Family Name: SOLUTIONS First Name: Student ID:

Final Exam

Design of Digital Circuits (252-0028-00L)

ETH Zürich, Spring 2019

Prof. Onur Mutlu

Problem 1 (12 Points):	Boolean Algebra	
Problem 2 (20 Points):	Verilog	
Problem 3 (30 Points):	Finite State Machines (FSM)	
Problem 4 (20 Points):	ISA vs. Microarchitecture	
Problem 5 (20 Points):	Performance Evaluation	
Problem 6 (40 Points):	Pipeline (Reverse Engineering)	
Problem 7 (36 Points):	Tomasulo's Algorithm	
Problem 8 (30 Points):	Systolic Arrays	
Problem 9 (35 Points):	GPUs and SIMD	
Problem 10 (40 Points):	Reverse Engineering Caches	
Problem 11 (30 Points):	Dataflow	
Problem 12 (BONUS: 30 Points):	Branch Prediction	
Total (343 (313 + 30 bonus) Points):		

Examination Rules:

1. Written exam, 180 minutes in total.
2. **No books, no calculators, no computers or communication devices**. 3 double-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 or red 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 Algebra [12 points]

(a) [6 points] Find the simplest sum-of-products representation of the following Boolean equation. Show your work step-by-step.

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

$F = B.C + A$

Explanation:
$F = (\overline{A}.A + \overline{A}.B + \overline{A}.\overline{C} + B.A + B.B + B.\overline{C} + C.A + C.B + C.\overline{C}).C + A$
$F = (0 + B.(\overline{A} + A) + \overline{A}.\overline{C} + B + B.(\overline{C} + C) + C.A + 0).C + A$
$F = (B + \overline{A}.\overline{C} + B + B + C.A).C + A$
$F = (B.C + \overline{A}.\overline{C}.C + B.C + C.A.C) + A$
$F = (B.C + 0 + C.A) + A$
$F = B.C + A.(C + 1)$
$F = B.C + A$

(b) [6 points] Convert the following Boolean equation so that it contains only NAND operations. Show your work step-by-step.

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

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

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

2 Verilog [20 points]

Please answer the following three questions about Verilog.

(a) [5 points] Does the following code result in a single D Flip-Flop with a synchronous active-low reset? Please explain your answer.

```
module mem (input clk, input reset, input [1:0] d, output reg [1:0] q);
always @ (posedge clk or negedge reset)
    begin
      if (!reset) q <= 0;
      else q <= d;
    end
endmodule
```

No.
The code implements *two* D Flip-Flops, not *one*. Each D Flip-Flop works with an *asynchronous* active-low reset signal.

Explanation:

- D and Q signals are two-bit-wide. Therefore, this code implements two D flip-flops.
- The reset input is included in the sensitivity list, therefore it is not synchronous.
- The code resets the output if the reset signal is low. Thus, the reset signal is active-low.

(b) [5 points] Does the following code result in a sequential circuit or a combinational circuit? Please explain your answer.

```
module Mask (input [1:0] data_in, input mask, output reg [1:0] data_out);
always @ (*)
    begin
        data_out[1] = data_in[1];
        if (mask)
            data_out[0] = 0;
    end
endmodule
```

Sequential circuit.

Explanation:
This code results in a sequential circuit, as all the left-hand side signals are not assigned in every possible condition. For example, `data_out[0]` is not assigned when mask signal equals to zero.

(c) [10 points] Is the following code syntactically correct? If not, please explain the mistake(s) and how to fix it/them.

```
1   module fulladd(input a, b, c, output reg s, c_out);
2      assign s = a^b;
3      assign c_out = (a & b) | (b & c) & (c & a);
4   endmodule
5
6   module top ( input wire [5:0] instr, input wire op, output z);
7
8     reg[1:0] r1, r2;
9     wire [3:0] w1, w2;
10
11    fulladd FA1 (.a(instr[0]), .b(instr[1]), .c(instr[2]),
12                               .c_out(r1[1]), .z(r1[0]));
13    fulladd FA2 (.a(instr[3]), .b(instr[4]), .c(instr[5]),
14                               .z(r2[0]), .c_out(r2[1]));
15
16    assign z = r1 | op;
17    assign w1 = r1 + 1;
18    assign w2 = r2 << 1;
19    assign op = r1 ^ r2;
20
21  endmodule
```

The code is *not* syntactically correct.

Explanation:

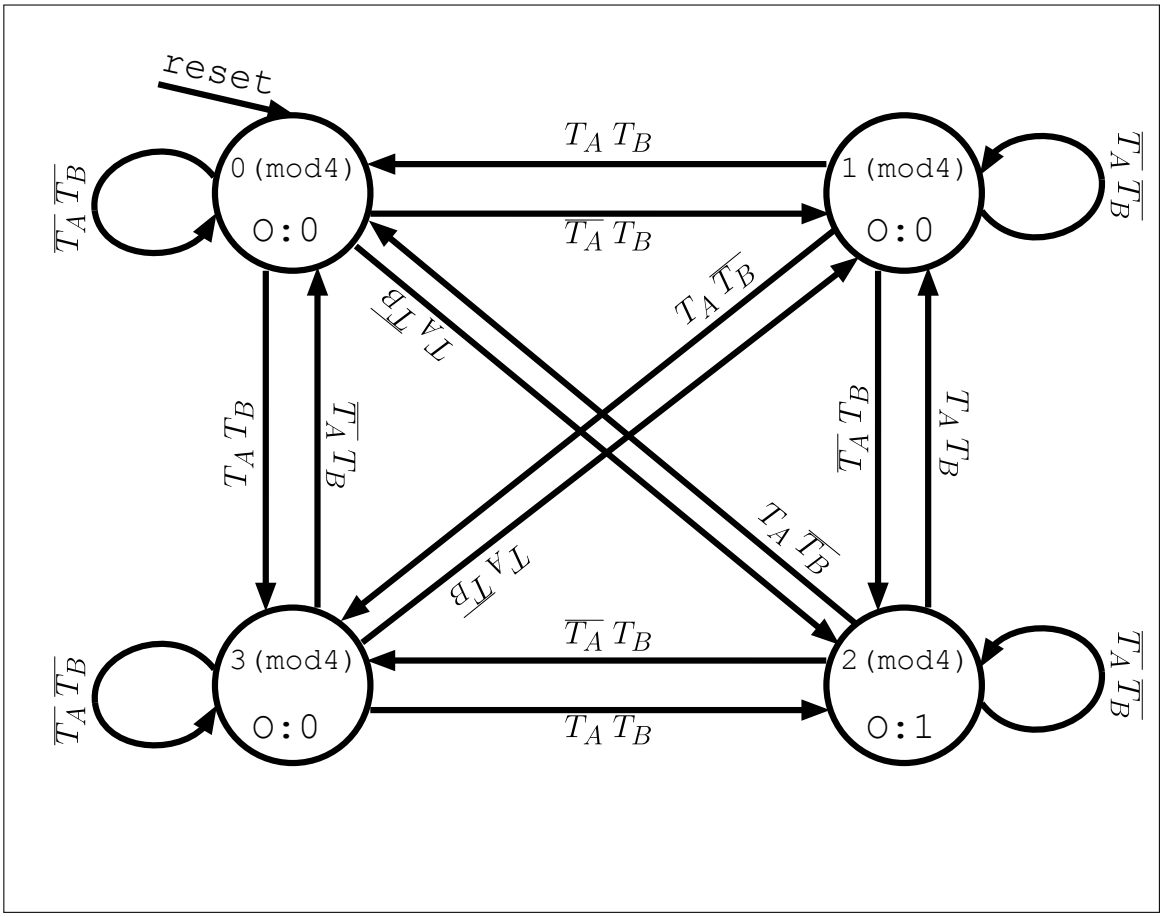
- 'r1' and 'r2' have to be declared as *wires*.
- 'op' signal is connected to multiple drivers. It gets assigned from the input port and in line 19.
- The module 'fulladd' does not have ports named 'z'. Those need to be changed to 's'.
- The output signals 's' and 'c_out' have to be declared as *wires* but not as *regs*, since they are driven by *assign* statements.

3 Finite State Machines (FSM) [30 points]

You are given two *one-bit* input signals (T_A and T_B) and one *one-bit* output signal (O) for the following modular equation: $2N(T_A)+N(T_B) \equiv 2 \pmod 4$. In this modular equation, $N(T_A)$ and $N(T_B)$ represent the **total number of times** the inputs T_A and T_B are high (i.e., logic 1) at each positive clock edge, respectively. The one-bit output signal, O, is set to 1 when the modular equation is satisfied (i.e., $2N(T_A) + N(T_B) \equiv 2 \pmod 4$), and 0 otherwise. An example that sets $O = 1$ at the end of the fourth cycle would be:

- (1^{st} cycle) $T_A = 0$ ($N(T_A) = 0$), $T_B = 0$ ($N(T_B) = 0$), $2N(T_A) + N(T_B) \equiv 0 \pmod 4 \Rightarrow O = 0$
- (2^{nd} cycle) $T_A = 1$ ($N(T_A) = 1$), $T_B = 1$ ($N(T_B) = 1$), $2N(T_A) + N(T_B) \equiv 3 \pmod 4 \Rightarrow O = 0$
- (3^{rd} cycle) $T_A = 1$ ($N(T_A) = 2$), $T_B = 0$ ($N(T_B) = 1$), $2N(T_A) + N(T_B) \equiv 1 \pmod 4 \Rightarrow O = 0$
- (4^{th} cycle) $T_A = 0$ ($N(T_A) = 2$), $T_B = 1$ ($N(T_B) = 2$), $2N(T_A) + N(T_B) \equiv 2 \pmod 4 \Rightarrow O = 1$

(a) [10 points] You are given a partial **Moore** machine state transition diagram that corresponds to the modular equation described above. However, the input labels of most of the transitions are still missing in this diagram. Please label the transitions with the correct inputs so that the FSM correctly implements the above specification.



(b) [10 points] Describe the FSM with Boolean equations assuming that the states are encoded with **one-hot encoding**. Assign state encodings while using the **minimum** possible number of bits to represent the states. Please indicate the values you assign to each state.

State assignments: 0 (mod 4): 0001, 1 (mod 4): 0010, 2 (mod 4): 0100, 3 (mod 4): 1000
CS denotes current states, and NS denotes next states.
$NS[0] = CS[0]\ \overline{T_A}\ \overline{T_B} + CS[1]\ T_A\ T_B + CS[2]\ T_A\ \overline{T_B} + CS[3]\ \overline{T_A}\ T_B$
$NS[1] = CS[1]\ \overline{T_A}\ \overline{T_B} + CS[2]\ T_A\ T_B + CS[3]\ T_A\ \overline{T_B} + CS[0]\ \overline{T_A}\ T_B$
$NS[2] = CS[2]\ \overline{T_A}\ \overline{T_B} + CS[3]\ T_A\ T_B + CS[0]\ T_A\ \overline{T_B} + CS[1]\ \overline{T_A}\ T_B$
$NS[3] = CS[3]\ \overline{T_A}\ \overline{T_B} + CS[0]\ T_A\ T_B + CS[1]\ T_A\ \overline{T_B} + CS[2]\ \overline{T_A}\ T_B$
$O[0] = CS[2]$

(c) [10 points] Describe the FSM with Boolean equations assuming that the states are encoded with **binary encoding** (i.e., fully encoding). Assign state encodings while using the **minimum** possible number of bits to represent the states. Please indicate the values you assign to each state.

State assignments: 0 (mod 4): 00, 1 (mod 4): 01, 2 (mod 4): 10, 3 (mod 4): 11
CS denotes current states, and NS denotes next states.
$NS[0] = \overline{CS[0]}\ T_B + CS[0]\ \overline{T_B}$
$NS[1] = CS[0]\ (CS[1]\ \text{XOR}\ T_A\ \text{XOR}\ T_B) + \overline{CS[0]}\ (T_A\ \text{XOR}\ CS[1])$
$O[0] = CS[1]\ \overline{CS[0]}$

4 ISA vs. Microarchitecture [20 points]

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

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

Table 1: Manual Costs

Unfortunately, the manuals are extremely expensive, and you can only afford one of the two. If both manuals might be useful, you would prefer the cheaper one.

For each of the following questions that you would like to answer, decide which manual is more likely to help. *Note: we will subtract 1 point for each* ***incorrect*** *answer. For an unanswered question, you will get +0 points.*

1. [2 points] The latency of a branch predictor misprediction.

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

2. [2 points] The size of a physical memory page.

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

3. [2 points] The memory-mapped locations of exception vectors.

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

4. [2 points] The function of each bit in a programmable branch-predictor configuration register.

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

5. [2 points] The bit-width of the interface between the CPU and the L1 cache.

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

6. [2 points] The number of pipeline stages in the CPU.

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

7. [2 points] The order in which loads and stores are executed by a multi-core CPU.

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

8. [2 points] The memory addressing modes available for arithmetic operations.

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

9. [2 points] The program counter width.

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

10. [2 points] The number of cache sets at each level of the cache hierarchy.

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

5 Performance Evaluation [20 points]

You are the leading engineer of a new processor. Both the design of the processor and the compiler for it are already done. Now, you need to decide if you will send the processor to manufacturing at its current stage or if you will delay the production to introduce last-minute improvements to the design. To make the decision, you meet with your team to brainstorm about how to improve the design. Together, after profiling the target applications for the processor, you come up with two options:

- **Keep the current project.** For version A of the processor, the clock frequency is 600 MHz, and the following measurements are obtained:

Instruction Class	CPI	Frequency of Occurrence
A	2	40%
B	3	25%
C	3	25%
D	7	10%

- **Include optimizations to the design.** For version B of the processor, the clock frequency is 700 MHz. The ISA for processor B includes three new types of instructions. Those three new types of instructions increase the total number of executed instructions for processor B by 50%, in comparison to processor A. The following measurements are obtained:

Instruction Class	CPI	Frequency of Occurrence
A	2	15%
B	2	15%
C	4	10%
D	6	10%
E	1	10%
F	2	20%
G	2	20%

(a) [7 points] What is the CPI of each version? Show your work.

CPI_A:

3

CPI_B:

2.5

$$CPI_A = 2 \times 0.4 + 3 \times 0.25 + 3 \times 0.25 + 7 \times 0.1 = 3$$
$$CPI_B = 2 \times 0.15 + 2 \times 0.15 + 4 \times 0.1 + 6 \times 0.1 + 1 \times 0.1 + 2 \times 0.2 + 2 \times 0.2 = 2.5$$

(b) [6 points] What are the MIPS (Million Instructions Per Second) of each version? Show your work.

$MIPS_A$:

200

$MIPS_B$:

280

$MIPS_A = \frac{600MHz}{3*10^6} = 200$

$MIPS_B = \frac{700MHz}{2.5*10^6} = 280$

(c) [7 points] Considering your team is aiming to release to the market the processor that gives better performance when executing the target application, which processor version will you choose as the final design? Show your work.

Processor A.

Explanation:
We calculate the execution time for each processor, $Time = N_{instr.} \times CPI \times \frac{1}{clockfrequency}$
Since the compiler for processor B generates 50% more instructions than the compiler for processor A, the total execution time for processor B is larger than the total execution time for processor A.

$Time_A = N_{instr.} \times 3 \times \frac{1}{600*10^6}$

$Time_B = 1.5N_{instr.} \times 2.5 \times \frac{1}{700*10^6}$

6 Pipeline (Reverse Engineering) [40 points]

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

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

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

(a) [5 points] How many cycles does it take for an adder and for a multiplier to calculate a result?

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

(b) [5 points] What is the minimum number of register file read/write ports that this architecture implements? Explain.

The register file has two read ports and one write port.

(c) [5 points] Can we reduce the execution time of this code by enabling more read/write ports in the register file? Explain.

It is not possible to reduce stall cycles of the given code by enabling more register file ports.

(d) [5 points] Does this architecture implement any data forwarding? If so, how is data forwarding done between pipeline stages? Explain.

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

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

(e) [5 points] Is it possible to run this code faster by adding more data forwarding paths? If it is, how? Explain.

Not possible.

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

(f) [5 points] Is there internal forwarding in the register file? If there is not, how would the execution time of the same program change by enabling internal forwarding in the register file? Explain.

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

(g) [10 points] Optimize the assembly code in order to reduce the number of stall cycles. You are allowed to *reorder*, *add*, or *remove* ADD and MUL instructions. You are expected to achieve the minimum possible execution time. Make sure that the register values that the optimized code generates at the end of its execution are identical to the register values that the original code generates at the end of its execution. Justify each individual change you make. Show the execution timeline of each instruction and what stage it is in the table below. (*Notice that the table below consists of two parts: the first ten cycles at the top, and the next ten cycles at the bottom.*)

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

New total execution time is 17 cycles instead of 18.

Instr. No	Instructions	Cycles									
		1	2	3	4	5	6	7	8	9	10
0	MUL R5, R6, R7	F	D	E1	E2	E3	E4	M	W		
3	MUL R4, R7, R7		F	D	E1	E2	E3	E4	M	W	
2	ADD R5, R5, R6			F	D	-	-	E1	E2	E3	M
4	ADD R6, R7, R5				F	-	-	D	-	-	E1
6	ADD R7, R1, R4							F	-	-	D
5	ADD R3, R0, R6										F

		11	12	13	14	15	16	17	18	19	20
0	MUL R5, R6, R7										
3	MUL R4, R7, R7										
2	ADD R5, R5, R6	W									
4	ADD R6, R7, R5	E2	E3	M	W						
6	ADD R7, R1, R4	E1	E2	E3	M	W					
5	ADD R3, R0, R6	D	-	E1	E2	E3	M	W			

7 Tomasulo's Algorithm [36 points]

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

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

The reservation stations are all initally empty. The processor fetches and executes *six* instructions. Table 2 shows the six instructions and their execution diagram.

Using the information provided above and in Table 2 (see the next page), fill in the blanks below with the configuration of the out-of-order microarchitecture. Write "Unknown" if the corresponding configuration cannot be determined using the information provided in the question.

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

Instruction/Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
1: ADD R1 ← R0, R1	F	D	E1	E2	W																												
2: LD R2 ← [R1]	F	D	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W																	
3: ADDI R1 ← R1, #4		F	D	-	-	E1	E2	W																									
4: LD R3 ← [R1]		F	D	-	-	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W														
5: MUL R4 ← R2, R3			F	-	-	D	-	-	-	-	-	-	-	-	-	-	-	-	-	E1	E2	W											
6: ST [R0] ← R4			F	-	-	-	-	-	-	-	-	-	-	-	-	-	D	-	-	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	W

Table 2: Execution diagram of the six instructions.

8 Systolic Arrays [30 points]

A systolic array consists of 3x4 Processing Elements (PEs), interconnected as shown in Figure 1. The inputs of the systolic array are labeled as H0, H1, H2 and V0,V1,V2,V3. Figure 2 shows the PE logic, which performs a multiply and accumulate operation (MAC), and it saves the result in an internal register (reg). Figure 2 also shows how each PE propagates its inputs. We make the following assumptions:

- The latency of each MAC 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 value of all registers is zero.
- You can input a value more than once in the systolic array.

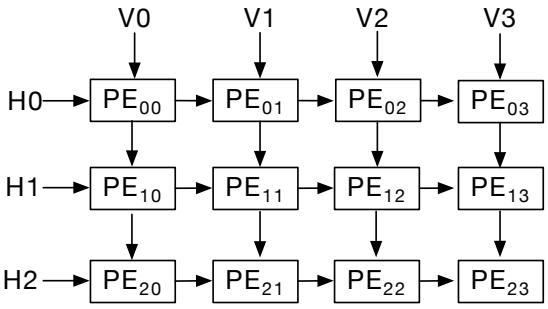


Figure 1: PE array

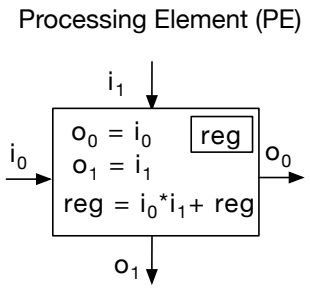


Figure 2: Processing Element (PE)

Your goal is to use this systolic array to perform the convolution of a 3x3 image (matrix I) with three 2x2 filters (matrices F, G, and H), to obtain three outputs (matrices O, U, and E):

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

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

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

As an example, the convolution of the matrix I with the filter F is computed as follows:

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

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

1. The input values (matrices I, F, G, and H) in the correct input ports of the systolic array (the values can be repeated).
2. The output values and the corresponding PE where the outputs (matrices O, U, and E) are generated.

Fill the gaps only with relevant information.

cycle	**H0**	**H1**	**H2**	**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}
0	F_{00}			I_{00}															
1	F_{01}	G_{00}		I_{01}	I_{01}														
2	F_{10}	G_{01}	H_{00}	I_{10}	I_{02}	I_{10}													
3	F_{11}	G_{10}	H_{01}	I_{11}	I_{11}	I_{11}	I_{11}	O_{00}											
4		G_{11}	H_{10}		I_{12}	I_{20}	I_{12}		O_{01}			U_{00}							
5			H_{11}			I_{21}	I_{21}			O_{10}			U_{01}			E_{00}			
6							I_{22}				O_{11}			U_{10}			E_{01}		
7															U_{11}			E_{10}	
8																			E_{11}
9																			
10																			
11																			
12																			
13																			
14																			
15																			

9 GPUs and SIMD [35 points]

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

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays `A` and `B` are already in vector registers so there are no loads and stores in this program. (Hint: Notice that there are 4 instructions in each iteration.) A warp in the GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU.

```
for (i = 0; i < 1026; i++) {
    if (A[i] < 33) {         // Instruction 1
        B[i] = A[i] << 1;    // Instruction 2
    }
    if (A[i] > 33) {         // Instruction 3
        B[i] = A[i] >> 1;    // Instruction 4
    }
}
```

Please answer the following five questions.

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

33 warps.

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

where
$\#Total\_threads = 1026 = 2^{10} + 2$ (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:
$\#Warps = \lceil \frac{2^{10}+2}{2^5} \rceil = 2^5 + 1 = 33$.

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

$\frac{3076}{3136} = \frac{769}{784}$.

Explanation:
The maximum SIMD utilization is achieved when all threads of the complete warps follow the same execution path and execute `Instruction 2` or `Instruction 4` (`A[i] > 33` or `A[i] < 33`), and the two active threads of the last warp do not execute `Instruction 2` or `Instruction 4` (`A[i] = 33`).
The maximum SIMD utilization sums to $\frac{1026+32\times32+1026}{1056+1024+1056} = \frac{3076}{3136}$.

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

For every 32 consecutive elements of `A` out of the first 1024 elements, every element should be lower than 33 (`if(A[i] < 33)`), or greater than 33 (`if(A[i] > 33)`). The last two elements should be equal to 33. (NOTE: The solution is correct if the three cases are given.)

(d) [13 points] What is the minimum possible SIMD utilization of this program? Show your work.

$\frac{353}{704}$.

Explanation:
`Instruction 1` is executed by every active thread ($\frac{1026}{1056}$ utilization).
The minimum SIMD utilization of `Instruction 2` occurs if only one thread per warp executes it.
`Instruction 3` is again executed by every active thread ($\frac{1026}{1056}$ utilization).
Finally, the minimum SIMD utilization of `Instruction 4` occurs if only one thread per warp executes it.

The minimum SIMD utilization sums to $\frac{1026+1\times33+1026+1\times33}{1056+1056+1056+1056} = \frac{353}{704}$.

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

For every 32 consecutive elements among the first 1024 elements of `A`, one element should be lower than 33 (`if(A[i] < 33)`), one element should be greater than 33 (`if(A[i] > 33)`), and the remaining 30 elements should be equal to 33.
For the last 2 elements of `A`, one element should be lower than 33 (`if(A[i] < 33)`), and the other element should be greater than 33 (`if(A[i] > 33)`).

10 Reverse Engineering Caches [40 points]

You are trying to reverse-engineer the characteristics of a cache in a system, so that you can design a more efficient, machine-specific implementation of an algorithm you are working on. To do so, you have come up with 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 (1-, 2-, 4-, or 8-way).
- Cache size (4 or 8 KB).
- Cache replacement policy (LRU or FIFO).

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

Sequence	Addresses Accessed (Oldest → Youngest)								Hit Rate
1.	31	8192	63	16384	4096	8192	64	16384	3/8
2.	32768	0	129	1024	3072	8192			0
3.	0	4	8	4096	64	128			1

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

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

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

64 B.

Explanation:
Cache hit rate is 3/8 in sequence 1. This means that there are 3 hits. As two of them should be the second accesses to 8192 and 16384, the other hit is the access to 63. With a cache block of 64 B, the access to address 64 results in a miss.

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

4-way.

Explanation:
We already know that the cache block size is 64 B. Thus, there are 6 offset bits.

Regardless of cache size or associativity, addresses 0, 8192, 16384, and 32768 map to the same set. Thus, the cache cannot be 1-way, because we would not see hits on 8192 and 16384 in sequence 1.

If the cache were 2-way, 4096 would also map to the same set as 0, 8192, 16384, and 32768. This would make impossible a cache hit on 8192 in sequence 1.

If the cache were 8-way, 0, 1024, 3072, 4096, 8192, 16384, and 32768 would all map to set 0. With 8 ways, address 0 would not be replaced, so it would hit in sequence 2.

Therefore, the cache is 4-way associative.

(c) [10 points] Cache size (4 or 8 KB)?

8 KB.

Explanation:
We know that the cache is 4-way associative. In the beginning of sequence 2, 32768 replaces 0 (regardless of the replacement policy).

The fact that 8192 misses in sequence 2 can be explained by two possible cases:
1. If the replacement policy is FIFO, the access to 0 in sequence 2 replaces 8192. Thus, the cache size can be either 4 or 8 KB.
2. If the replacement policy is LRU, the access to 0 in sequence 2 replaces 4096. If the cache size is 4 KB, 1024 and 3072 map to the same set as 0 and 8192, and 1024 replaces 8192.

Since there is a hit on 4096 in sequence 3, the size should be 8 KB. Otherwise, 3072 would have replaced 4096.

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

FIFO.

Explanation:
As explained above, if the cache size is 8 KB, only FIFO can make address 0 replace address 8192 in sequence 2.

11 Dataflow [30 points]

- We define the *switch node* in Figure 3 to have 2 inputs (**I**, **Ctrl**) and 1 output (**O**). The *Ctrl* input always enters perpendicularly to the switch node. If the *Ctrl* input has a *True* token (i.e., a token with a value of 1), the **O** wire propagates the value on the **I** wire. Else, the 2 input tokens (**I**, **Ctrl**) are consumed, and no token is generated at the output (**O**).
- We define the *inverter node* in Figure 4 to have 1 input (**I**) and 1 output (**O**). The node negates the input token (i.e., O = !I).
- We define the *TF node* in Figure 5 to have 3 inputs (I_F, I_T, **Ctrl**) and 1 output (**O**). When **Ctrl** is set to True, **O** takes I_T. When **Ctrl** is set to False, **O** takes I_F.
- The $\geq$ node outputs True only when the left input is greater than or equal to the right input.
- The +1 node outputs the input plus one.
- The + node outputs the sum of the two inputs.
- A node generates an output token when tokens exist at *every* input, and *all* input tokens are consumed.
- Where a single wire splits into multiple wires, the token travelling on the wire is replicated to all wires.

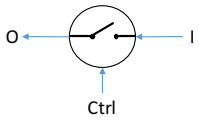


Figure 3: Switch Node

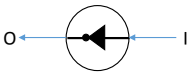


Figure 4: Inverter Node

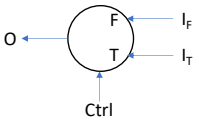


Figure 5: TF Node

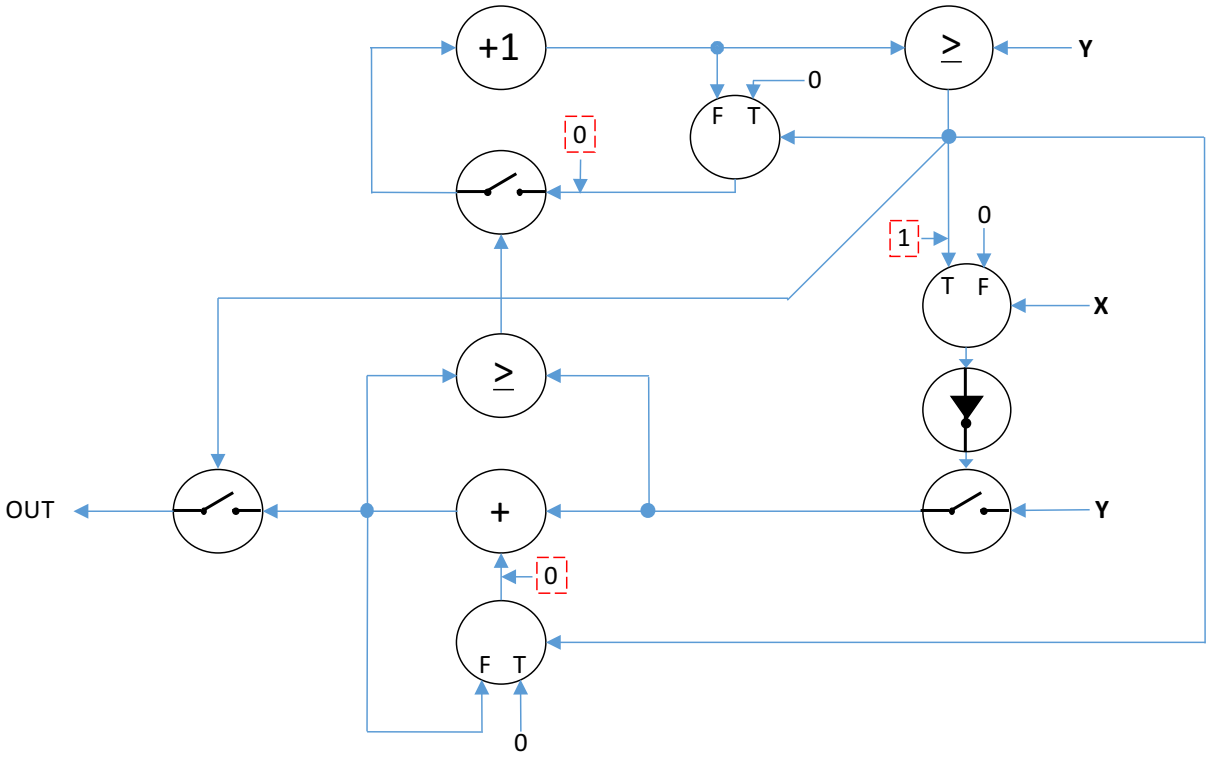
Consider the dataflow graph on the following page. Numbers in dashed boxes represent tokens (with the value indicated by the number) in the initial state. The **X** and **Y** inputs automatically produce tokens as soon as the previous token on the wire is consumed. The order of these tokens follows the pattern (*note, the following are all single digit values spaced appropriately for the reader to easily notice the pattern*):

X: 0 01 011 0111 01111

Y: 1 22 333 4444 55555

Consider the dataflow graph on the following page. Please clearly describe the sequence of tokens generated at the output (OUT).

1, 4, 9, 16, 25


+1
≥
Y
0
F T
0
1
0
T F
X
≥
OUT
+
Y
0
F T
0

12 BONUS: Branch Prediction [30 points]

Assume a machine with a two-bit global history register (GHR) shared by all branches, which starts with Not Taken, Not Taken (2'b00). Each pattern history table entry (PHTE) contains a 2-bit saturating counter. 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

Assume the following piece of code runs on this machine. The code has two branches (labeled `B1` and `B2`). When we say that a branch is taken, we mean that the code inside the curly brackets is executed. For the following questions, assume that this is the only block of code that will ever be run, and the loop-condition branch (`B1`) is resolved first in the iteration before the if-condition branch (`B2`).

```
for (int i = 0; i < 1000000; i++) { /* B1 */
                                    /* TAKEN PATH for B1 */
    if (i % 3 == 0) {               /* B2 */
        j[i] = k[i] -1;             /* TAKEN PATH for B2 */
    }
}
```

(a) [20 points] Is it possible to observe that the branch predictor mispredicts 100% of the times in the first 5 iterations of the loop? If yes, fill in the table below with all possible initial values each entry can take. We represent Not Taken with N, and Taken with T.

Table 3: PHT

PHT Entry	Value
TT	01
TN	00
NT	01
NN	00 or 01

Show your work here.

Yes, it is possible.
The pattern after 5 iterations: TTTNTNTTTN.
In order to be more clear, we add indices to each branch outcome in the pattern above, to represent their positions in the pattern: T_1 T_2 T_3 N_4 T_5 N_6 T_7 T_8 T_9 N_{10}

- For GHR=NN, the only observed branch is T_1. Therefore, the PHTE for NN has to be either 00 or 01 so that the branch predictor mispredicts the taken branch.
- For GHR=TT, the observed branches are T_3 N_4 T_9 N_{10}. The PHTE for TT has to be initialized to 01 in order to cause the predictor to always mispredict. This way, each N and T moves the saturating counter to their respective direction. This will cause misprediction for the next branch which is always in the opposite direction.
- For GHR=TN, the observed branches are T_5 T_7. Thus, the initial PHTE value for TN has to be 00 to mispredict both taken branches.
- For GHR=NT, the observed branches are T_2 N_6 T_8. Similar to the TT entry, NT's PHTE has to be initialized 01.

(b) [10 points] At steady-state, we observe the following pattern which repeats over time: TTTNTN, with T representing Taken, and N representing Not Taken. When GHR pattern equals to NT or TT, the predictor will observe that the branch outcome will be either T or N. Therefore, no matter what the initial values for these two entries are in the pattern history table (PHT), only one of the branches can be predicted correctly. Thus prediction accuracy will never reach 100%. Explain how using local history registers instead of the global history register will help bring the prediction accuracy up to 100% during the steady state, by showing what each PHTE will saturate to.

For the outer loop, we will keep observing all Ts, and the counters will be set to 2'b11 for TT and lead to 100% accuracy for this branch.
The second branch will keep observing this repeated pattern: TNN. So entry TN will be saturated to 2'b00, entry NN will saturate to 2'b11, and entry NT will saturate to 2'b00.