Family Name: SOLUTIONS First Name: Student ID:

Final Exam

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

ETH Zürich, Spring 2021

Prof. Onur Mutlu

Problem 1 (20 Points):	Boolean Logic Circuits	
Problem 2 (60 Points):	Verilog	
Problem 3 (45 Points):	Finite State Machines	
Problem 4 (30 Points):	ISA vs. Microarchitecture	
Problem 5 (45 Points):	Performance Evaluation	
Problem 6 (65 Points):	Pipelining	
Problem 7 (60 Points):	Tomasulo's Algorithm	
Problem 8 (75 Points):	GPUs and SIMD	
Problem 9 (45 Points):	Branch Prediction	
Problem 10 (70 Points):	Caches	
Problem 11 (BONUS: 25 Points):	Prefetching	
Problem 12 (BONUS: 35 Points):	Systolic Arrays	
Total (575 (515 + 60 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 [20 points]

(a) [10 points] Using Boolean algebra, find the simplest Boolean algebra equation for the following min-terms:
$\sum(1111, 1110, 1000, 1001, 1011, 1010, 0000)$. Show your work step-by-step.

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

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

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

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

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

(b) [10 points] Convert the following Boolean equation so that it only contains NOR operations. Show your work step-by-step.

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

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

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

2 Verilog [60 points]

2.1 Complete the Verilog code [30 points]

For each numbered blank (1)-(5) in the following Verilog code, **mark the choice below** (i.e., one of options A, B, C, D) that makes the Verilog module operate as described in the comments. The resulting code must have correct syntax.

```
module my_module (input clk, input rst,
  input[15:0] idata, input[1:0] op, (1)[31:0] odata);

  (2) nval = 32'd0; // defining a 32-bit signal with an initial value of 0

  always@* begin
    case (op)
      2'b00:
        nval = odata + idata; // when 'op' is decimal 0, add 'idata' to
                              // 'odata' and assign the result to 'nval'
      2'b01:
        nval = odata - idata; // when 'op' is decimal 1, subtract 'idata'
                              // from 'odata' and assign the result to 'nval'
      2'b10:
        nval = idata; // when 'op' is decimal 2, assign 'idata' to 'nval'
      (3):
        nval = 0; // when 'op' is decimal 3, assign 0 to 'nval'
    endcase
  end

  // executing the following always block on the rising edge of 'clk'
  always@ (posedge clk) begin
    if (rst)
      (4) // resetting 'odata' to 0 for the next cycle
    else
      (5) // assigning 'nval' to 'odata' for the next cycle
  end
endmodule
```

Provide your choice for each blank (1)-(5) below:

(1):	A. output	B. output reg	C. output wire	D. input reg
(2):	A. reg[31:0]	B. input[31:0]	C. wire[31:0]	D. int[31:0]
(3):	A. 2'b3	B. 3'b3	C. 2'h11	D. default
(4):	A. assign odata <= 0;	B. assign odata = 0;	C. odata == 0;	D. odata <= 0;
(5):	A. assign odata <= nval;	B. assign odata = nval;	C. odata == nval;	D. odata <= nval;

Explanation.
(1): `odata` must be declared as an `output` signal since values are assigned to it in the second `always` block. It cannot be an `input` signal since inputs are read-only signals and no assignments are allowed to them. `odata` must be also declared as `reg` since the assignments are made inside an always block.
(2): `nval` must be declared as `reg[31:0]` since values are assigned to it inside the first `always` block.
(3): `default` is a correct choice since all other cases for a 2-bit values (i.e., `2'b00`, `2'b01`, and `2'b10`) are defined in the `case` statement. The other choices are not correct since they do not properly specify the value of 3. For example, in `2'b3`, the problem is that `3` is not a valid binary digit but `2'b` must be followed by a 2-bit binary value.
(4): Choices with `assign` are not valid since the `assign` keyword cannot be used in an `always` block. Choice C does not specify an assignment operator but an equality comparison, hence it is not a valid choice either. The correct choice is D, which assigns 0 to `odata` using non-blocking assignment operator.
(5): The correct choice is D due to the same reasons as in (4).

2.2 What Does This Code Do? [30 points]

You are given a Verilog code that you are asked to analyze and find out what it does.

```
module my_module2 (input clk, output[1:0] out);

  reg state = 1'b0;
  reg[1:0] my_reg = 0;

  always@(posedge clk) begin
    state <= &out ? ~state : state;
  end

  always@(posedge clk) begin
    case(state)
      1'b0: begin
        my_reg <= my_reg + 1;
      end
      1'b1: begin
      my_reg <= my_reg - 1;
      end
    endcase
  end

  assign out = my_reg;
endmodule
```

Show the values (as unsigned decimal numbers) that the `out` signal takes, starting from the initial state of the module, for 16 consecutive clock (i.e., `clk`) cycles. Explain your answer briefly.

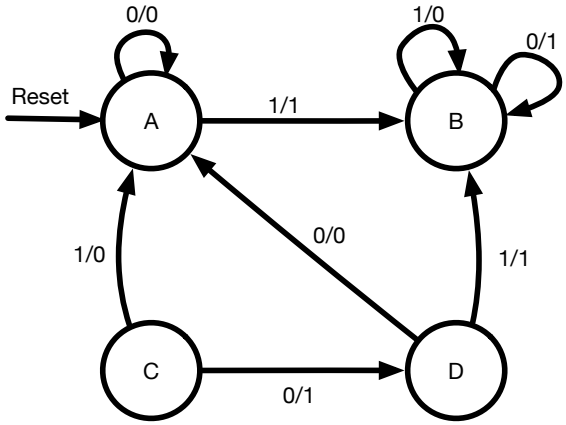
`out` is equal to $0, 1, 2, 3, 0, 3, 2, 3, 0, 3, 2, 3, 0, 3, 2, 3$ in the first 16 clock cycles.

Explanation.
The module either increments or decrements `my_reg` depending on the `state`. When `state` is equal to 0, `my_reg` is incremented by 1 and otherwise decremented by 1. The value of `my_reg` is directly assigned to the `out` signal, and both signals are 2-bit wide.
`my_reg` and `state` are both initially 0. Therefore, in subsequent cycles, `my_reg` gets incremented until it reaches 3. During the next cycle, a new value for `state` is being computed (i.e., the inverse of `state` as $\sim$`state`). However, since the new value of the `state` is not updated until the next positive edge of the `clk`, the second always block reads `state` as 0, and thus `my_reg` gets incremented again to become 0 (the maximum value a 2-bit register can represent is 3 and incrementing `my_reg` one more time makes it 0).
During the next cycle, `state` is 1 and `my_reg` is decremented back to 3. Since `my_reg` (and thus `out`) being 3 inverts `state`, `state` becomes 0 in the subsequent cycle and `my_reg` becomes 2 during the positive edge of `clk` when `state` is inverted. Then, `my_reg` gets incremented to 3 and 0 in the next consecutive cycles. Because `state` remains as 0 or 1 for two consecutive cycles and then gets inverted, the values of `my_reg` forever repeat the sequence of $(0, 3, 2, 3, 0, 3, 2, 3, \ldots)$.

3 Finite State Machines [45 points]

3.1 Simplifying an FSM [20 points]

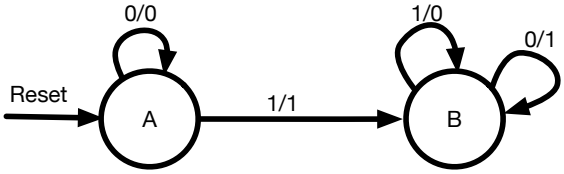
You are given the Mealy state machine of a *one input / one output* digital circuit design. Answer the following questions for the given state diagram.



(a) [10 points] 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.

- There is no way the state goes to C, so it is a non-used state.
- After deleting C, there is no way the state goes to D, so D is also a useless state.
- We can simplify the state diagram as shown next:



(b) [10 points] Assume this state machine is used to process binary numbers from the least significant bit to the most significant bit. You are given an input bit stream: "10110100". Please show the output bit stream produced by this FSM.

"01001100"
When processing a bit stream from the least significant bit to the most significant bit, this state machine keeps the bits unchanged until the first "1" comes. Then, all the bits are flipped (not include the first "1") after the first "1" comes. Therefore, the output is "01001100".

3.2 Designing an FSM [25 points]

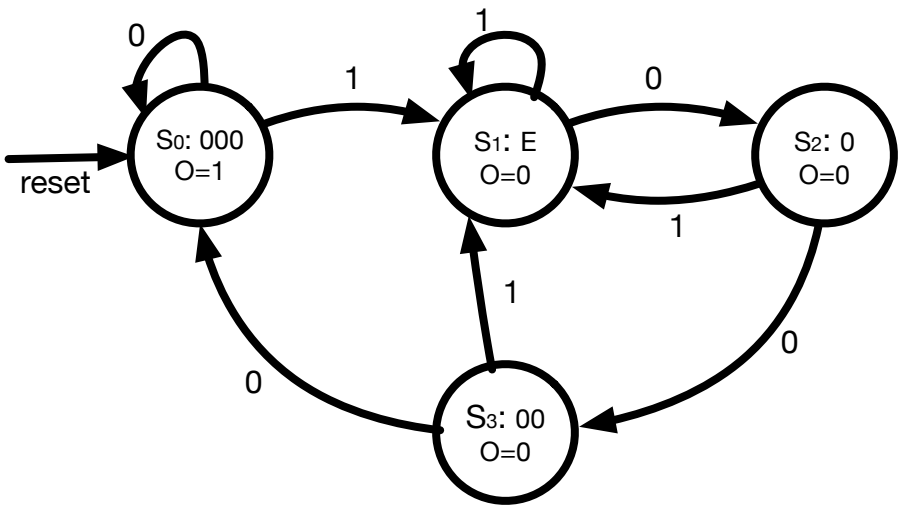
Design a Moore finite state machine (FSM) with one input and one output. The input provides an unsigned binary number in a bit-serial fashion from the most-significant bit to the least-significant bit. The output should be logic-1 in a clock cycle if the provided input so far is divisible by 8 (i.e., [the input number] $\bmod 8 = 0$). (Hint: Recall that the output depends only on the current state in a Moore FSM.)

Below are some example bit-streams that should output a logic-1 value.

- 1000
- 10000
- 11000
- 111000
- 101000

To start an input bit stream, the user should reset the FSM. 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.

From the given examples, we can see that strings can be exactly divided by 8 are all ended with "000" (i.e., three "0"s). Then, we define S_0, a state where the number is ended with "000".
If "1" comes, then the number cannot be exactly divided by 8 and it lacks three "0"s at the end. We define this state as "E" state (S_1), which means no zero at the end.
When there is a "0", then the number lacks two "0"s to be exactly divided by 8. Therefore, we define the state as "0" (S_2).
When there are two "0"s, then the number lacks one more "0" to be exactly divided by 8. Therefore, we define the state as "00" (S_3) .
Based on the analysis above, we can draw the finite state machine whose output (i.e., O) is "1" at S_0 (is "0" at other states):



4 ISA vs. Microarchitecture [30 points]

A new CPU has two comprehensive user manuals available for purchase which describe the ISA and the microarchitecture of the CPU, respectively.

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 ISA manual since it is much cheaper than the microarchitecture manual.

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 (the minimum number of total points you can get for this question is 0).*

1. [2 points] Number of uniquely identifiable memory locations.

   [1. ISA] 2. Microarchitecture

2. [2 points] Number of instructions fetched per clock cycle.

   1. ISA [2. Microarchitecture]

3. [2 points] Support for branch prediction hints conveyed by the compiler.

   [1. ISA] 2. Microarchitecture

4. [2 points] Number of general-purpose registers.

   [1. ISA] 2. Microarchitecture

5. [2 points] Number of non-programmable registers.

   1. ISA [2. Microarchitecture]

6. [2 points] SIMD processing support.

   [1. ISA] 2. Microarchitecture

7. [2 points] Number of integer arithmetic and logic units (ALUs).

   1. ISA [2. Microarchitecture]

8. [2 points] Number of read ports in the physical register file.

   1. ISA [2. Microarchitecture]

9. [2 points] Endianness (big endian vs. small endian).

   [1. ISA] 2. Microarchitecture

10. [2 points] Size of a virtual memory page.

    [1. ISA] 2. Microarchitecture

11. [2 points] Cache coherence protocol.

    1. ISA [2. Microarchitecture]

12. [2 points] Number of cache blocks in the L3 cache.

    1. ISA [2. Microarchitecture]

13. [2 points] Ability to flush (i.e., invalidate) a cache line using the operating system code.

    [1. ISA] 2. Microarchitecture

14. [2 points] Number of pipeline stages.

    1. ISA [2. Microarchitecture]

15. [2 points] How many prefetches the hardware prefetcher generates in a clock cycle.

    1. ISA [2. Microarchitecture]

5 Performance Evaluation [45 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) [10 points] What is the CPI 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) [10 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) [5 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-instructions.

6 Pipelining [65 points]

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

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

Machines A and B have the following specifications:

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

Consider the following code segment:

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

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

(a) [15 points] Re-write the code segment above *with minimal changes* so that it gets correctly executed in Machine A *with minimal latency*. You can either insert `nop` instructions or reorder instructions as needed.

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

(b) [15 points] Fill the table below with the timeline of the first loop iteration of the code segment in Machine A.

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

(c) [10 points] Calculate the number of cycles it takes to execute the code segment on Machine A. Show your work in the box.

Total number of cycles: 1303.

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

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

(d) [15 points] Fill the table below with the timeline of the first loop iteration of the code segment in Machine B.

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

(e) [10 points] Calculate the number of cycles it takes to execute the code segment on Machine B. Show your work in the box.

Total number of cycles: 803.

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

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

7 Tomasulo's Algorithm [60 points]

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

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

7.1 Problem Definition

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

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

(a) Initial state of the RAT

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

(b) State of the RAT at the snapshot time

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

+

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

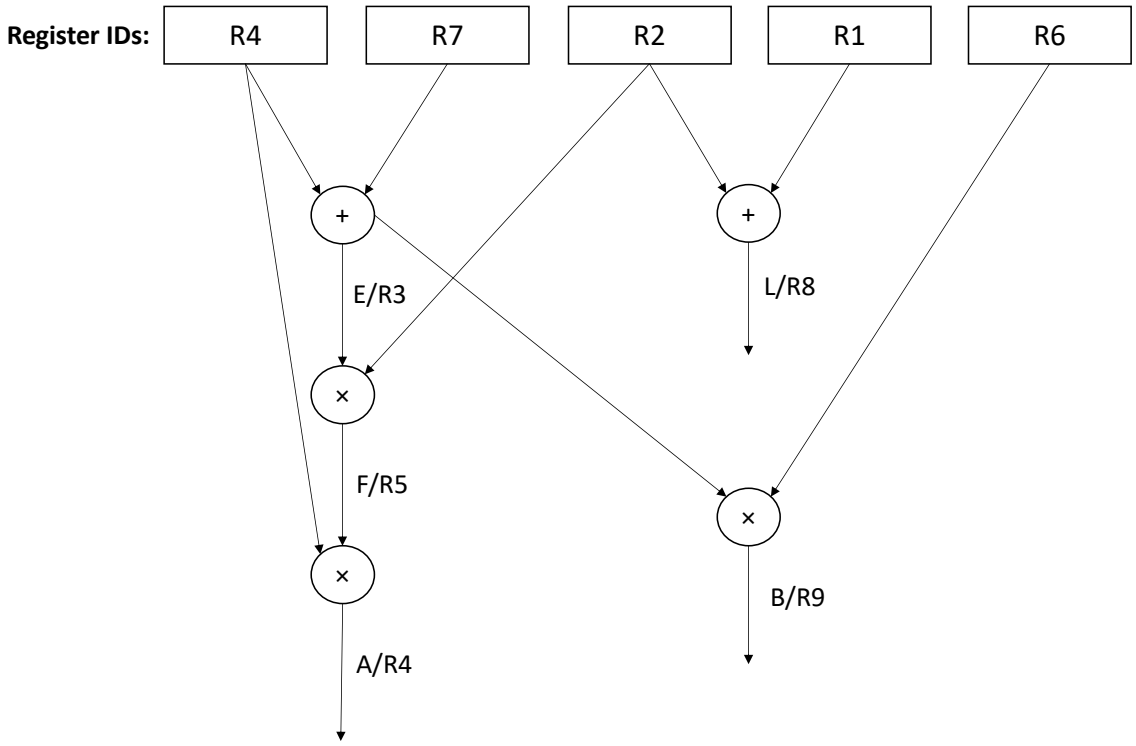
×

(c) State of the RS at the snapshot time

7.2 Questions

7.2.1 Dataflow Graph [40 points]

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



7.2.2 Program Instructions [20 points]

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

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

8 GPUs and SIMD [75 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.

The following code segments are run on a GPU. We assume that (1) `A` resides in memory and is shared by all threads, (2) `s` resides in a register and is private to each thread, and (3) the code segments are correct (i.e., do not think about any correctness issues when answering this question).

A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU. Each thread executes **a single iteration** of the outermost loop (with index `i`). Assume that the data values of the array `A` are already in vector registers so there are no memory loads and stores in this program. (Hint: Notice that there are 4 instructions in each iteration of the outermost loop of both code segments.)

```
s = 1;
for (i = 0; i < 1024; i++) {
    for (j = 0; j < 10; j++) { // Inst. 1
        if (i % (2 * s) == 0)  // Inst. 2
            A[i] += A[i + 1];  // Inst. 3
        s = s << 1;            // Inst. 4
    }
}
```

Code Segment 1

```
s = 512;
for (i = 0; i < 1024; i++) {
    for (j = 0; j < 10; j++) { // Inst. 1
        if (i < s)             // Inst. 2
            A[i] += A[i + s];  // Inst. 3
        s = s >> 1;            // Inst. 4
    }
}
```

Code Segment 2

Please answer the following questions.

(a) [5 points] How many warps does it take to execute these code segments?

32 warps.

Explanation:
The number of warps is calculated as:
$\#Warps = \lceil \frac{\#Total\_threads}{\#Warp\_size} \rceil$,

where
$\#Total\_threads = 1024 = 2^{10}$ (i.e., one thread per loop iteration),
and
$\#Warp\_size = 32 = 2^5$ (given).

Thus, the number of warps needed to run this program is:
$\#Warps = \lceil \frac{2^{10}}{2^5} \rceil = 2^5 = 32$.

(b) [10 points] What is the SIMD utilization of the first iteration of the inner loop (`j = 0`) for Code Segment 1? Show your work. (Hint: The warp scheduler does *not* issue instructions when no thread is active).

The utilization of the first iteration (`j = 0`) of Code Segment 1 is $\frac{7}{8}$.

Explanation:
Instructions 1, 2, and 4 are executed by all threads in Code Segment 1.

In Code Segment 1, `s = 1` during the first iteration. Thus, only even numbered threads fulfill the predicate of the `if` statement, and only half of the threads of each warp execute Instruction 3.
Code Segment 1, `j = 0`: $SIMD\_utilization = \frac{1024+1024+512+1024}{1024+1024+1024+1024} = \frac{7}{8}$.

(c) [10 points] What is the SIMD utilization of the first iteration of the inner loop (`j = 0`) for Code Segment 2? Show your work. (Hint: The warp scheduler does *not* issue instructions when no thread is active).

The utilization of the first iteration (`j = 0`) of Code Segment 2 is 100%.

Explanation:
Instructions 1, 2, and 4 are executed by all threads in Code Segment 2.

In Code Segment 2, `s = 512` during the first iteration. Thus, only threads with `i < 512` fulfill the predicate of the `if` statement, and all threads of only half of the warps execute Instruction 3.
Code Segment 2, `j = 0`: $SIMD\_utilization = \frac{1024+1024+512+1024}{1024+1024+512+1024} = \frac{7}{7} = 100\%$.

(d) [15 points] What is the SIMD utilization of any iteration of the inner loop (`0 <= j < 10`) for Code Segment 1? Show your work. (Hint: Derive an analytical expression, which may be piecewise).

As mentioned in part (b), Instructions 1, 2, and 4 are executed by all threads.

In Code Segment 1, with `0 <= j < 5`, all 32 warps are active, but the number of active threads per warp divides by half in each iteration. With `5 <= j < 10`, only one thread per warp is active, and the number of active warps divides by half in each iteration. As a result:

Code Segment 1, iteration `j`:

$$SIMD\_utilization = \begin{cases} \frac{3072+2^{(9-j)}}{4096}, & \text{if } 0 \leq j < 5 \\ \frac{3072+2^{(9-j)}}{3072+32*2^{(9-j)}}, & \text{if } 5 \leq j < 10 \end{cases} \quad (1)$$

(e) [15 points] What is the SIMD utilization of any iteration of the inner loop (`0 <= j < 10`) for Code Segment 2? Show your work. (Hint: Derive an analytical expression, which may be piecewise).

As mentioned in part (b), Instructions 1, 2, and 4 are executed by all threads.

In Code Segment 2, with `0 <= j < 5`, all 32 threads per warp are active, but the number of active warps divides by half in each iteration. With `5 <= j < 10`, only one warp is active, and the number of active threads divides by half in each iteration. As a result:

Code Segment 2, iteration `j`:

$$SIMD\_utilization = \begin{cases} \frac{3072+32*2^{(4-j)}}{3072+32*2^{(4-j)}} = 100\%, & \text{if } 0 \leq j < 5 \\ \frac{3072+2^{(9-j)}}{3072+32}, & \text{if } 5 \leq j < 10 \end{cases} \quad (2)$$

(f) [10 points] Is there any iteration (`0 <= j < 10`) where both code segments have the same utilization? Explain your reasoning.

Yes, with `j = 9` only one thread of only one warp is active, since only one thread (out of 1024) is needed to perform the last addition.

(g) [10 points] Which code is expected to run faster on a GPU? Explain your reasoning.

Code Segment 2 is faster because it has less intra-warp divergence, and thus higher SIMD utilization. In each iteration (except the last one), the number of warps that Code Segment 2 schedules is smaller than the number of warps that Code Segment 1 schedules. This results in fewer execution cycles.

9 Branch Prediction [45 points]

You are given the following piece of code that iterates through two large arrays, j and k, each populated with completely (i.e., truly) random positive integers. The code has five branches (labeled `B1`, `B2`, `B3`, `B4`, and `B5`). When we say that a branch is *taken*, we mean that the code inside the curly brackets is executed. Assume that the code is run to completion without any errors or interruptions (i.e., there are no exceptions). For the following questions, assume that this is the only block of code that will ever be run on the machines, and that the loop condition branch is resolved first in the iteration (i.e., the if statements execute only *after* resolving the loop condition branch).

```
1  for(int i = 0; i < 1000; i++) {     //B1
2                                      //TAKEN PATH for B1
3      if (i % 2 == 0) {               //B2
4          j[i] = k[i] * i;            //TAKEN PATH for B2
5      }
6      if (i < 250) {                  //B3
7          j[i] = k[i] - i;            //TAKEN PATH for B3
8      }
9      if (i < 500) {                  //B4
10         j[i] = k[i] + i;            //TAKEN PATH for B4
11     }
12     if (i >= 500) {                 //B5
13         j[i] = k[i] / i;            //TAKEN PATH for B5
14     }
15 }
```

Listing 1: Application to evaluate.

You are given three machines whose components are identical in every way, except for their branch predictors.

- Machine A uses an always-taken branch predictor.
- Machine B uses one single-level global two-bit saturating counter branch predictor *shared by all branches*, which starts at Weakly Taken (2'b10).
- Machine C uses a *per-branch* two-bit saturating counter as its branch predictor. All counters start at Weakly Not Taken (2'b01).

The saturating counter values are as follows:

- 2'b00 - Strongly Not Taken
- 2'b01 - Weakly Not Taken
- 2'b10 - Weakly Taken
- 2'b11 - Strongly Taken

Answer the following questions:

1. [15 points] What is the branch misprediction rate when the above piece of code runs on Machine A? Show your work.

$$45.01\% = \frac{2251}{5001}.$$

Explanation:
B1 will generate 1 misprediction out of 1001 iterations (B1 is not taken in the 1001th iteration and the loop body does not execute). B2 will generate 500 mispredictions out of 1000 iterations, B3 will generate 750 mispredictions out of 1000 iterations, and both B4 and B5 will generate 500 mispredictions out of 1000 iterations.

2. [15 points] What is the branch misprediction rate when the above piece of code runs on Machine B? Show your work.

$$59.97\% = \frac{2999}{5001}.$$

Explanation:
From (0-249): 375 mispredictions (125 for B2 and 250 for B5) for 1250 branches.
From (250-499): 874 mispredictions (2 for iteration 250, 4 for every odd iteration, 3 for every even iteration except for iteration 250) for 1250 branches.
From (500-1000): 1750 mispredictions (3 for odd iterations, 4 for even iterations, 0 for i = 1000) for 2501 branches.

3. [15 points] What is the branch misprediction rate when the above piece of code runs on Machine C? Show your work.

$$20.20\% = \frac{1010}{5001}.$$

Explanation:
You can split this up by branch.
B1: mispredicts at i = 0, and i = 1000 (2 mispredictions out of 1001).
B2: mispredicts every time since it oscillates between Weakly Not Taken and Weakly Taken (1000 mispredictions out of 1000).
B3: mispredicts at i = 0, i = 250, and i = 251 (3 mispredictions out of 1000).
B4: mispredicts at i = 0, i = 500, and i = 501 (3 mispredictions out of 1000).
B5: mispredicts at i = 500, and i = 501 (2 mispredictions out of 1000).

10 Caches [70 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 three 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 (2-, 4-, or 8-way).
- Cache replacement policy (LRU or FIFO).
- Cache size (4 or 8 KiB).

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 → Youngest)								Hit Rate
1.	0	16	24	25	1024	255	1100	305	2/8
2.	31	65536	65537	131072	262144	8	305	1060	3/8
3.	262145	65536	4						2/3

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

Based on what you observe, what are the following characteristics of the cache? Explain to get points.

(a) [20 points] Cache block size (8, 16, 32, 64, or 128 B)?

16 B.

Explanation:
Cache hit rate is 2/8 in sequence 1. This means that there are 2 hits. Depending on the cache block size, we can group addresses that belong to the same cache block as follows:

- **8B:** {0}, {16}, {24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 1.
- **16B:** {0}, {16,24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 2.
- **32B:** {0,16,24,25}, {255}, {305}, {1024}, {1100}. ∴ Number of possible hits = 3.
- **64B:** {0,16,24,25}, {255,305}, {1024}, {1100}. ∴ Number of possible hits = 4.
- **128B:** {0,16,24,25}, {255,305}, {1024,1100}. ∴ Number of possible hits = 5.

Therefore, we can know that the cache block size is 16B.

(b) [20 points] Cache associativity (2-, 4-, or 8-way)?

2-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 16B. Thus, there are 4 offset bits.

The access to address 31 in sequence 2 would hit because the cache block would not be replaced.
The access to address 305 in sequence 2 would hit because the cache block would not be replaced.
The access to address 65537 in sequence 2 would hit because the cache block would not be replaced.

Therefore, all the other accesses should miss.
The access to address 65536, 131072 and 262144 in sequence 2 would miss because addresses 65536, 131072 and 262144 do not belong to any cache block previously accessed.
Addresses 65536, 131072 and 262144 would be placed in set 0 if the cache associativity is 2-way, 4-way, or 8-way, independently of the cache size.
Address 8 must be a miss, so its cache block must be replaced by cache blocks that map to set 0 (addresses 65536, 131072 and 262144). For this to happen, the associativity must be 2-way.
Therefore, the cache is 2-way associative.

(c) [20 points] Cache replacement policy (LRU or FIFO)?

FIFO.

Explanation:
From questions (a) and (b), we already know the following facts:
- The cache block size is 16 B.
- The cache is 2-way.

Cache hit rate is 2/3 in sequence 3, which means that there are 2 hits.
With the LRU policy only the access to address 262145 in sequence 3 would hit. With the FIFO policy, accesses to addresses 262145 and 4 in sequence 3 would hit.
Therefore, the cache adopts the FIFO policy.

(d) [10 points] To identify the cache size (4 or 8KiB), you can access two addresses right after sequence 3 (i.e., the contents are the same as at the end of the third sequence) and measure the cache hit rate. Which two addresses would you choose? Explain your answer (there may be several correct answers).

Address 2048 and address 0 (there are other correct answers as well)

Explanation:
From questions (a), (b) and (c), we already know the following facts:
- The cache block size is 16 B.
- The cache is 2-way.
- FIFO replacement policy

We know that there are 4 bits for indexing the byte in a block, and there are 7 bits (if the cache size is 4KiB) or 8 bits (if the cache size is 8 KiB). Therefore, address 2048 would be in set 0 only if the cache size is 4KiB: we can access address 2048, and then check if a block in set 0 was replaced by address 2048 by accessing address 0. If it is a miss, the cache size is 4KiB, and if it is a hit, the cache size is 8KiB.

11 BONUS: Prefetching [25 points]

A runahead execution processor is designed with an unintended hardware bug: every other instruction in runahead mode is dropped by the processor after the fetch stage. Recall that the runahead mode is the speculative processing mode where the processor executes instructions solely to generate prefetch requests. All other behavior of the runahead mode is exactly as we described in lectures. When a program is executed, which of the following scenarios could happen compared to a runahead processor without the hardware bug and why? Circle YES if there is a possibility to observe the described behavior and explain in the box (either if you answer YES or NO). Assume that the program has no bug in it and executes correctly on the processor without the hardware bug.

(a) [8 points] The buggy runahead processor finishes the program *correctly* and *faster* than the non-buggy runahead processor.

YES *NO*

Why?

Dropping instructions enables the discovery of more cache misses than not dropping the instructions.

(b) [8 points] The buggy runahead processor finishes the program *correctly* and *slower* than the non-buggy runahead processor.

YES *NO*

Why?

The buggy runahead processor is not able to generate cache misses that are dependent on dropped instructions.

(c) [9 points] The buggy runahead processor executes the program *incorrectly*.

YES *NO*

Why?

Not possible as all executions in runahead mode is purely speculative and do not commit. Hence it cannot affect the correctness of the program.

12 BONUS: Systolic Arrays [35 points]

A systolic array consists of 4x4 Processing Elements (PEs), interconnected as shown in Figure 1. The inputs of the systolic array are labeled as H_0, H_1, H_2, H_3 and V_0, V_1, V_2, V_3. Figure 2 shows the PE logic, which performs a multiply and accumulate MAC operation and saves the result to an internal register (reg). Figure 2 also shows how each PE propagates its inputs. We make the following assumptions:

- The latency of each MAC operation is one cycle.
- 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.

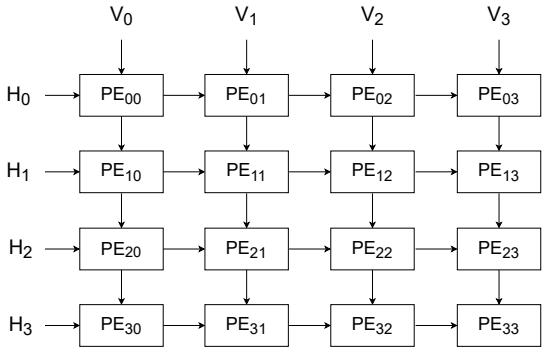


Figure 1: PE array

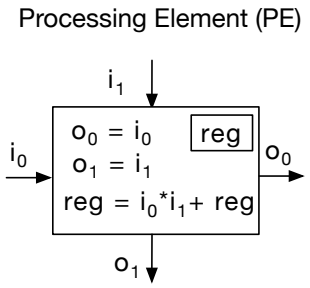


Figure 2: Processing Element (PE)

Your goal is to use the example systolic array shown in Figure 1 to perform the convolution ($\circledast$) of a 3x3 image (matrix I_{3x3}) with four 2x2 filters (matrices A_{2x2}, B_{2x2}, C_{2x2}, and D_{2x2}), to obtain four 2x2 outputs (matrices W_{2x2}, X_{2x2}, Y_{2x2}, and Z_{2x2}):

$$\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix} \circledast \begin{matrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{matrix} = \begin{matrix} W_{00} & W_{01} \\ W_{10} & W_{11} \end{matrix}$$

$$\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix} \circledast \begin{matrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{matrix} = \begin{matrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{matrix}$$

$$\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix} \circledast \begin{matrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{matrix} = \begin{matrix} Y_{00} & Y_{01} \\ Y_{10} & Y_{11} \end{matrix}$$

$$\begin{matrix} I_{00} & I_{01} & I_{02} \\ I_{10} & I_{11} & I_{12} \\ I_{20} & I_{21} & I_{22} \end{matrix} \circledast \begin{matrix} D_{00} & D_{01} \\ D_{10} & D_{11} \end{matrix} = \begin{matrix} Z_{00} & Z_{01} \\ Z_{10} & Z_{11} \end{matrix}$$

As an example, the convolution of the matrix I_{3x3} with the filter A_{2x2} is computed as follows:

- $W_{00} = I_{00} * A_{00} + I_{01} * A_{01} + I_{10} * A_{10} + I_{11} * A_{11}$
- $W_{01} = I_{01} * A_{00} + I_{02} * A_{01} + I_{11} * A_{10} + I_{12} * A_{11}$
- $W_{10} = I_{10} * A_{00} + I_{11} * A_{01} + I_{20} * A_{10} + I_{21} * A_{11}$
- $W_{11} = I_{11} * A_{00} + I_{12} * A_{01} + I_{21} * A_{10} + I_{22} * A_{11}$

You should compute the four convolutions in the minimum possible number of cycles. Fill the following table with:

1. The input elements (from matrices I_{3x3}, A_{2x2}, B_{2x2}, C_{2x2}, and D_{2x2}) in the correct input ports of the systolic array (H_0, H_1, H_2, H_3 and V_0, V_1, V_2, V_3). (Hint: If necessary, an input element can be concurrently streamed into several input ports of the array.)
2. The output values and the corresponding PE where the output elements (of matrices W_{2x2}, X_{2x2}, Y_{2x2}, and Z_{2x2}) are generated.

Fill the blanks only with relevant information.

cycle	**H0**	**H1**	**H2**	**H3**	**V0**	**V1**	**V2**	**V3**	PE_{00}	PE_{01}	PE_{02}	PE_{03}	PE_{10}	PE_{11}	PE_{12}	PE_{13}	PE_{20}	PE_{21}	PE_{22}	PE_{23}	PE_{30}	PE_{31}	PE_{32}	PE_{33}
0	A_{00}				I_{00}																			
1	A_{01}	B_{00}			I_{01}	I_{01}																		
2	A_{10}	B_{01}	C_{00}		I_{10}	I_{02}	I_{10}																	
3	A_{11}	B_{10}	C_{01}	D_{00}	I_{11}	I_{11}	I_{11}	I_{11}	W_{00}															
4		B_{11}	C_{10}	D_{01}		I_{12}	I_{20}	I_{12}		W_{01}			X_{00}											
5			C_{11}	D_{10}			I_{21}	I_{21}			W_{10}			X_{01}			Y_{00}							
6				D_{11}				I_{22}				W_{11}			X_{10}			Y_{01}			Z_{00}			
7																X_{11}			Y_{10}			Z_{01}		
8																				Y_{11}			Z_{10}	
9																								Z_{11}
10																								
11																								
12																								
13																								
14																								
15																								