Digital Design & Computer Arch.

Problem Solving IV

Prof. Onur Mutlu

ETH Zürich

Spring 2023

12 July 2023

Problem Solving IV

- We solve 11 questions from Final Exam Spring 2020
  - Boolean Circuit Minimization (Q1)
  - Verilog (Q2)
  - Finite State Machine (Q3)
  - ISA vs. Microarchitecture (Q4)
  - Performance Evaluation (Q5)
  - Pipelining (Q6)
  - Tomasulo's Algorithm (Q7)
  - GPUs and SIMD (Q8)
  - Caches (Q9)
  - Branch Prediction (Q10)
  - VLIW (Q11)

Boolean Circuit Minimization (Q1)

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

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

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

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

Verilog

(Q2)

Please answer the following four questions about Verilog

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

```
1
2  module sevensegment (input [3:0] data, output reg [6:0] segments);
3      always @ ( * )
4          case (data)
5              4'd0: segments = 7'b111_1110;
6              4'd1: segments = 7'b011_0000;
7              4'd2: segments = 7'b110_1101;
8              4'd3: segments = 7'b111_1001;
9              4'd4: segments = 7'b011_0011;
10         endcase
11 endmodule
```

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

```
1  module divideby3 (input clk, input reset, output q);
2      reg [1:0] curVal, nextVal;
3      parameter S0 = 2'b00; parameter S1 = 2'b01; parameter S2 = 2'b10;
4      always @ (*)
5          case (curVal)
6              S0: nextVal = S1;
7              S1: nextVal = S2;
8              S2: nextVal = S0;
9              default: nextVal = S0;
10         endcase
11     assign q = (curVal == S0);
12 endmodule
```

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

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

```
1  module mod1 (input sel, input a, input b, input c, output out);
2      reg tmp = 1'b0;
3      always @ (sel)
4          if (sel)
5              tmp <= ~(a & b);
6              out <= tmp ^ c;
7          else
8              tmp <= 0;
9              out <= 0;
10 endmodule
```

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

```
1  module top (input [1:0] in1, in2 , input op, output reg [1:0] z, output reg s);
2
3  wire tmp;
4  always@(*) begin
5      tmp = in1[0] & in2[0];
6      z[0] = tmp & op;
7  end
8  always@(*) begin
9      tmp = in1[1] | in2[1];
10     z[1] = tmp & (~p)
11 end
12 assign s = (z[1] > z[0])
13 endmodule
```

Finite State Machines (Q3)

3.1 Designing an FSM

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

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

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

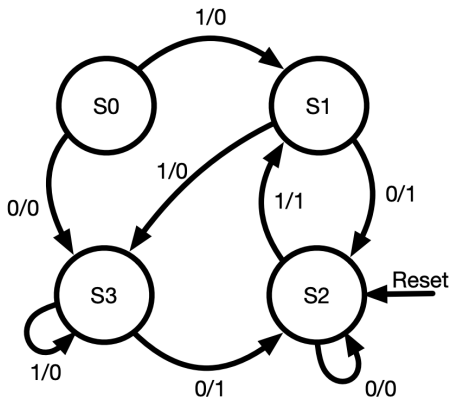
3.2 Simplifying an FSM

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

(a) Is this a Mealy or a Moore machine? Explain why.

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

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



ISA vs. Microarchitecture (Q4)

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

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

Table 1: Manual Costs

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

1.	The integer multiplication algorithm used by the ALU.	1. the_isa.pdf 2. the_microarchitecture.pdf
2.	The program counter width.	1. the_isa.pdf 2. the_microarchitecture.pdf
3.	Branch misprediction penalty.	1. the_isa.pdf 2. the_microarchitecture.pdf
4.	The ability to flush the TLB from the OS.	1. the_isa.pdf 2. the_microarchitecture.pdf
5.	The size of the Reorder Buffer in an Out-of-Order CPU.	1. the_isa.pdf 2. the_microarchitecture.pdf
6.	The fetch width of a superscalar CPU.	1. the_isa.pdf 2. the_microarchitecture.pdf
7.	SIMD instruction support.	1. the_isa.pdf 2. the_microarchitecture.pdf
8.	The memory addresses of the memory-mapped devices of the CPU (e.g., keyboard).	1. the_isa.pdf 2. the_microarchitecture.pdf
9.	The number of non-programmable registers in the CPU.	1. the_isa.pdf 2. the_microarchitecture.pdf
10.	The replacement policy of the L1 data cache.	1. the_isa.pdf 2. the_microarchitecture.pdf
11.	The memory controller's scheduling algorithm.	1. the_isa.pdf 2. the_microarchitecture.pdf
12.	The number of bits required for the destination register of a load instruction.	1. the_isa.pdf 2. the_microarchitecture.pdf
13.	Description of the support for division and multiplication between integers.	1. the_isa.pdf 2. the_microarchitecture.pdf
14.	The mechanism to enter in a system call in the OS.	1. the_isa.pdf 2. the_microarchitecture.pdf
15.	The size of the addressable memory.	1. the_isa.pdf 2. the_microarchitecture.pdf

Performance Evaluation (Q5)

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

(a) What is the CPI of application *A* when executing on processor *P1*? Show your work.

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

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

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

Pipelining (Reverse Engineering) (Q6)

Algorithm 1 contains a piece of assembly code. Table 2 presents the execution timeline of this code.

```
1     MOVI R1, X          # R1 <- X
2     MOVI R2, Y          # R2 <- Y
3  L1:
4     ADD  R1, R1, R2     # R1 <- R1 + R2
5     MUL  R4, R2, R3     # R4 <- R2 x R3
6     SUBI R3, R1, 100    # R3 <- R1 - 100, set condition flags
7     JZ   L1             # Jump to L1 if zero flag is set
8     MUL  R1, R1, R2     # R1 <- R1 x R2
9     MUL  R2, R3, R4     # R2 <- R3 x R4
10    ADD  R5, R6, R7     # R5 <- R6 + R7
```

Algorithm 1: Assembly Program

Dyn. Instr. Number	Instructions	Cycles 1	2	3	4	5	6	7	8	9	10	11	12	13	...
1	MOV R1, X	F	D	E1	E2	E3	M	W							
2	MOV R2, Y		F	D	E1	E2	E3	M	W						
3	ADD R1, R1, R2			F	D	-	-	E1	E2	E3	M	W			
4	MUL R4, R2, R3				F	-	-	D	E1	E2	E3	M	W		
5	SUBI R3, R1, 100							F	D	-	E1	E2	E3	M	...
6	JZ L1								F	-	D	-	-	E1	...
7	...														

Table 2: Execution timeline (F:Fetch, D:Decode, E:Execute, M:Memory, W:WriteBack)

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

(a) List the necessary data forwardings between pipeline stages to exhibit this behavior.

(b) Does this machine use hardware-interlocking or software-interlocking? Explain.

(c) Consider another machine that uses the opposite of your choice in the previous question. (e.g., if your answer is software-interlocking for the previous question, consider another machine using hardware-interlocking, or vice-versa). How would the execution timeline shown in Table 2 change? What would be different? Fill the following table and explain your reasoning below. *(Notice that the table below consists of two parts: the first seven cycles at the top, and the next seven cycles at the bottom.)*

For the rest of this question, assume the following:

- X = Y = 1 in Algorithm 1.
- Branch conditions are resolved at the stage E1.
- Branch predictor is static and predicts "always taken".
- The machine uses hardware-interlocking.

At a given clock cycle T,

- the value stored in R1 is 98.
- the processor fetches the dynamic instruction N which is ADD R1, R1, R2

Dyn. Instr. Number	Instructions	Cycles 1	2	3	4	5	6	7

		8	9	10	11	12	13	14

(d) Calculate the value of T. Show your work.

(e) Calculate the value of N. Show your work.

Tomasulo's Algorithm (Q7)

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

- The processor has four main pipeline stages: Fetch (F), Decode (D), Execute (E), and Write-back (W).
- The processor implements a single-level data cache.
- The processor has the following *two types of execution units* but it is unknown how many of each type the processor has.
  - **Integer ALU:** Executes integer instructions (i.e., addition, multiplication, move, branch).
  - **Memory Unit:** Executes load/store instructions.
- The processor is connected to a main memory that has a fixed access latency.
- Load/store instructions spend cycles in the E stage exclusively for accessing the data cache or the main memory.
- There are two reservation stations, one for each execution unit type.

The reservation stations are all initially empty. The processor executes an arbitrary program. From the beginning of the program until the program execution finishes, seven dynamic instructions enter the processor pipeline. Table 3 shows the seven instructions and their execution diagram.

Instruction semantics:

- MV R0 ← #0x1000: moves the hexademical number 0x1000 to register R0.
- LD R1 ← [R0]: loads the value stored at memory address R0 to register R1.
- BL R1, #100, #LB1: a branch instruction that conditionally takes the path specified by label "#LB1" if the content of register R1 is smaller than integer value 100.
- MUL R1 ← R1, #5: multiplies R1 and 5 and writes the result to R1.
- ST [R0] ← R1: stores R1 to memory address specified by R0.
- ADD R1 ← R1, R0: adds R1 and R0 and writes the result to R1.

Instruction/Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1: MV R0 ← #0x1000	F	D	E1	E2	E3	E4	W																			
2: LD R1 ← [R0]		F	D	-	-	-	E1	E2	E3	E4	E5	E6	E7	E8	W											
3: BL R1 #100, #LB1						F	D	-	-	-	-	-	-	-	E1	E2	E3	E4	W							
4: MUL R1 ← R1, #5														F	D	E1	E2	E3	//squashed (i.e., killed)							
5: ST [R0] ← R1															F	D	-	-	//squashed (i.e., killed)							
6: ADD R1 ← R1, R0																			F	D	E1	E2	E3	E4	W	
7: ST [R0] ← R1																				F	D	-	-	-	E1	W

Table 3: Execution diagram of the seven instructions.

(a) Using the information provided above, answer the following questions regarding the processor design. If a question has more than one correct answer or a correct answer cannot be determined using the information provided in the question, answer the question as specifically as possible.

For example, use phrases such as "at least/at most" and try to narrow down the answer using the information that is provided in the question and can be inferred from Table 3. If nothing can be inferred, write "Unknown" as an answer. Explain your reasoning briefly.

What is the cache hit latency? What is the cache miss latency? What is the cache line size? What is the number of entries in each reservation station (R)? How many ALUs does the processor have? Is the integer ALU pipelined? Does the processor perform branch prediction? At which pipeline stage is the correct outcome of a branch evaluated?

(b) What is the program (i.e., static instructions) that leads to the execution diagram shown in Table 3? Fill in the blanks below with the known instructions of the program and also (if applicable) show where and how many unknown instructions there are in the program.

Program:

GPUs and SIMD

(Q8)

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

The following code segment is run on a GPU. A warp in the GPU consists of 64 threads, and there are 64 SIMD lanes in the GPU. Each thread executes a single iteration of the shown loop. Assume that the data values of the arrays A and B are already in vector registers so there are no loads and stores in this program. Both A and B are arrays of integers. (Hint Notice that there are 6 instructions in each thread.)

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

Please answer the following four questions.

(a) How many warps does it take to execute this program?

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

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

(d) What is the minimum possible SIMD utilization of this program?

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

Caches

(Q9)

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 size (4 or 8 KiB).
- Cache replacement policy (LRU or FIFO).

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

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

Assume that the cache is initially empty at the beginning of the first sequence, but not at the beginning of the second 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. If a characteristic cannot be known, then write "Unknown" and explain.

(a) Cache block size (8, 16, 32, 64, or 128 B)?
(b) Cache associativity (1-, 2-, 4-, or 8-way)?
(c) Cache replacement policy (LRU or FIFO)?
(d) To identify the cache size, you execute the following sequence right after sequence 2 (i.e., the contents are the same as at the end of the second sequence) and measure the cache hit rate:

Addresses Accessed (Oldest → Youngest): 8192 → X → Y

Which addresses should you use for X and Y?

Branch Prediction (Q10)

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

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

(b) In a new, higher-performance version of the previous processor, the architects implement a mysterious branch prediction mechanism to improve the performance of the processor. They keep the rest of the design exactly the same as before. The new design with the mysterious branch predictor completes the execution of the following piece of code in 136 cycles. Please note that the number of pipeline stages and the stage at which the branch condition is evaluated are same as the previous question. Also, assume that the pipeline never stalls due to any other reasons than conditional branches.

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

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

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

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

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

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

(I) **Static Branch Predictor**

Could this be the mysterious branch predictor? YES/NO

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

i. Static Prediction Direction Always taken/Always not taken

Explain clearly to receive points

(II) **Last Time Branch Predictor**

Could this be the mysterious branch predictor? YES/NO

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

i. Initial Prediction Direction Taken Not taken

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

Explain clearly to receive points.

(III) **Backward taken, Forward not taken (BTFN)**

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

Could this be the mysterious branch predictor? YES/NO

Explain clearly to receive points.

(IV) **Two-bit Counter Based Prediction (using saturating arithmetic)**

Could this be the mysterious branch predictor? YES/NO

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

i. Initial Prediction Direction 00 (Strongly not taken) 01 (Weakly not taken) 10 (Weakly taken) 11 (Strongly taken)

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

Explain clearly to receive points.

BONUS: VLIW

(Q11)

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

MEMORY	INTEGER	CONTROL	FLOAT

Table 4: VLIW instruction format.

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

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

Table 5: Instruction latencies and descriptions.

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

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

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

(a) Warm-up: which of the following are goals of VLIW CPU design (circle all that apply)?
(i) Simplify code compilation.
(ii) Simplify application development.
(iii) Reduce overall hardware complexity.
(iv) Simplify hardware inter-instruction dependence checking.
(v) Reduce processor fetch width.

(b) Your task is to determine the optimal VLIW scheduling of the short instructions by hand.
Fill in the following table with the highest performance (i.e., fewest number of execution cycles) instruction sequence that may be directly input into the VLIW CPU and have the same functionality as the code in Table 6. Where possible, you may write instruction IDs corresponding to the numbers given in Table 6 and leave any NOP instructions as blank slots.
Consider **only one loop iteration** (including the BEQ instruction), **ignore initialization** and any cross-iteration optimizations (e.g., loop unrolling), and **do not** optimize the code by removing or changing existing instructions.

Cycle	MEMORY	INTEGER	CONTROL	FLOAT
1				
2				
3				

Hint: you should not require more than 20 cycles.

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

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

Digital Design & Computer Arch.

Problem Solving IV

Prof. Onur Mutlu

ETH Zürich

Spring 2023

12 July 2023