CMPU-224 Exam 2 Solutions
Spring 2026

Name: _____

This is a closed book, closed notes quiz. No electronic devices are allowed. You have 75 minutes to complete the quiz. There are a total of 18 questions and 100 points. This quiz has 14 pages. Make sure you have all 14 pages before starting the quiz.

There should be enough space on the quiz for your answers. If you need more space to work out a problem, blank paper will be available, just ask.

Good Luck!

1. (8 points) You are given the following truth table with three inputs **A**, **B**, **C** and a single output **X**.

A	**B**	**C**	**X**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

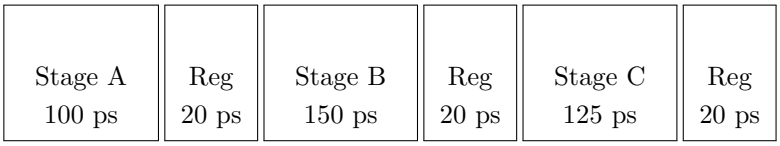
Write the Boolean equation for **X** using the **Sum of Products (SOP)** form and **do not minimize** the boolean equation. Use `~` for `NOT`, `&` for `AND`, and `|` for `OR`.

$X =$ ~A & ~B & C | A & ~B & C | A & B & ~C

Solution: Identify every row of the truth table where $X = 1$ and write the corresponding product (minterm) term. Then OR all the terms together.

$$X = \sim A \,\&\, \sim B \,\&\, C \mid A \,\&\, \sim B \,\&\, C \mid A \,\&\, B \,\&\, \sim C$$

2. (8 points) Consider the following three-stage pipeline. The propagation delay of each stage is shown in the diagram below. Each pipeline register takes 20 ps to latch its value.



What is the shortest possible clock cycle period (in ps) for this pipeline? Be sure to count the propagation delay and the time to latch the data into the register.

Minimum clock period: **170** ps

What is the total latency of this pipeline assuming the shortest possible clock cycle? I.e., what is the total time for a single instruction to pass through all three stages and be latched into the final pipeline register?

Total latency: **510** ps

Solution: In a pipeline, every stage is clocked at the same rate, determined by the **slowest** stage. The slowest stage is Stage B at 150 ps, so each stage requires at least 150 ps per clock cycle plus 20 ps to latch the result. This gives us a minimum period of 170 ps. With 3 stages at 150 ps each and 3 pipeline registers at 20 ps each:

$$\text{Latency} = 3 \times 150 + 3 \times 20 = 450 + 60 = \mathbf{510\ ps}$$

3. (16 points) Consider a memory system with the following properties:

   - Memory address are **24**-bits long.
   - The cache is **4**-way Set Associative.
   - The cache has **8** sets.
   - The cache block size is **16** bytes.

   The CPU requests the byte at memory address **0x75C813**.

   Parse the above address address into its Tag, Set Index, and Block Offset fields. Give your numeric answers in **hexadecimal**.

   Tag: **0x0EB90**

   Set index: **0x1**

   Block offset: **0x3**

   Byte Value at address or (N/A) if there is a cache miss: **0xE6**

   The table below shows the complete contents of the cache (Valid bits, Tags, and Data). In the Data Block, the first (leftmost) column is data byte 0, and the last (rightmost) colum is data byte 15.

Set	Line	V	Tag	Data (Bytes 0 – 15 in Hex)
0	0	1	03C0C	57 A4 33 8A AF 2F 83 8A 3E D2 B2 0E 04 87 5F F3
0	1	1	0EB90	17 73 B2 ED 07 D8 F5 DB 19 93 EE 6A F3 A4 85 42
0	2	1	15317	FB 3E 9B C5 90 3F 2E 02 6C D5 5C 1A 68 EE 52 B8
0	3	1	1B044	67 3C FB 57 FA E6 A6 B1 C5 FD 57 5E EC 59 9F E6
1	0	1	0D232	FB CA 2D A2 95 0E 20 27 7B B5 C4 BE 58 28 41 B9
1	1	1	0EB90	3E B8 2B E6 7E 24 06 5E B2 F2 33 F7 16 30 82 41
1	2	1	0DF42	AF 59 C5 7B 59 4A BE B8 C7 62 58 39 81 D4 1D 5A
1	3	1	0132B	AE 6B DD AE 50 BE 50 15 A4 33 6C 76 0E 52 66 4C
2	0	1	09359	B6 F3 7E F3 1E 99 BC 79 5A 55 53 BC D6 A0 3C 81
2	1	1	0EB90	EC CD 18 5A 61 4A 9F FC D4 F8 0E 04 86 B1 5F 5F
2	2	1	11790	69 48 0A D2 F2 E5 E5 06 62 E1 0D 96 A5 D8 B2 92
2	3	1	1AE88	3C B6 A8 AC E4 B8 81 2D 19 CF A7 B2 39 1B 90 8F
3	0	1	1CA53	4C 97 F2 AE 0C CF 6A 8C 21 42 04 7F 2A 8C 82 53
3	1	1	0358A	B7 AC 75 9D D6 20 36 70 2E D6 34 7B CF 5F FE 66
3	2	1	01FA6	79 33 46 8B 97 0F 4B 4E 11 5E EB F4 72 15 D0 9A
3	3	1	0D430	52 1E 76 6D 53 E5 A2 E1 AE 42 9C 8C 86 F2 59 06
4	0	1	07FFD	D1 0B 0F CE B1 E9 FF 93 55 6E 1B 82 CF F6 1E 75
4	1	1	1D591	E8 4E DA A1 6C 45 9B C8 7A FE 1A EB 71 8D 47 B3
4	2	1	0506E	44 92 BA 0C D4 BF 84 AB 0D C9 A8 6B CF 22 0B 71
4	3	1	0528E	7B 76 65 1C 0D BB EF A5 05 D0 6D C0 06 7A E6 03
5	0	1	15950	E0 A1 78 F2 FD 87 20 24 D1 C2 59 51 18 50 62 E3
5	1	1	170BC	E0 66 BC 41 3F 1C 71 EA BD 7D F6 EA A3 4F 65 16
5	2	1	1610F	73 48 C0 12 3C 31 34 9A E0 49 7F 13 EF 47 04 10
5	3	0	-	-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
6	0	1	1835F	11 80 35 8A BE 02 14 CE 12 44 D5 09 B5 72 28 40
6	1	1	05EF8	22 31 7B 06 2A EB C7 78 78 8B D2 AF 41 E6 E7 00
6	2	1	012BA	5D BD 94 4D 28 2D 55 7B D2 0A A8 37 6C 58 75 5C
6	3	0	-	-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
7	0	1	0BC92	04 56 BC 22 82 E2 DA 71 DD 72 FC EE 7C E5 01 DA
7	1	1	1F63F	9E 9E F3 8F DC FB DB 2A 8E 10 F7 AE 5B 94 35 02
7	2	0	-	-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
7	3	1	16C77	41 F3 A8 EE 12 82 DF D3 A6 90 41 B6 A1 55 AC 92

4. Consider the following RISC-V assembly program (left) and its disassembled machine code (right).

Assembly Program	**Disassembled Binary**

`.data`
`arr: .word 10, 20, 30, 40`
 	
`.text`
`main:`
`  addi a0, x0, 0`
`  la  t0, arr`
`  addi t1, t0, 12`
`loop:`
`  lw  t2, 0(t0)`
`  add  a0, a0, t2`
`  addi t0, t0, 4`
`  bne  t0, t1, loop`
 	
`  li  a7, 10  # exit`
`  ecall`
	`00000000 <main>:`
	`  0: 00000513  addi x10 x0 0`
	`  4: 10000297  auipc x5 0x10000`
	`  8: ffc28293  addi x5 x5 -4`
	`  c: 00c28313  addi x6 x5 12`
	
	`00000010 <loop>:`
	`10: 0002a383  lw x7 0 x5`
	`14: 00750533  add x10 x10 x7`
	`18: 00428293  addi x5 x5 4`
	`1c: fe629ae3  bne x5 x6 -12 <loop>`
	`20: 00a00893  addi x17 x0 10`
	`24: 00000073  ecall`

(a) (2 points) How many times is the `bne` instruction executed?

3

Solution: The loop runs 3 iterations (for array elements 10, 20, and 30). Each iteration executes the `bne` instruction once, so it is executed 3 times. On the third iteration `t0` equals `t1`, so the branch is not taken and the loop exits.

(b) (2 points) What is the value in the `a0` register when the program terminates?

60

Solution: `a0` starts at 0 and accumulates the first three elements of the array: $0 + 10 + 20 + 30 = 60$.

(c) (2 points) How many cycles does it take this program to run on the **single-cycle processor**? Note that the disassembled binary shows the instructions that the hardware executes. In particular, the pseudo-instruction `la t0, arr` expands to two instructions (`auipc` and `addi`).

18

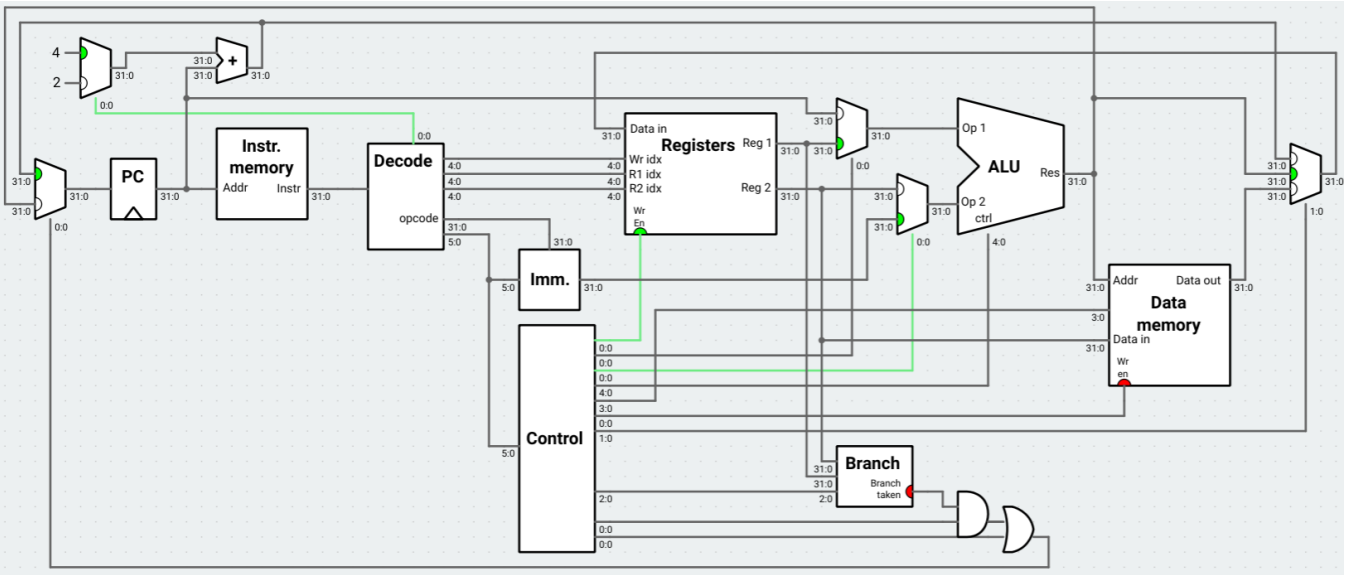
Solution: On a single-cycle processor, each instruction takes exactly one cycle.
Prologue: 4 instructions (`addi`, `auipc`, `addi`, `addi`).
Loop body: 4 instructions per iteration $\times$ 3 iterations = 12.
Epilogue: 2 instructions (`addi`, `ecall`).
Total: $4 + 12 + 2 = 18$ cycles.

(d) (2 points) How many cycles does it take this program to run on the **5-stage pipelined processor**?

31

Solution: Start with the 18 instructions retired. Then add pipeline effects:
Pipeline drain: +4 cycles (to flush the remaining stages after the last instruction).
`ecall`: +2 cycles (treated as a special stall).
Load-use hazards: The `lw`/`add` pair causes a 1-cycle bubble each iteration: +3 cycles (3 iterations).
Branch mispredictions: `bne` is taken in iterations 1 and 2 (mispredicted assuming not-taken): +4 cycles (2 mispredictions $\times$ 2 cycles each).
Total: $18 + 4 + 2 + 3 + 4 = 31$ cycles.

5. (4 points) Consider the single-cycle processor datapath shown below. The dark circle in the muxes show which input of the mux is active. The signal "`Wr En`" in the Register File is set to `High`, the signal "`Branch taken`" is set to `Low`, and the signal"`Wr En`" for `Data memory` is set to `Low`.



Which instruction generated the state shown in the diagram above?

- ◯ `beq x0, x0, exit`
- ◯ `xor a0, a0, a0`
- √ **`addi t0, t0, 32`**
- ◯ `lw t0, 0(a0)`

Solution: The correct answer is `addi t0, t0, 32`.

- `beq x0, x0, exit` is incorrect because the branch-taken enable signal is not active in the diagram.
- `xor a0, a0, a0` is incorrect because the ALU's Op 2 input is getting its value from the Immediate generator, not from a register — this indicates an I-type instruction, not an R-type.
- `lw t0, 0(a0)` is incorrect because the Write Enable signal in the data memory stage is not active, and the data memory read path is not selected.

This leaves `addi t0, t0, 32`, an I-type instruction that reads a register for Op 1, uses an immediate for Op 2, computes through the ALU, and writes the result back to the register file — exactly matching the active signals shown.

6. (8 points) You are given the following 32-bit RISC-V machine instruction in hexadecimal:

0x0062F533

Decode this instruction by filling in the fields below, then give the full assembly instruction.

Field	**Value**
`opcode` (binary)	`0110011`
Instruction format (R, I, S, B)	`R`
`rd` (ABI register name or N/A)	`x10 (a0)`
`funct3` (binary)	`111`
`rs1` (ABI register name)	`x5 (t0)`
`rs2` (ABI register name or N/A)	`x6 (t1)`
`funct7` (binary or N/A)	`0000000`

Assembly instruction: `and a0, t0, t1`

Solution: First, convert to binary:

`0000 0000 0110 0010 1111 0101 0011 0011`

The opcode is `0110011` (bits 6–0), which identifies this as an R-type arithmetic/logic instruction. Extracting the remaining fields: `rd` = `01010` (x10/a0), `funct3` = `111`, `rs1` = `00101` (x5/t0), `rs2` = `00110` (x6/t1), `funct7` = `0000000`. With opcode `0110011`, `funct3` = `111`, and `funct7` = `0000000`, the instruction is `AND`.

The full assembly instruction is: `and a0, t0, t1`.

7. (4 points) Which of the following best describes a **combinational circuit**?

◯ A circuit whose outputs depend on both current and past inputs

√ **A circuit whose outputs depend only on the current inputs**

◯ A circuit that requires a clock signal to operate

◯ A circuit that contains at least one feedback loop

Solution: Combinational circuits are acyclic networks of logic gates where the output is a function of only the current inputs. Sequential circuits have memory and depend on past inputs.

8. (4 points) In a ripple-carry adder, what determines the **critical path** delay?

◯ The delay through a single full adder

◯ The delay through the final XOR gate

√ **The propagation of the carry chain from the least significant to the most significant bit**

◯ The fan-out of the input signals

Solution: In a ripple-carry adder, each full adder must wait for the carry-out of the previous stage. The worst-case delay is the carry propagating through all n stages from LSB to MSB.

9. (4 points) In the register file of a RISC-V processor, reading a register is a **combinational** operation while writing is a **synchronous** operation. What does this mean?

   ◯ Reads happen on the falling edge; writes happen on the rising edge

   √ **Reads produce output after a propagation delay without waiting for a clock edge; writes update the register only on the rising clock edge**

   ◯ Both reads and writes require a clock edge, but reads are given higher priority

   ◯ Reads use DRAM and writes use SRAM

   Solution: Reading is combinational: the register address propagates through a multiplexor and the data appears at the output after a gate delay. Writing is synchronous: the new value is latched into the selected register only on the rising clock edge when the write-enable signal is asserted.

10. (4 points) In the RISC-V single-cycle processor, which instruction determines the minimum clock period?

    ◯ `add` — because it uses the ALU

    ◯ `beq` — because it must compute the branch target

    √ **`lw` — because it uses all five datapath stages**

    ◯ `jal` — because it must update both the PC and a register

    Solution: The `lw` instruction has the longest critical path: instruction memory → register read → ALU (address calculation) → data memory read → register write. The clock period must be long enough for this entire path.

11. (4 points) In RISC-V, what is the purpose of the `lui` instruction?

    ◯ It loads a byte from memory into a register

    ◯ It performs an unsigned comparison between two registers

    √ **It places a 20-bit immediate value into the upper 20 bits of a register and zeros the lower 12 bits**

    ◯ It shifts a register left by an unsigned immediate amount

    Solution: `lui` (Load Upper Immediate) is a U-type instruction that writes its 20-bit immediate into bits 31–12 of the destination register and sets bits 11–0 to zero. It is commonly paired with `addi` to construct full 32-bit constants.

12. (4 points) In a 5-stage pipelined RISC-V processor, a **load-use hazard** occurs when:

    ◯ A store instruction writes to the same address a load is reading

    ◯ Two consecutive instructions write to the same register

    √ **An instruction in the Decode stage needs a value that is currently being loaded from memory by the instruction in the Execute stage**

    ◯ A branch instruction immediately follows a load instruction

Solution: A load-use hazard is a specific data hazard where a `lw` instruction's result is not available until the end of the Memory stage, but the very next instruction needs it during the Decode/Execute stage. This cannot be resolved by forwarding alone and requires a one-cycle stall (bubble).

13. (4 points) **Forwarding** (also called bypassing) in a pipelined processor resolves most data hazards by:

    ◯ Reordering instructions at compile time to avoid conflicts

    ◯ Stalling the pipeline until the result is written to the register file

    ◯ Predicting the result of the computation before it completes

    √ **Routing the result directly from a later pipeline stage back to an earlier stage where it is needed, without waiting for the writeback stage**

Solution: Forwarding passes the computed result from the EX/MEM or MEM/WB pipeline registers directly to the input of the ALU or Decode stage, eliminating the need to wait for the value to be written back to the register file.

14. (4 points) When a branch is **mispredicted** in a 5-stage pipeline with a predict-not-taken policy, how many cycles are wasted?

    ◯ 1 cycle

    √ **2 cycles**

    ◯ 3 cycles

    ◯ 5 cycles

Solution: With predict-not-taken, the processor fetches the two instructions sequentially after the branch. If the branch is actually taken (resolved in the Execute stage), those two instructions in the Fetch and Decode stages must be squashed (replaced with bubbles), wasting 2 cycles.

15. (4 points) In a 5-stage pipeline, the ideal CPI (cycles per instruction) is 1.0. What is the actual CPI if a program executes 200 instructions and incurs 40 bubble cycles due to hazards?

◯ 0.80

◯ 1.00

√ **1.20**

◯ 1.40

Solution: CPI $= 1.0 + B/I = 1.0 + 40/200 = 1.20$.

16. (4 points) Which of the following correctly describes the difference between SRAM and DRAM?

◯ SRAM uses capacitors; DRAM uses flip-flops

◯ SRAM is cheaper per bit and used for main memory

◯ DRAM is faster and does not need refreshing

√ **SRAM uses 6 transistors per cell and is faster; DRAM uses 1 transistor and 1 capacitor per cell, is denser, but requires periodic refresh**

Solution: SRAM stores each bit in a stable circuit of 6 transistors (no refresh needed). DRAM stores each bit as charge on a tiny capacitor (1 transistor + 1 capacitor), making it much denser and cheaper, but the capacitor leaks and must be periodically refreshed.

17. (4 points) The principle of **temporal locality** states that:

√ **If a memory location is accessed, it is likely to be accessed again in the near future**

◯ If a memory location is accessed, nearby locations are likely to be accessed soon

◯ Memory accesses are uniformly distributed across the address space

◯ Programs tend to access memory in sequential order only

Solution: Temporal locality means recently accessed data is likely to be accessed again soon (e.g., loop variables, frequently called functions). Spatial locality (nearby addresses accessed soon) is the other key form of locality.

18. (4 points) A program accesses a two-dimensional array declared as `int A[1024][1024]` using the following loop:

```
for (int j = 0; j < 1024; j++)
    for (int i = 0; i < 1024; i++)
        sum += A[i][j];
```

Why does running this code result in poor performance?

◯ The array is too large to fit in main memory

◯ The inner loop variable `i` causes branch mispredictions

◯ The compiler cannot optimize the addition

√ **This code has poor spatial locality**

Solution: C stores arrays in row-major order. Accessing `A[i][j]` with `i` varying in the inner loop means each access jumps 1024 `int`s (4096 bytes) ahead in memory, likely landing in a different cache line each time. Swapping the loop order to iterate over `j` in the inner loop gives stride-1 access with excellent spatial locality.