive points.

YES, if the static prediction direction is *always not taken.*

Explanation: 98 instructions (36 of them are conditional branches) finishes execution in 136 cycles. This means there are 6 branch mispredictions. So, any predictor that produces 6 mispredictions can be our mysterious predictor.
A static predictor with always not taken predicton generates 6 mispredictions. Hence, YES.

(II) [15 points] **Last Time Branch Predictor**

Could this be the mysterious branch predictor?

YES [✓ NO]

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

i. Initial Prediction Direction

Taken Not taken

ii. Local for each branch instruction (i.e., PC-based) or global (i.e., shared among all branches) history?

Local Global

Explain clearly to receive points.

NO.

Explanation: There is no configuration for this branch predictor that results in 6 mispredictions for the above program.

Local-taken: 12 mispredictions, Local-NotTaken: 10 mispredictions,
Global-taken: 10 mispredictions, Global-NotTaken: 9 mispredictions.

(III) [5 points] **Backward taken, Forward not taken (BTFN)**

Please recollect, a conditional branch is said to be backward if its target address is lower than the branch PC, and vice-versa.

Could this be the mysterious branch predictor?

`YES` [✓ `NO`]

Explain clearly to receive points.

NO.

Explanation: BTFN predictor makes 26 mispredictions. Hence it cannot be our mysterious branch predictor.

(IV) [15 points] **Two-bit Counter Based Prediction** (using saturating arithmetic)

Could this be the mysterious branch predictor?

[✓ `YES`] `NO`

If YES, for which configuration is the answer *YES*? Pick an option for each configuration parameter.

i. Initial Prediction Direction

[✓ `00 (Strongly not taken)`] [✓ `01 (Weakly not taken)`]
`10 (Weakly taken)` `11 (Strongly taken)`

ii. Local for each branch instruction (i.e., PC-based, without any interference between different branches) or global (i.e., a single counter shared among all branches) history?

[✓ `Local`] [✓ `Global`]

Explain clearly to receive points.

YES, if *local* or *global* history registers with *00* or *01* initial values are used.

Explanation: Such a configuration yields exactly 6 mispredictions, which results in 136 cycles execution time for the above program.

11 BONUS: VLIW [50 points]

Consider a VLIW (very long instruction word) CPU that uses the long instruction format shown in Table 4. Each long instruction is composed of four short instructions, but there are restrictions on which type of instruction may go in which of the four slots.

MEMORY	INTEGER	CONTROL	FLOAT

Table 4: VLIW instruction format.

Table 5 provides a detailed description of the available short instructions and the total execution latency of each type of short instruction. Each short instruction execution unit is fully pipelined, and its result is available *on the cycle* given by the latency, e.g., a `CONTROL` instruction's results (if any) are available for other instructions to use *in the next cycle*.

Category	Latency (cycles)	Instruction(s)	Description	Functionality
`CONTROL`	`1`	`BEQ LABEL, Rs1, Rs2`	Branch IF equal	`IF Rs1 == Rs2: PC = LABEL`
		`NOP`	No operation	`PC = Next PC`
`MEMORY`	`3`	`LD Rd, [Rs]`	Memory load	`Rd = MEM[Rs]`
`INTEGER`	`2`	`IADD Rd, Rs1, Rs2`	Integer add	`Rd = Rs1 + Rs2`
`FLOAT`	`4`	`FADD Rd, Rs1, Rs2`	Floating-point add	`Rd = Rs1 + Rs2`

Table 5: Instruction latencies and descriptions.

Consider the piece of code given in Table 6. Unfortunately, it is written in terms of short instructions that cannot be directly input to the VLIW CPU.

	Instruction		Notes
	`< Initialize R0-R2 >`		`R0-R2 point to valid memory`
	`LOOP:`		
1	`LD`	`R0, [R0]`	`R0 <- MEM[R0]`
2	`LD`	`R1, [R1]`	`R1 <- MEM[R1]`
3	`IADD`	`R4, R0, R1`	`R4 <- R0 + R1`
4	`FADD`	`R5, R0, R4`	`R5 <- R0 + R4`
5	`LD`	`R6, [R2]`	`R6 <- MEM[R2]`
6	`LD`	`R2, [R0]`	`R2 <- MEM[R0]`
7	`FADD`	`R3, R1, R6`	`R3 <- R1 + R6`
8	`IADD`	`R4, R2, R4`	`R4 <- R2 + R4`
9	`IADD`	`R5, R5, R4`	`R5 <- R5 + R4`
10	`IADD`	`R0, R6, R2`	`R0 <- R6 + R2`
11	`IADD`	`R0, R0, R3`	`R0 <- R0 + R3`
12	`BEQ`	`LOOP, R0, R5`	`GOTO LOOP if R0 == R5`

Table 6: Proposed code for calculating the results of the next Swiss referendum.

(a) [10 points] Warm-up: which of the following are goals of VLIW CPU design (circle all that apply)?

(i) Simplify code compilation.

(ii) Simplify application development.

(iii) [✓ Reduce overall hardware complexity.]

(iv) [✓ Simplify hardware inter-instruction dependence checking.]

(v) Reduce processor fetch width.

(b) [25 points] Your task is to determine the optimal VLIW scheduling of the short instructions by hand. Fill in the following table with the highest performance (i.e., fewest number of execution cycles) instruction sequence that may be directly input into the VLIW CPU and have the same functionality as the code in Table 6. Where possible, you may write instruction IDs corresponding to the numbers given in Table 6 and leave any `NOP` instructions as blank slots.

Consider **only one loop iteration** (including the `BEQ` instruction), **ignore initialization** and any cross-iteration optimizations (e.g., loop unrolling), and **do not** optimize the code by removing or changing existing instructions.

Cycle	**MEMORY**	**INTEGER**	**CONTROL**	**FLOAT**
1	1 (`LD R0, [R0]`)			
2	2 (`LD R1, [R1]`)			
3	5 (`LD R6, [R2]`)			
4	6 (`LD R2, [R0]`)			
5		3 (`IADD R4, R0, R1`)		
6				7 (`FADD R3, R1, R6`)
7		8 (`IADD R4, R2, R4`)		4 (`FADD R5, R0, R4`)
8		10 (`IADD R0, R6, R2`)		
9		8 (`IADD R4, R2, R4`)		
10		11 (`IADD R0, R0, R3`)		
11		9 (`IADD R5, R5, R4`)		
12				
13			12 (`BEQ LOOP, R0, R5`)	
14				
15				
16				
17				
18				
19				
20				

Hint: you should not require more than 20 cycles.

Note: Instruction 8 may go in EITHER of the red slots.

(c) [5 points] How many total cycles are required to complete execution of all instructions in the previous question? Ignore pipeline fill overheads and assume the instruction latencies given in Table 5.

14

(d) [10 points] What is the utilization of the instruction scheduling slots (computed as the ratio of utilized slots to total execution slots throughout execution)?

(12 slots used) / (14 cycles * 4 slots/cycle) $= \frac{3}{14} \approx 21.4\%$