Family Name: SOLUTIONS First Name: Student ID:

Final Exam

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

ETH Zürich, Spring 2023

Prof. Onur Mutlu

Problem 1 (45 Points):	Boolean Logic Circuits	
Problem 2 (45 Points):	Finite State Machines	
Problem 3 (45 Points):	ISA vs. Microarchitecture	
Problem 4 (60 Points):	Verilog	
Problem 5 (45 Points):	Memory Potpourri	
Problem 6 (50 Points):	Performance Evaluation	
Problem 7 (70 Points):	Pipelining	
Problem 8 (80 Points):	Vector Processing	
Problem 9 (60 Points):	VLIW	
Problem 10 (50 Points):	Caches	
Problem 11 (BONUS: 50 Points):	Systolic Arrays	
Problem 12 (BONUS: 50 Points):	Prefetching	
Total (650 (550+100 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 Logic Circuits [45 points]

During your job interview, you are asked to design a combinational circuit with a four-bit input, $\{A, B, C, D\}$ (A is the most significant bit and D is the least significant bit), and two 1-bit outputs, Fib and $G3$. The value of each output is determined as follows:

- The output Fib is 1 only when the input 4-bit number is a Fibonacci number. You can calculate Fibonacci numbers as follows, $f(0) = 0$, $f(1) = 1$, and $f(n) = f(n-1) + f(n-2)$ for $n \geq 2$.
- The output $G3$ is 1 only when the input 4-bit number is greater than 3.
- Otherwise, the corresponding output is zero.

Please answer the following three questions.

(a) [10 points] Fill in the missing entries in the truth table below for the combinational circuit you are designing and express the output Fib in the *sum of products* representation.

Inputs				**Outputs**	
A	B	C	D	Fib	$G3$
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	1	0	1

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

(b) [15 points] Simplify the Fib expression using Boolean minimization rules. Show your work step-by-step.

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

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

$Fib = ((\overline{A} \cdot \overline{B}) \cdot (1)) + (\overline{A} \cdot B \cdot \overline{C} \cdot D) + (A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}) + (A \cdot B \cdot \overline{C} \cdot D)$

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

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

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

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

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

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

(c) [20 points] Find the simplest representation of the $G3$ output by using *only* 2-input NAND gates. Show your work step-by-step.

$G3 = \overline{\overline{(A \cdot A)} \cdot \overline{(B \cdot B)}}$

Explanation:

$G3 = (\overline{A} \cdot B \cdot \overline{C} \cdot \overline{D}) + (\overline{A} \cdot B \cdot \overline{C} \cdot D) + (\overline{A} \cdot B \cdot C \cdot \overline{D}) + (\overline{A} \cdot B \cdot C \cdot D) + (A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}) + (A \cdot \overline{B} \cdot \overline{C} \cdot D) + (A \cdot \overline{B} \cdot C \cdot \overline{D}) + (A \cdot \overline{B} \cdot C \cdot D) + (A \cdot B \cdot \overline{C} \cdot \overline{D}) + (A \cdot B \cdot \overline{C} \cdot D) + (A \cdot B \cdot C \cdot \overline{D}) + (A \cdot B \cdot C \cdot D)$

$G3 = (\overline{A} \cdot B \cdot ((\overline{C} \cdot \overline{D}) + (\overline{C} \cdot D) + (C \cdot \overline{D}) + (C \cdot D))) + (A \cdot \overline{B} \cdot ((\overline{C} \cdot \overline{D}) + (\overline{C} \cdot D) + (C \cdot \overline{D}) + (C \cdot D))) + (A \cdot B \cdot ((\overline{C} \cdot \overline{D}) + (\overline{C} \cdot D) + (C \cdot \overline{D}) + (C \cdot D)))$

$G3 = (\overline{A} \cdot B \cdot (1)) + (A \cdot \overline{B} \cdot (1)) + (A \cdot B \cdot (1))$

$G3 = (\overline{A} \cdot B) + (A \cdot \overline{B}) + (A \cdot B)$

$G3 = A + B$

$G3 = \overline{\overline{A + B}}$

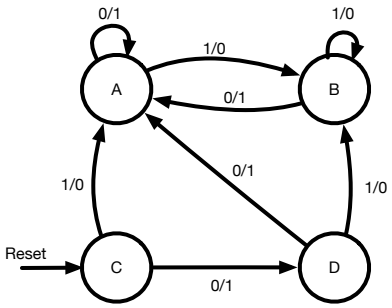
$G3 = \overline{\overline{A} \cdot \overline{B}}$

$G3 = \overline{\overline{(A \cdot A)} \cdot \overline{(B \cdot B)}}$

2 Finite State Machines [45 points]

2.1 Simplifying an FSM [20 points]

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



Is it possible to simplify this state diagram and 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. If *not*, explain why it is *not* possible to simplify the state diagram.

Yes, it is possible. Below is the state transition table of the given state machine:

Current State	Input	Next State	Output
A	0	A	1
A	1	B	0
B	0	A	1
B	1	B	0
C	0	D	1
C	1	A	0
D	0	A	1
D	1	B	0

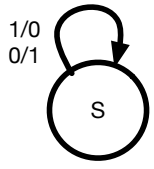
We can see that the states A, B, and D are identical. For all of these states,

- an input of 0 leads to the next state A and the output 1
- an input of 1 leads to the next state B and the output 0

Therefore, we can merge states A, B, and D. Let's use the name X:

Current State	Input	Next State	Output
X	0	X	1
X	1	X	0
C	0	X	1
C	1	X	0

We can further simplify this state machine as both states C and X are identical in terms of their next state and output. As a result, this state machine has *only* one state and the output is always the inverse of the input.



2.2 Designing an FSM [25 points]

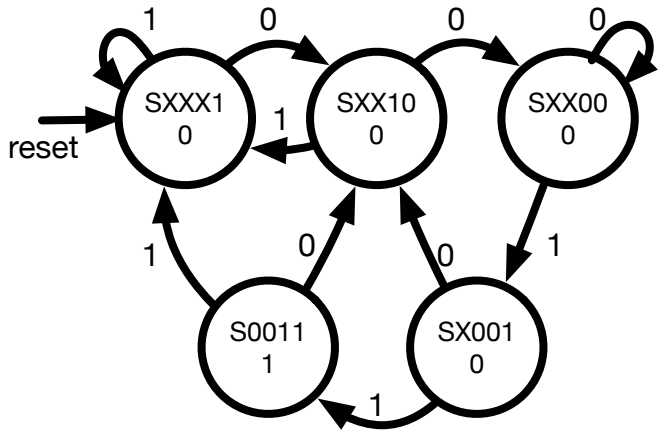
Design a Moore finite state machine (FSM), where each output is solely determined by the current state of the machine and *not* directly influenced by the inputs. The state machine should have one input and one output. This FSM's goal is to detect a stable transition in the input signal from repeated logic-0 to repeated logic-1. The output should be logic-1 *only* when the input sequence of "0-0-1-1" is observed. The output should be zero in all other cases.

When the circuit is reset, your state machine should assume that the input signal has been high (logic-1) for a long time. Draw the state diagram and explain why it works. Your state machine should use as few states as possible and each state should have a precise definition and output.

We need to keep track of the bit values in the last four bits. This requires 16 states. However, many of these states behave the same. We can reduce the number of states down to five.

- Since this is a Moore machine, the output should be independent from the input. Therefore, there should be a state for the posedge where the output is "1". All other states will have the output of "0". The FSM goes to the posedge state only when the last four bits are 0-0-1-1. We call this state S0011.
- The FSM should reach to the posedge state from another state where the last three input bits are 0-0-1. We call this state SX001.
- The FSM should reach to the 0-0-1 state from a state where the last two input bits are 0-0. Note that it does not matter what the input bits are, earlier than the last two zeros. We call this state SXX01.
- The FSM should not stay in state S0011 for more than one clock cycle as when the new input comes, the last four bits will not be 0-0-1-1 anymore. If the input is 1, the next state should be SXXX1: the last bit is zero but it is not a posedge and the earlier bits are not important. If the input is 0, the next state should be SXX10: the last two bits are 1-0 and the earlier bits are not important.
- Intuitively, if the state is SXXX1, the FSM should remain at this state if the input is 1 and go to SXX10 if the input is 0.
- If the state is SXX10, the FSM should *not* remain at this state regardless of the input. If the input is 0, the next state is SXX00. If the input is 1, the next state is SXXX1.

Therefore, it is possible to design this state machine with five states. The state diagram is shown below.



3 ISA vs. Microarchitecture [45 points]

Circle whether each of the following is an aspect of the ISA or the microarchitecture.

Note: we will subtract 2 points for each ***incorrect*** *answer and award 0 points for each unanswered question.*

1. [3 points] Width of the immediate value in an ADD instruction.

   [1. ISA] 2. Microarchitecture

2. [3 points] The algorithm used by the ALU to perform multiplication.

   1. ISA [2. Microarchitecture]

3. [3 points] Number of bits required for indexing the source register of a store instruction.

   [1. ISA] 2. Microarchitecture

4. [3 points] Number of entries in the L3 cache.

   1. ISA [2. Microarchitecture]

5. [3 points] The data cache organization (e.g., direct-mapped, set-associative).

   1. ISA [2. Microarchitecture]

6. [3 points] Support for conveying prefetching hints to the hardware via the compiler.

   [1. ISA] 2. Microarchitecture

7. [3 points] Available data types (e.g., integer) for arithmetic and logic operations.

   [1. ISA] 2. Microarchitecture

8. [3 points] Cache coherence protocol in multi-core processors.

   1. ISA [2. Microarchitecture]

9. [3 points] Width of the data bus between the processor and main memory.

   1. ISA [2. Microarchitecture]

10. [3 points] The memory controller's memory request scheduling algorithm.

    1. ISA [2. Microarchitecture]

11. [3 points] Instruction encoding for control flow and branch instructions.

    [1. ISA] 2. Microarchitecture

12. [3 points] The design of the register renaming logic.

    1. ISA [2. Microarchitecture]

13. [3 points] Number of instructions decoded per cycle in a superscalar processor.

    1. ISA [2. Microarchitecture]

14. [3 points] L2 cache miss latency.

    1. ISA [2. Microarchitecture]

15. [3 points] Width of the program counter.

    [1. ISA] 2. Microarchitecture

4 Verilog [60 points]

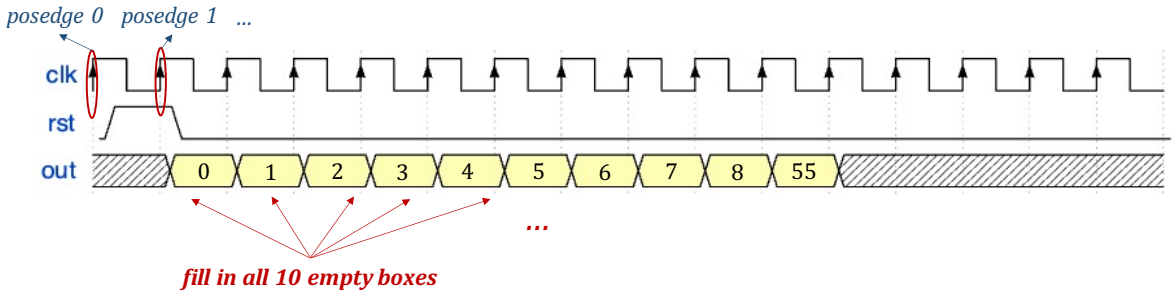
4.1 What Does This Code Do? [30 points]

Analyze the following Verilog module and answer the question.

```
1  module module_x (input wire clk, input wire rst,
2    input wire [7:0] in, output wire [7:0] out);
3
4  reg [7:0] var1, var2, var3, var4;
5
6  assign out = (var4 == in) ? var3 : var4;
7
8  always @(posedge clk) begin
9    if (rst) begin
10     var1  <= 8'b0;    var2  <= 8'b1;
11     var3  <= 8'b0;    var4  <= 8'b0;
12   end else begin
13     var1  <= var2;    var2  <= var1 + var2;
14     var3  <= var1 + var2;
15     var4  <= var4 + 8'b1;
16   end
17 end
18 endmodule
```

Assume that the input `in` *always* has the following value:
`in = 8'h09`

What **unsigned decimal** values does the `out` signal get in the following waveform diagram? Fill in the gray boxes with an `out` value for each `clk` cycle. Briefly explain your answer.



Brief explanation (to help us award you partial credit):

Explanation.
The module outputs the `in`th number in the Fibonacci sequence after `in` clock cycles. Until then, it outputs the number of clock cycles that have passed since reset.
For the given value of `in` (8'h09), the values for `out` are from leftmost yellow box to the rightmost yellow box:
`0, 1, 2, 3, 4, 5, 6, 7, 8, 55`

4.2 Is ChatGPT *not* Right? [30 points]

You gave ChatGPT the following prompt to help with your lab report: "A Verilog module that simulates a character's movement on a 2D-plane. The module takes four inputs for four directions (direction inputs) the character can move to. The module outputs x and y coordinates. The character stays in the same coordinate if none of the direction inputs are set. Initial coordinates (set on reset) are 0, 0. Stride determines how many units the character moves in one step."

```
module movement (
  input clk,    input rst,
  input up,     input down,
  input left,   input right,
  (1) stride,
  output [7:0] x_coord,
  output [7:0] y_coord
);
(2) x_internal, y_internal; // 8-bit signals
wire [2:0] move_amount = (3); // if stride is not zero, move by stride amount, else move by 1
always @(posedge clk) begin
  if (rst) begin
    x_internal <= 0; y_internal <= 0;
  end else begin
    if (up) y_internal <= y_internal + move_amount;
    else if (down) y_internal <= y_internal - move_amount;
    else if (left) x_internal <= x_internal - move_amount;
    x_internal <= x_internal + move_amount;
  end
end
(4) x_coord = x_internal; // output coordinate
(4) y_coord = y_internal; // output coordinate
endmodule
```

Provide your choice for each blank (1), (2), and (4) below. Circle only one of A, B, C, D. Provide a one-line expression for (3) (*Hint:* Use the ternary operator (`?`) to implement a MUX).

(1): A. output B. output reg C. input wire [2:0] D. input reg

(2): A. wire [7:0] B. [7:0] wire C. wire [8:0] D. reg [7:0]

(3): `stride != 3'b0 ? stride : 3'b1;`

(4): A. B. `assign` C. == D. `let`

Explanation.
(1): Signal `stride` is used as an input to the module, so it should be declared as an input. Among options that describe input signals (C and D), `input reg` is not valid Verilog syntax.
(2): The correct way to describe signals that we can assign values to in an always block is `reg [7:0]`.
(3): We describe a mux using the ternary operator as such: `stride != 3'b0 ? stride : 3'b1;`. If stride is zero, the left-hand side of the ternary operator (i.e., `stride`) is the output of the mux and otherwise the right-hand side (i.e., `3'b1`) is the output of the mux.
(4): The correct syntax for assigning a value to a signal is `assign x_coord = x_internal;`. Other options are not valid Verilog syntax.

Did ChatGPT inject any errors in this code? Write down line number(s) and a short explanation (to help us award you partial credit).

Explanation. Line 18 introduces a logical error, causing `x_internal` to always be incremented by `move_amount` regardless of the direction of movement.

5 Memory Potpourri [45 points]

Read the following statements about memory organization & technology. Circle "True" if the statement is true and "False" otherwise. *Note: we will subtract 2 points for each* ***incorrect*** *answer and award 0 points for each unanswered question.*

1. [3 points] A main memory access typically has larger latency than a register file access.
   [1. True] 2. False
2. [3 points] SRAM is commonly used as main memory in modern computers.
   1. True [2. False]
3. [3 points] A DRAM cell requires larger power to store data compared to an SRAM cell.
   1. True [2. False]
4. [3 points] Reads are faster than writes in DRAM.
   1. True [2. False]
5. [3 points] Reads are faster than writes in phase change memory.
   [1. True] 2. False
6. [3 points] A bitline in a DRAM array connects all DRAM cells in a DRAM row to the row decoder circuitry.
   1. True [2. False]
7. [3 points] Using virtual memory reduces the memory access latency.
   1. True [2. False]
8. [3 points] Phase Change Memory (PCM) is non-volatile.
   [1. True] 2. False
9. [3 points] If a hypothetical system is *not* constrained by chip area, memory cost ($), and energy consumption, PCM would be the best memory technology to use in that system.
   1. True [2. False]
10. [3 points] A program with a streaming memory access pattern leads to very high temporal locality in the last level data cache.
    1. True [2. False]
11. [3 points] In DRAM, accesses to different rows in one bank can be serviced faster compared to accesses to different rows in different banks.
    1. True [2. False]
12. [3 points] TLB is a specialized instruction cache that caches instructions based on branch prediction results.
    1. True [2. False]
13. [3 points] Virtual memory simplifies software design.
    [1. True] 2. False
14. [3 points] A page fault happens when the TLB does not contain the entry needed by an instruction.
    1. True [2. False]
15. [3 points] A fully-associative L1 TLB that only stores 4KB virtual-to-physical mappings and has 1024 entries can cover up to 4MB of memory.
    [1. True] 2. False

6 Performance Evaluation [50 points]

A multi-cycle processor $P1$ executes *load instructions* in **6 cycles**, *store instructions* in **6 cycles**, *arithmetic instructions* in **2 cycles**, and *branch instructions* in **2 cycles**. Consider an application A where 40% of all instructions are load instructions, 20% of all instructions are store instructions, 30% of all instructions are arithmetic instructions, and 10% of all instructions are branch instructions.

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

$CPI = 0.4 \times 6 + 0.2 \times 6 + 0.3 \times 2 + 0.1 \times 2$
$CPI = 4.4$

(b) [5 points] A new design of the processor doubles the clock frequency of $P1$. However, the latencies of *all* instructions increase by 4 cycles. 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.

$CPI = 0.4 \times 10 + 0.2 \times 10 + 0.3 \times 6 + 0.1 \times 6$
$CPI = 8.4$

(c) [20 points] Which processor is faster ($P1$ or $P2$)? By how much (i.e., what is the speedup)? Show your work.

$P2$ is 1.05× faster than $P1$.

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

Assuming that $Execution\_Time\_P2 < Execution\_Time\_P1 \implies \frac{Execution\_Time\_P1}{Execution\_Time\_P2} > 1$. Thus:
$\implies \frac{instructions \times CPI_{P1} \times clock\_time}{instructions \times CPI_{P2} \times \frac{clock\_time}{2}}$
$\implies \frac{4.4 \times clock\_time}{8.4 \times \frac{clock\_time}{2}}$
$\implies \frac{4.4}{4.2}$
$\implies 1.05$

(d) [20 points] You want to improve the original $P1$ design by including one new optimization *without* changing the clock frequency. You can choose **only one** of the following options:

(1) **ALU**: An optimized *ALU*, which *halves* the latency of both arithmetic and branch instructions.

(2) **LSU**: An *asymmetric load-store unit*, which *halves* the latency of load operations but *doubles* the latency of store operations.

Which optimization do you add to $P1$ for application A? Show your work and justify your choice.

The ALU optimization.

Explanation.
Application A executes 40% load, 20% store, 30% arithmetic, and 10% branch instructions.
By Amdahl's Law, we have:

$$Speedup_{ALU} = \frac{1}{(1-0.3-0.1)+\frac{0.3+0.1}{2}} = 1.25$$
$$Speedup_{LSU} = \frac{1}{(1-0.4-0.2)+\frac{0.4}{2}+0.2\times 2} = 1.0$$

The ALU optimization provides 1.25× speedup, while the LSU provides no speedup at all.

Alternative Solution.
With the ALU, the new CPI of processor $P1$ will be:
$CPI_{ALU} = 0.4 \times 6 + 0.2 \times 6 + 0.3 \times \frac{2}{2} + 0.1 \times \frac{2}{2}$
$CPI_{ALU} = 4.0$

With the LSU, the new CPI of processor $P1$ will be:
$CPI_{LSU} = 0.4 \times \frac{6}{2} + 0.2 \times (6 \times 2) + 0.3 \times 2 + 0.1 \times 2$
$CPI_{LSU} = 4.4$

Since $CPI_{ALU} < CPI_{LSU}$, integrating the ALU will improve the overall cycles-per-instruction.

7 Pipelining [70 points]

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

```
1     MOVI R1, X          # R1 <- X
2     MOVI R2, Y          # R2 <- Y
3  L1:
4     MUL  R4, R1, R1     # R4 <- R1 × R1
5     MUL  R1, R1, R2     # R1 <- R1 × R2
6     ADD  R4, R5, R6     # R4 <- R5 + R6
7     ADD  R5, R2, R4     # R5 <- R2 + R4
8     SUBI R3, R1, 2048   # R3 <- R1 - 2048, set condition flags
9     JNZ  L1             # Jump to L1 if zero flag is NOT set
10    MUL  R1, R1, R2     # R1 <- R1 × R2
```

Code Listing 1: Assembly Program

		Cycles															
	Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	MOVI R1, X	F	D	E1	E2	E3	M	W									
2	MOVI R2, Y		F	D	E1	E2	E3	M	W								
3	MUL R4, R1, R1			F	D	-	E1	E2	E3	M	W						
4	MUL R1, R1, R2				F	-	D	E1	E2	E3	M	W					
5	ADD R4, R5, R6						F	D	E1	E2	E3	M	W				
6	ADD R5, R2, R4							F	D	-	-	E1	E2	E3	M	W	
7	SUBI R3, R1, 2048								F	-	-	D	E1	E2	E3	M	W
8	JNZ L1											F	D	-	-	E1	...
9	...	...															

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. If the provided information is not sufficient to answer a question, answer "Unknown" and explain your reasoning clearly.

(a) [15 points] List the data forwarding paths between pipeline stages.

The result of E3 stage is forwarded to E1 stage (e.g., R1's value at clock cycle 6 and R4's value at clock cycle 11). The result of M stage is forwarded to E1 stage (e.g., R1'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 15).
There is no other information for any other data forwarding. Therefore, other data forwardings are unknown.

(b) [10 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.

For the rest of this question, assume the following:

- X = 4, Y = 2 in Code Listing 1.
- Branch predictor always predicts correctly.
- The machine uses hardware interlocking.

At a given clock cycle T,

- the value stored in R1 is 1024.
- the processor fetches the dynamic instruction N which is `MUL R4, R1, R1`

(c) [15 points] Calculate the value of T (the clock cycle of the given snapshot). Show your work.

T = 82.

Explanation.
The instruction `MUL R4, R1, R1` is fetched for the first time at the clock cycle 3. After the first iteration of the loop, the instruction is fetched for the second time at the clock cycle 12.

The instruction `JNZ L1` stalls at the Decode stage and delays `MUL R4, R1, R1`. Due to this delay, there are 10 cycles in between the Nth and (N+1)th times the instruction is fetched, after the first iteration of the loop.

If R1 = 1024, this instruction is fetched and executed 8 times so far.

Since in cycle T the first instruction in the loop (`MUL R4, R1, R1`) is being fetched, no cycles of the 9th iteration have executed so far.

Then, $T = 12 + 7 \times 10 = 82$

(d) [15 points] Calculate the value of N (the total number of dynamic instructions fetched by the clock cycle T). Show your work.

N = 51.

Explanation.
Loop iterates for 8 times before the processor reaches to clock cycle T.

There are two instructions before the loop starts.

Then, $N = 2 + 8 \times 6 + 1 = 51$ (assuming that the instruction indices start from 1).

(e) [15 points] Calculate the total execution time of the assembly code in Code Listing 1 until the completion in terms of the number of clock cycles. Show your work.

100 cycles.

Explanation.
Until the end of the second iteration, the loop takes 19 cycles as shown above.
The steady-state throughput of an iteration after the first iteration is 6 instructions in 10 cycles.

Loop will iterate until R1 becomes 2048, which means 9 iterations in total.

There is only one instruction after the loop, which takes 1 cycle to complete.

Then, $T = 19 + 8 \times 10 + 1 = 100$

8 Vector Processing [80 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)
LDM	V_i	1	$V_{MSK} \leftarrow LSB(V_i)$ (V_{MSK} = Vector Mask Register)
VLD	V_i, #A	50 row hit, 100 row miss, pipelined	$V_i \leftarrow Mem[Address]$
VST	V_i, #A	50 row hit, 100 row miss, 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	$V_i \leftarrow V_j + V_k$
VSHFR	V_i, V_j	10, pipelined	$V_i \leftarrow V_j >> 1$
VNOT	V_i	4, pipelined	$V_i \leftarrow BitwiseNOT(V_i)$
VCMPZ	V_i, V_j	4, pipelined	if($V_j == 0$) $V_i \leftarrow$ 0xFFFF; else $V_i \leftarrow$ 0x0000

Assume the following:

- The processor has an in-order pipeline and issues one instruction per cycle.
- There are 8 vector registers (V_0,V_1,V_2,V_3,V_4,V_5,V_6,V_7), and 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.
- LDM moves the least-significant bit (LSB) of each vector element in a vector register V_i into the corresponding position in V_{MSK}. This instruction is executed in one single cycle.
- The main memory is composed of N banks, and each bank has a row buffer of size 64 bits.
- All rows in main memory are initially closed (i.e., all banks are precharged).
- The memory is byte addressable, and the address space is represented using 32 bits.
- Vector elements are stored in memory in a 4-byte-aligned manner. The first element of a vector always starts at the beginning of a memory row.
- 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 $A + 4$ 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 latency of accessing memory is 100 cycles when the memory request misses in the row buffer, and 50 cycles when the memory request hits in the 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 is one functional unit for executing VLD instructions and a separate functional unit for executing VST instructions. This means the load and store operations for *different vectors* cannot be overlapped.
- The operations on a vector do not affect the vector elements corresponding to the locations in the Vector Mask Register (V_{MSK}) that are set to 0.

(a) [20 points] What should the minimum number of banks (N) be to avoid stalls while executing a VLD or VST instruction? Calculate the minimum number of banks for every stride from 1 to 10. Explain.

101 banks for even strides, 100 banks for odd strides

Explanation.

To calculate the minimum value, we have to assume the worst case, which is when all memory accesses are row buffer misses (latency = 100). To avoid stalls, we need to ensure that consecutive vector elements access 100 different banks.

We illustrate the solution for even strides (2 and 4) and odd strides (1 and 3).

101 banks are enough to avoid stalls with even numbers. For example, with a vector stride of 2, consecutive elements of a vector will map to banks 0, 2, 4 ... 96, 98, 100, 1, 3 ... 97, 99. With a vector stride of 4, consecutive elements of a vector will map to banks 0, 4, 8 ... 96, 100, 3, 7 ... 95, 99, 2, etc.

100 banks are enough to avoid stalls with odd numbers. For example, with a vector stride of 1, consecutive elements of a vector will map to banks 0, 1, 2, 3 ... 98, 99. With a vector of stride 3, consecutive elements of a vector will map to banks 0, 3, 6 ... 96, 99, 2, 5 ... 95, 98

So, the minimum number of banks is 100 for odd strides, and 101 for even strides.

(b) [30 points] Translate the following loop into assembly code that can be executed in the least possible number of cycles in the previously described vector machine:

```
for i= 0 to 45:
    if(a[i] == 0):
        c[i] = b[i]
    else:
        c[i] = a[i] * b[i] + a[i]/2
```

Assume:

- The same machine as in part (a).
- In the for loop, 45 is inclusive, i.e., [0, 45]
- The size of the elements of vectors a, b, and c is 4 bytes
- Vectors a, b, and c do not share parts of the same DRAM row

```
SET V_st, 1        # Load Vector Stride Register
SET V_ln, 46       # Load Vector Length Register
VLD V_1, a         # Read from array a
VLD V_2, b         # Read from array b
VCMPZ V_3, V_1     # Compare V1 to 0
LDM V3             # Load Vector Mask Register
VST c, V2          # Write to array c
VNOT V3            # BitwiseNOT
LDM V3             # Load Vector Mask Register
VSHFR V_4, V_1     # Shift to divide
VMUL V_5, V_1, V_2 # Multiply
VADD V_6, V_5, V_4 # Add
VST c, V_6         # Write to array c
```

(c) [30 points] What is the number of cycles the previous code takes to execute in the vector processor described in this question? Assume:

- Vectors a and b are in different rows
- A machine that has a memory with 8 banks.
- The rest of the machine is the same as in part (a).

1822 cycles
Explanation.

The memory accesses look like:

```
bank0 --MISS-|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank1  --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank2   --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank3    --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank4     --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank5      --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|--HIT--|
bank6       --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|
bank7        --MISS--|--HIT--|--MISS--|--HIT--|--MISS--|
```

Therefore, the latency of the load corresponds to the latency of the bank with the larger latency. In this case, bank 5 (300+150+5 = 455 cycles). The latency of a store is also 455 cycles.

The general picture is:

```
SET: |-S-|
SET:     |-S-|
VLD:         |-----VLD-----|
VLD:                         |-----VLD-----|
VCMPZ:                        |VCMPZ|
LDM:                                |L|
VST:                                        |-----VST-----|
VNOT:                                        |VNOT|
LDM:                                              |L|
VSHFR:                                              |VSHFR|
VMUL:                                                |VMUL|
VADD:                                                      |VADD|
VST:                                                             |-----VST-----|
```

S = 1
VLD_cycles = VST_cycles = 455
VMUL_cycles = 10 + 45 = 55
VCMPZ_cycles = 4 + 45 = 49
VNOT_cycles = 4 + 45 = 49
L = 1
VSHFR = 10 + 45 = 55
VADD = 5 + 45 = 50

Considering how the latency of some instructions is hidden by the other instructions, the total cycles can be calculated as:
$total\_cycles = S + S + VLD\_cycles + VLD\_cycles + VST\_cycles + VST\_cycles =$ 1822 *cycles*

9 VLIW [60 points]

You are the human compiler for a VLIW processor whose specifications are as follows:

- There are a total of 7 functional units: 3 load units, 1 store unit, 1 addition unit, 1 multiplication unit, and 1 branch unit.
- The VLIW processor can **only** execute assembly operations listed in Table 1. The table shows the instructions that each functional unit can execute and each instructon's semantics. Note that the `load_inc`/`store_inc` instructions automatically increment the address source register r_{src2} by 1, *after* data is loaded/stored.
- All assembly operations have a 1-cycle latency (including `load`, `load_inc`, `store`, and `store_inc`).
- This machine has 32 registers (`r0`, `r1`, ..., `r31`).
- The registers are read at the rising edge and written at the falling edge of the clock.
- The memory is word-addressable (1 word = 4 bytes).
- The VLIW processor operates at 1 GHz.

Functional Unit Type	**Operation (in assembly notation)**	**Semantics**
load	`load` r_{dst}, [r_{src1}, r_{src2}, #offset]	r_{dst} := MEM[r_{src1} + r_{src2} + #offset]
	`load_inc` r_{dst}, [r_{src1}, r_{src2}, #offset]	r_{dst} := MEM[r_{src1} + r_{src2} + #offset] r_{src2} := r_{src2} + 1
store	`store` [r_{src1}, r_{src2}, #offset], r_{src3}	MEM[r_{src1} + r_{src2} + #offset] := r_{src3}
	`store_inc` [r_{src1}, r_{src2}, #offset], r_{src3}	MEM[r_{src1} + r_{src2} + #offset] := r_{src3} r_{src2} := r_{src2} + 1
addition	`add` r_{dst}, r_{src1}, r_{src2}	r_{dst} := r_{src1} + r_{src2}
multiplication	`mult` r_{dst}, r_{src1}, r_{src2}	r_{dst} := r_{src1} × r_{src2}
branch	`bne` r_{src1}, #offset, TARGET	branch to TARGET if r_{src1} is not equal to #offset
(any of the above)	`NOP`	Functional unit is idle for one cycle

Table 1: Assembly operations of the target VLIW processor. `#offset` indicates an immediate value.

Figure 1 shows the C code and its equivalent assembly code for the application that we will execute in this VLIW processor. Assume that N **is an even positive integer** throughout this question.

In the assembly code, registers `r29`, `r30`, and `r31` hold the base addresses of the C-code arrays `A`, `B`, and `C`, respectively. Register `r0` is initialized with 0 and register `r1` is initialized with 1.

C code

```
// An integer is 4 bytes long
int A[N+1];
int B[N+1];
int C[N+1];
... // code to initialize A and B
for (int i = 1; i <= N; i++)
    C[i] = C[i-1] * A[i] + B[i];
```

Assembly code

```
LOOP:
    (v1) load_inc   r2, [r31, r0, #0]  // r2 := [r31 + r0 + #0]; r0 := r0 + 1
    (v2) load       r3, [r29, r1, #0]  // r3 := [r29 + r1 + #0]
    (v3) load       r4, [r30, r1, #0]  // r4 := [r30 + r1 + #0]
    (v4) mult       r5, r2, r3         // r5 := r2 * r3
    (v5) add        r6, r5, r4         // r6 := r5 + r4
    (v6) store_inc  [r31, r1, #0], r6  // [r31 + r1 + #0] := r6; r1 := r1 + 1
    (v7) bne        r1, #N, LOOP       // branch to LOOP if r1 not equal to #N
```

Figure 1: C and assembly codes. ($v1$) .. ($v7$) are instruction labels.

(a) [30 points] Your goal in this question is to statically schedule the instructions in Figure 1 to the VLIW processor specified above. Table 2 (on the next page) represents the occupancy of each functional unit during the execution of the assembly code in Figure 1.

For the assembly code given in Figure 1, **fill in Table 2** with the appropriate VLIW instructions.

In your solution, **minimize the number of VLIW instructions**, and ensure that each instruction is scheduled to execute **as soon as possible**. Table 2 should only contain assembly operations supported by the VLIW processor, as described in Table 1.

	Functional Unit						
VLIW Instruction	**Load**	**Load**	**Load**	**Store**	**Mult**	**Add**	**Branch**
1 LOOP:	load_inc r2, [r31, r0, #0]	load r3, [r29, r1, #0]	load r4, [r30, r1, #0]	NOP	NOP	NOP	NOP
2	NOP	NOP	NOP	NOP	mult r5, r2, r3	NOP	NOP
3	NOP	NOP	NOP	NOP	NOP	add r6, r5, r4	NOP
4	NOP	NOP	NOP	store_inc [r31, r1, #0], r6	NOP	NOP	NOP
5	NOP	NOP	NOP	NOP	NOP	NOP	bne r1, #N, LOOP
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							

Table 2

(b) [15 points] What is the ratio between the number of useful operations and the number of VLIW instructions in your code? A useful operation refers to any assembly operation that is *not* a `NOP`.

$\frac{7}{5}$ useful operations per VLIW instruction.

Explanation.
There are a total of 7 assembly operations (excluding `NOP`s) composing 5 VLIW instructions.

(c) [15 points] What is the execution time (in cycles) of the VLIW processor when executing the sequence of instructions in Table 2, as a function of the loop counter N? Show your work.

Execution time $= 5 \times N$.

Explanation.
A single iteration of the loop takes 5 clock cycles to execute. Since the loop repeats N times, the total execution time is equal to $5 \times N$.

10 Cache [50 points]

Consider a processor using a 4-block LRU-based L1 data cache with a block size of 1 byte. Starting with an empty cache, an application accesses three cache blocks with the following addresses in the order given below:

$$0 \rightarrow 2 \rightarrow 4$$

A malicious programmer tries to reverse-engineer the number of sets and ways in the L1 data cache by issuing *only* two more accesses and observing the cache hit rate across these two accesses. Assume that the programmer can insert the malicious accesses only after the above three accesses of the application.

(a) [20 points] What are the addresses of the next two cache blocks that should be accessed to successfully reverse-engineer the number of sets and ways in the cache? There may be multiple solutions; ***please give the lowest possible addresses that can enable the identification of the number of sets and ways***. Please explain every step in detail to get full points.

0 → 2

Explanation. There are two possible answers:

- **[0 → 2]**
- **[0 → 4]**

There are three possible set/way configurations, shown below labeled by their respective sets/ways. Each configuration shows a drawing of the cache state after the three initial accesses. Rows and columns represent sets and ways, respectively, and the LRU address is shown for each occupied set:

(a) **(4 sets, 1 way)**

4
-
2
-

(b) **(2 sets, 2 ways)**

4	2
-	-

(c) **(1 set, 4 ways)**

0	2	4	-

At this point, all three configurations have a 100% miss rate since they started cold. In order to differentiate between the three configurations with *just two* more accesses, we need to induce *different* hit/miss counts in each of them. The only way this is possible is if one configuration experiences two hits, another two misses, and the last one hit and one miss.

Only two solutions exist to produce this case:

- **[0 → 2]**
  (a) 0 miss, 2 hit = 50% miss rate
  (b) 0 miss, 2 miss = 100% miss rate
  (c) 0 hit, 2 hit = 0% miss rate
- **[0 → 4]**
  (a) 0 miss, 4 miss = 100% miss rate
  (b) 0 miss, 4 hit = 50% miss rate
  (c) 0 hit, 4 hit = 0% miss rate

Choosing the lowest possible addresses, the correct solution is **0 → 2**

(b) [15 points] What is the number of sets and ways if the cache hit rate observed over the two extra addresses accessed in Part (1) were:

L1 hit rate	# sets	# ways
100%		
50%		
0%		

Explain your reasoning:

Based on the solution to Part (1), these are the number of sets and ways corresponding to different hit rates.

Solution:

L1 hit rate	# sets	# ways
100%	1	4
50%	4	1
0%	2	2

(c) [15 points] Is it possible to reverse-engineer the number of sets and ways of the cache using two accesses (after the application's first three accesses) if the Most Recently Used (MRU) block is replaced first? Explain your reasoning.

No. There is no solution for just two more accesses because with an MRU policy, no permutation of two more accesses is able to assign a unique L1 hit rate to each of the three cache configurations.

11 BONUS: Systolic Arrays [50 points]

You are given a systolic array of 2×2 Processing Elements (PEs), interconnected as shown in Figure 2. The inputs of the systolic array are labeled as H_0, H_1 and V_0, V_1. Figure 3 shows the PE logic, which performs a multiply-accumulate (MAC) operation and saves the result to an internal register (reg). Figure 3 also shows how each PE propagates its inputs. We make the following assumptions:

- The latency of each MAC operation is one cycle, i.e., if the inputs to a PE are available in cycle c, the updated register value will be available in cycle $c+1$.
- The propagation of the values from i_0 to o_0, and from i_1 to o_1, takes one cycle.
- The initial values of all internal registers is zero.

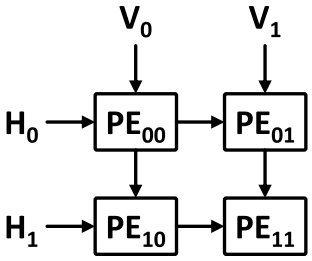


Figure 2: PE array

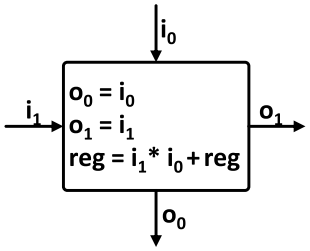


Figure 3: Processing Element (PE)

Your goal is to use the systolic array shown in Figure 2 to perform the multiplication $C = A \times B$, where A, B, and C are 2×2 matrices. Recall that the multiplication of two $K \times K$ matrices is defined as follows:

$$C_{ij} = \sum_{k=0}^{K-1} A_{ik} \times B_{kj}$$

As an example, for $K = 2$, the calculation for C_{00} is as follows:

$$C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10}$$

Compute the multiplication in the minimum possible number of cycles. Fill the following table with:

1. Each input element (from matrices $A_{2\times2}$ and $B_{2\times2}$) in the correct cycle and input port of the systolic array (H_0, H_1 and V_0, V_1).
2. Each output element (for matrix $C_{2\times2}$) in the cycle and PE that generates each output.

(a) [25 points] Fill in the blanks only with relevant information. Input cells left blank are interpreted as 0.

cycle	H0	H1	V0	V1	PE_{00}	PE_{01}	PE_{10}	PE_{11}
0	A_{00}		B_{00}					
1	A_{01}	A_{10}	B_{10}	B_{01}				
2		A_{11}		B_{11}	C_{00}			
3						C_{01}	C_{10}	
4								C_{11}
5								
6								
7								

(b) [25 points] Suppose that the same systolic array from Figure 2 is used to compute the multiplication of two 4×4 matrices. How many cycles does it take to perform the multiplication? Assume that the register in a PE resets to 0 immediately after an output is generated, i.e., PEs can start accumulating for the next output element in the next cycle without waiting for an extra cycle to reset the register to 0. Show your work.

19 cycles.

Each PE needs to calculate four elements to calculate the $4 \times 4 = 16$ output elements.

For the first element calculated by each PE, the timeline looks similar to (a), but requires two additional cycles for the four MAC operations instead of two per element, i.e., seven cycles in total until PE_{11} produces its output.

The remaining three elements calculated by each PE require four cycles each if pipelined with the previously calculated element.

Thus, the total number of cycles is $7 + 3 \times 4 = 19$.

cycle	**H0**	**H1**	**V0**	**V1**	**PE_{00}**	**PE_{01}**	**PE_{10}**	**PE_{11}**
0	A_{00}		B_{00}					
1	A_{01}	A_{10}	B_{10}	B_{01}				
2	A_{02}	A_{11}	B_{20}	B_{11}				
3	A_{03}	A_{12}	B_{30}	B_{21}				
4	A_{00}	A_{13}	B_{02}	B_{31}	C_{00}			
5	A_{01}	A_{10}	B_{12}	B_{03}		C_{01}	C_{10}	
6	A_{02}	A_{11}	B_{22}	B_{13}				C_{11}
7	A_{03}	A_{12}	B_{32}	B_{23}				
8	A_{20}	A_{13}	B_{00}	B_{33}	C_{02}			
9	A_{21}	A_{30}	B_{10}	B_{01}		C_{03}	C_{12}	
10	A_{22}	A_{31}	B_{20}	B_{11}				C_{13}
11	A_{23}	A_{32}	B_{30}	B_{21}				
12	A_{20}	A_{33}	B_{02}	B_{31}	C_{20}			
13	A_{21}	A_{30}	B_{12}	B_{03}		C_{21}	C_{30}	
14	A_{22}	A_{31}	B_{22}	B_{13}				C_{31}
15	A_{23}	A_{32}	B_{32}	B_{23}				
16		A_{33}		B_{33}	C_{22}			
17						C_{23}	C_{32}	
18								C_{33}

12 BONUS: Prefetching [50 points]

An ETH student writes two programs (A and B) and runs them on two different toy machines (M1 and M2) to determine the type of the prefetcher used in each of these machines. She observes programs A and B to generate the following memory access patterns (note that these are *cacheblock addresses*, not byte addresses).

Program A: 27 memory accesses

```
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64,
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64
```

Program B: 501 memory accesses

```
b, b + 2, b + 4, ...., b + 998, b + 1000
```

The student measures the coverage (i.e., the fraction of program's memory accesses correctly predicted by the prefetcher) and accuracy (i.e., the fraction of sent prefetch requests that are used by the program) of the prefetching mechanism in each of the machines. The following table shows her measurement results:

	Machine M1		**Machine M2**	
	Coverage	Accuracy	Coverage	Accuracy
Program A	6/27	6/27	1/3	9/26
Program B	499/501	499/501	499/501	499/500

The student knows the following information about the machines:

- There are three possible choices for the prefetching mechanism:
  1. Stride prefetcher
  2. 1st-next-block prefetcher with degree 1: Prefetches cacheline $A+1$ after seeing access to block A
  3. 4th-next-block prefetcher with degree 1: Prefetches cacheline $A+4$ after seeing access to block A
- Each prefetcher has large enough resources to detect and store access patterns.
- Each prefetcher starts with an empty table.
- Each prefetcher sends only one prefetch request for each program access.
- Each memory access is separated long enough in time so that all prefetch requests sent can complete before the next access occurs.
- No prefetcher employs any confidence mechanism (e.g., the stride prefetcher will send a prefetch request to address $A+4$ by only seeing two consecutive memory accesses to addresses A and $A+2$).

Determine what type of prefetching mechanism is used by M1 and M2. Show your work. Answers without explanation will not be rewarded.

Machine M1: 4th-next-line prefetcher

Machine M2: Stride prefetcher

Space for explanation:

M1: 4th-next-line prefetcher
M2: Stride prefetcher

Explanation

We calculate the accuracy and coverage for all three types of prefetchers, and then we can answer what prefetcher each machine is using. Underlined and red-marked cacheline addresses are correctly and incorrectly prefetched, respectively.

Each prefetcher works in the following way while running Application A:

Stride: Coverage: 1/3, Accuracy: 9/26
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 12, a + 24, a + 48, a + 96)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a - 64, a + 5, a + 12, a + 24, a + 48, a + 96)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64 (incorrect: a - 64, a + 5, a + 12, a + 24, a + 48, a + 96)

1st-next-line: Coverage: 4/9, Accuracy: 4/9
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 9, a + 17, a + 33, a + 65)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 9, a + 17, a + 33, a + 65)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 9, a + 17, a + 33, a + 65)

4th-next-line: Coverage: 6/27, Accuracy: 6/27
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64, (incorrect: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68)
a, a + 1, a + 2, a + 3, a + 4, a + 8, a + 16, a + 32, a + 64 (incorrect: a + 5, a + 6, a + 7, a + 12, a + 20, a + 36, a + 68)

The three prefetechers work in the following way while running Application B:

Stride: Coverage: 499/501, Accuracy: 499/500
b, b + 2, b + 4, b + 6, b + 8, b + 10, ..., b + 998, b + 1000 (incorrect: b + 1002)

1st-next-line: Coverage: 0, Accuracy: 0
b, b + 2, b + 4, b + 6, b + 8, b + 10, ... , b + 998, b + 1000 (incorrect: b + 1, b + 3, ..., b + 999, b + 1001)

4th-next-line: Coverage: 499/501, Accuracy: 499/501
b, b + 2, b + 4, b + 6, b + 8, b + 10, ..., b + 998, b + 1000 (incorrect: b +1002, b + 1004)