ECE-6913 Computer System Architecture
Homework 5

New York University, Fall 2024

Due on 11/07, 11:59 PM

Name: Raman Kumar Jha
NYU ID: N13866145

Problem 1: In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: 0x00c6ba23.

Instruction word: 0x00c6ba23
First, we decode the instruction:

- **Binary:** `0000 0000 1100 0110 1011 1010 0010 0011`
- **Opcode (bits 6:0):** `0100011` (This is a `STORE` instruction, S-type)
- **funct3 (bits 14:12):** `011` (This specifies `sd`, Store Doubleword)
- **rs1 (bits 19:15):** `01101` (This is register `x13`)
- **rs2 (bits 24:20):** `01100` (This is register `x12`)
- **Immediate (imm):** Reconstructed from bits 31:25 and 11:7:
  - `imm[11:5]` = `0000000`
  - `imm[4:0]` = `10100`
  - **Immediate** = $000000010100_2 = 20_{10}$
- **Full Instruction:** `sd x12, 20(x13)`

1.1 What are the values of the ALU control unit's inputs for this instruction?

The ALU control unit takes two inputs: the 2-bit `ALUOp` signal (from the main Control unit) and the function bits from the instruction.

- The main Control unit sees the `STORE` opcode (`0100011`) and generates `ALUOp = 00` (which signifies an "add" operation for loads/stores).
- The function bits are `funct3 = 011`.

Answer: The inputs are `ALUOp = 00` and `funct3 = 011`.

1.2 What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

Answer: The new PC address is `PC + 4`.
Path: This is not a branch or jump.

1. The current `PC` value is read (at T=30ps).
2. This value enters an **Adder** with the constant **4**.
3. The result, `PC + 4`, goes to **Input 0** of the `PCSrc` Mux.
4. The `Branch` control signal is 0, so the `PCSrc` Mux selects Input 0.
5. The output of the Mux, `PC + 4`, is fed to the `PC` register's input, ready for the next clock edge.

1.3 For each mux, show the values of its inputs and outputs during the execution of this instruction.

- **Mux 1:** `ALUSrc` **(selects 2nd ALU operand)**
  - **Input 0:** `Reg[x12]` (Read Data 2 from Register File)
  - **Input 1:** `20` (from Sign Extend unit)
  - **Control Signal (`ALUSrc`):** 1
  - **Output:** `20`
- **Mux 2:** `MemtoReg` **(selects data for register write-back)**
  - **Input 0:** ALU Result (`Reg[x13] + 20`)
  - **Input 1:** Data from Data Memory (Not used)
  - **Control Signal (`MemtoReg`):** X (Don't Care, as `RegWrite`=0)
  - **Output:** X (Don't Care)
- **Mux 3:** `PCSrc` **(selects next PC)**
  - **Input 0:** `PC + 4` (from PC+4 Adder)
  - **Input 1:** Branch Target Address (from Branch Adder, path is unused)
  - **Control Signal (`PCSrc`):** 0 (since `Branch` signal is 0)
  - **Output:** `PC + 4`

1.4 What are the input values for the ALU and the two add units?

- **Main ALU:**
  - **Input 1:** `Reg[x13]` (from Register File, Read Data 1)
  - **Input 2:** `20` (from `ALUSrc` Mux)
- **Adder 1 (PC+4 Adder):**
  - **Input 1:** Current `PC` value
  - **Input 2:** `4`
- **Adder 2 (Branch Target Adder):**
  - (This path is not used, but the inputs would be `PC` and the sign-extended immediate)

1.5 What are the values of all inputs for the register's unit?

- **Read register 1:** `13` (for `x13`)
- **Read register 2:** `12` (for `x12`)
- **Write register:** `20` (from `imm[4:0]` field, but this is a "Don't Care" as `RegWrite` is 0)
- **Write data:** X (Don't Care, from `MemtoReg` Mux)
- **RegWrite (control signal):** `0` (A store does not write to the register file)

Problem 2: Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

2.1 What is the latency of an R-type instruction?

Path: PC Read → I-Mem → RegFile Read → ALUSrc Mux → ALU → MemtoReg Mux → RegFile Setup

- **ALU Input 1 stable:** 30 (PC Read) + 250 (I-Mem) + 150 (RegFile) = 430 ps
- **ALU Input 2 stable:** 30 (PC Read) + 250 (I-Mem) + 150 (RegFile) + 25 (ALUSrc Mux) = 455 ps
- **ALU Out stable:** 455 (ALU In 2) + 200 (ALU) = 655 ps
- **Write Data stable:** 655 (ALU Out) + 25 (MemtoReg Mux) = 680 ps
- **Total Period:** 680 (Write Data) + 20 (RegFile Setup) = 700 ps

Answer: 700 ps

2.2 What is the latency of ld?

Path: PC Read → I-Mem → RegFile Read (for `rs1`) → ALUSrc Mux → ALU (address) → D-Mem → MemtoReg Mux → RegFile Setup

- **ALU Input 1 stable:** 30 (PC Read) + 250 (I-Mem) + 150 (RegFile) = 430 ps
- **ALU Input 2 stable:** 30 (PC Read) + 250 (I-Mem) + 50 (Sign Ext) + 25 (ALUSrc Mux) = 355 ps
- **ALU Out (Address) stable:** max(430, 355) + 200 (ALU) = 630 ps
- **D-Mem Out stable:** 630 (Address) + 250 (D-Mem) = 880 ps
- **Write Data stable:** 880 (D-Mem Out) + 25 (MemtoReg Mux) = 905 ps
- **Total Period:** 905 (Write Data) + 20 (RegFile Setup) = 925 ps

Correction: The `ALUSrc` Mux needs `Reg[rs1]` as an input for the address calculation (`ld x1, 20(x2)` → `x2` is `rs1`). This `Reg[rs1]` data (stable at 430ps) goes *directly* to the ALU, not through the `ALUSrc` Mux. *Recalculating ld:*

- **ALU Input 1 (Reg[rs1]) stable:** 30 (PC) + 250 (IMem) + 150 (RegFile) = 430 ps

- **ALU Input 2 (Imm) stable:** 30 (PC) + 250 (IMem) + 50 (SignExt) = 330 ps. Path via Mux: max(330, 330(control)) + 25 (Mux) = 355 ps.
- **ALU Out (Address) stable:** max(430, 355) + 200 (ALU) = 630 ps
- **D-Mem Out stable:** 630 (Address) + 250 (D-Mem) = 880 ps
- **Write Data stable:** 880 (D-Mem Out) + 25 (MemtoReg Mux) = 905 ps
- **Total Period:** 905 (Write Data) + 20 (RegFile Setup) = 925 ps

Answer: 925 ps

2.3 What is the latency of sd?

Path: The `sd` instruction must calculate the address and fetch the data to be written. The cycle must be long enough for the D-Mem *inputs* to be stable. It does not write back to the RegFile.

- **Path to D-Mem Address:** 30 (PC) + 250 (IMem) + 150 (RegFile rs1) + 200 (ALU) = 630 ps (path is Reg[rs1] -¿ ALU In1, Imm -¿ Mux -¿ ALU In2) *Let's trace properly:*
- **ALU In 1 (Reg[rs1]) stable:** 30+250+150 = 430 ps
- **ALU In 2 (Imm) stable:** 30+250+50(SignExt) + 25(Mux) = 355 ps
- **D-Mem Address stable:** max(430, 355) + 200 (ALU) = 630 ps
- **Path to D-Mem Write Data (Reg[rs2]):** 30 (PC) + 250 (IMem) + 150 (RegFile rs2) = 430 ps
- **Path to Next PC:** (from 2.4/2.6) 375 ps

The longest path that must complete for an `sd` is the D-Mem Address calculation. **Answer: 630 ps**

2.4 What is the latency of beq?

Path: This path determines the `PCSrc` Mux control signal, which depends on the ALU `Zero` output.

- **ALU Zero Flag stable:** 30 (PC) + 250 (IMem) + 150 (RegFile) + 25 (ALUSrc Mux) + 200 (ALU) = 655 ps. (This is the R-type ALU path: 455 (ALU In) + 200 (ALU) = 655 ps)
- **Branch Signal stable:** 30 (PC) + 250 (IMem) + 50 (Control) = 330 ps
- **PCSrc Control stable:** max(655, 330) + 5 (Single Gate) = 660 ps
- **PC+4 Add stable:** 30 (PC) + 150 (Adder) = 180 ps
- **Branch Target stable:** 30 (PC) + 250 (IMem) + 50 (SignExt) = 330. Then 330 + 150 (Adder) = 480 ps.
- **PCSrc Mux Out stable:** max(180, 480, 660) + 25 (Mux) = 685 ps
- **Total Period:** 685 (PC In) + 20 (PC Setup) = 705 ps

Answer: 705 ps

2.5 What is the latency of an I-type instruction?

Path: PC Read → I-Mem → RegFile Read (`rs1`) → ALUSrc Mux → ALU → MemtoReg Mux → RegFile Setup

- **ALU Input 1 (Reg[rs1]) stable:** 30 (PC) + 250 (IMem) + 150 (RegFile) = 430 ps
- **ALU Input 2 (Imm) stable:** 30 (PC) + 250 (IMem) + 50 (Sign Ext) + 25 (ALUSrc Mux) = 355 ps
- **ALU Out stable:** max(430, 355) + 200 (ALU) = 630 ps
- **Write Data stable:** 630 (ALU Out) + 25 (MemtoReg Mux) = 655 ps
- **Total Period:** 655 (Write Data) + 20 (RegFile Setup) = 675 ps

Answer: 675 ps

2.6 What is the minimum clock period for this CPU?

The minimum clock period is the latency of the longest possible instruction path. We compare all the latencies calculated:

- R-type: 700 ps
- **ld: 925 ps (Longest)**
- sd: 630 ps
- beq: 705 ps
- I-type: 675 ps

The critical path is the `ld` instruction. **Answer: 925 ps**

Problem 3

(a) Speedup of a variable-cycle clock

R-type/I-type (non-ld)	ld	sd	beq
52%	25%	11%	12%

This problem uses the instruction latencies calculated in Problem 2 and the instruction mix provided.

- **Single-Cycle (Old) CPU:** The clock period is set by the longest instruction, the `ld`. $T_{\text{old}} = 925$ ps.
- **Variable-Cycle (New) CPU:** The average time per instruction is the weighted average of each instruction's latency.
  - R-type/I-type (non-ld) latency = 700 ps (from R-type)
  - `ld` latency = 925 ps
  - `sd` latency = 630 ps
  - `beq` latency = 705 ps

  $T_{\text{new}} = (0.52 \times 700 \text{ ps}) + (0.25 \times 925 \text{ ps}) + (0.11 \times 630 \text{ ps}) + (0.12 \times 705 \text{ ps})$
  $T_{\text{new}} = 364 \text{ ps} + 231.25 \text{ ps} + 69.3 \text{ ps} + 84.45 \text{ ps}$
  $T_{\text{new}} = 749 \text{ ps}$
- **Speedup:** Speedup $= \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{925 \text{ ps}}{749 \text{ ps}} \approx 1.235$

Answer: The speedup of the variable-cycle CPU is **1.235**.

(b) Adding a multiplier

3.b1 What is the clock cycle time with and without this improvement?

- **Without Improvement:** The clock time is the original single-cycle period from Problem 2, set by the `ld` instruction. $T_{\text{without}} = 925$ ps.
- **With Improvement:** The ALU latency increases from 200 ps to 500 ps (an increase of 300 ps). We must re-calculate the latency of all instructions that use the ALU.
  - **New R-type:** 700 ps (old) + 300 ps (ALU Δ) = **1000 ps**
  - **New ld:** 925 ps (old) + 300 ps (ALU Δ) = **1225 ps** (New Critical Path)
  - **New sd:** 630 ps (old) + 300 ps (ALU Δ) = **930 ps**
  - **New beq:** 705 ps (old) + 300 ps (ALU Δ) = **1005 ps**
  - **New I-type:** 675 ps (old) + 300 ps (ALU Δ) = **975 ps**

  The new clock period is set by the new longest instruction, `ld`. $T_{\text{with}} = 1225$ ps.

Answer: The clock cycle time is **925 ps** without the improvement and **1225 ps** with it.

3.b2 What is the speedup achieved by adding this improvement?

Speedup $= \frac{\text{CPU Time}_{\text{old}}}{\text{CPU Time}_{\text{new}}} = \frac{IC_{\text{old}} \times \text{CPI}_{\text{old}} \times T_{\text{old}}}{IC_{\text{new}} \times \text{CPI}_{\text{new}} \times T_{\text{new}}}$

- $IC_{\text{new}} = 0.95 \times IC_{\text{old}}$ (Instructions reduced by 5%)
- CPI = 1 for both (single-cycle)

Speedup $= \frac{IC_{\text{old}} \times 1 \times 925 \text{ ps}}{(0.95 \times IC_{\text{old}}) \times 1 \times 1225 \text{ ps}}$
Speedup $= \frac{925}{0.95 \times 1225} = \frac{925}{1163.75} \approx 0.795$
Answer: The speedup is **0.795**. This is a slowdown.

3.b3 What is the slowest the new ALU can be and still result in improved performance?

For performance to improve, Speedup > 1.

$$\frac{\text{CPU Time}_{\text{old}}}{\text{CPU Time}_{\text{new}}} > 1 \implies \text{CPU Time}_{\text{old}} > \text{CPU Time}_{\text{new}}$$

$$IC_{\text{old}} \times 925 \text{ ps} > (0.95 \times IC_{\text{old}}) \times T_{\text{new\_clock}}$$

$$925 > 0.95 \times T_{\text{new\_clock}}$$

$$T_{\text{new\_clock}} < \frac{925}{0.95} \implies T_{\text{new\_clock}} < 973.68 \text{ ps}$$

The clock period must be **973 ps** or less.

The new clock period is set by the `ld` instruction. Let Δ be the *additional* time added to the ALU. $T_{\text{new\_clock}} = T_{\text{ld\_old}} + \Delta = 925 + \Delta$

$$925 + \Delta \leq 973$$

$$\Delta \leq 48 \text{ ps}$$

The new ALU latency is $L_{\text{ALU\_new}} = L_{\text{ALU\_old}} + \Delta = 200 \text{ ps} + \Delta$. $L_{\text{ALU\_new}} \leq 200 + 48 \text{ ps} = 248 \text{ ps}$
Answer: The slowest the new ALU can be is **248 ps**.

I-Mem	Register File	Mux	ALU	Adder	D-Mem	Single Register	Sign extend	Single gate	Control
1000	200	10	100	30	2000	5	100	1	500

(c) Adding more registers

3.c1 What is the speedup achieved by adding this improvement?

1. **Old CPU Time:** $T_{\text{old}} = 925$ ps. CPU Time$_{\text{old}} = IC_{\text{old}} \times 1 \times 925$ ps.
2. **New Clock Period:** RegFile latency increases from 150 ps to 160 ps (a Δ of 10 ps). We must re-calculate the latency of all instructions that use the RegFile.
   - **New R-type:** 700 ps (old) + 10 ps (RegFile Δ) = **710 ps**
   - **New ld:** 925 ps (old) + 10 ps (RegFile Δ) = **935 ps** (New Critical Path)
   - **New sd:** 630 ps (old) + 10 ps (RegFile Δ) = **640 ps**
   - **New beq:** 705 ps (old) + 10 ps (RegFile Δ) = **715 ps**
   - **New I-type:** 675 ps (old) + 10 ps (RegFile Δ) = **685 ps**

   $T_{\text{new\_clock}} = 935$ ps.
3. **New Instruction Count:** The number of `ld` and `sd` instructions is reduced by 12%.
   - Original Mix: 52% (R/I), 25% (ld), 11% (sd), 12% (beq)
   - New `ld` mix: $0.25 \times (1 - 0.12) = 0.22$
   - New `sd` mix: $0.11 \times (1 - 0.12) = 0.0968$
   - New total instruction fraction: 0.52 (R/I) + 0.12 (beq) + 0.22 (ld) + 0.0968 (sd) = 0.9568

   $IC_{\text{new}} = 0.9568 \times IC_{\text{old}}$.
4. **Calculate Speedup:** Speedup $= \frac{IC_{\text{old}} \times 925 \text{ ps}}{(0.9568 \times IC_{\text{old}}) \times 935 \text{ ps}}$
   Speedup $= \frac{925}{0.9568 \times 935} = \frac{925}{894.418} \approx 1.0342$

Answer: The speedup achieved is **1.034**.

3.c2 Compare the change in performance to the change in cost.

- **Performance Change:** A speedup of 1.034 represents a **3.4% performance increase**.
- **Cost Change:**
  - Old Cost (from image 'q3c.png'): 1000 (IMem) + 200 (RegFile) + 10 (Mux) + 100 (ALU) + 30 (Adder) + 2000 (DMem) + 5 (SingleReg) + 100 (SignExt) + 1 (Gate) + 500 (Control) = **3946**
  - New Cost: The cost of the Register File doubles from 200 to 400. New Cost = 3946 − 200 + 400 = **4146**
  - Percent Cost Increase: $\frac{4146-3946}{3946} = \frac{200}{3946} \approx 0.0507$

  Answer: The **5.07% increase in cost** is greater than the **3.4% increase in performance**.

3.c3 Given the cost/performance ratios you just calculated, describe a situation where it makes sense to add more registers and describe a situation where it doesn't make sense to add more registers.

- **When it makes sense:** It makes sense to add more registers in a market where performance is the primary goal and cost is a secondary concern. For example, in **high-performance computing (HPC)** or **high-end servers**, a 3.4% performance gain is valuable and worth the 5.1% cost increase, as it can reduce the time for complex scientific simulations or large data-center jobs.
- **When it doesn't make sense:** It does not make sense to add more registers in a **cost-sensitive or power-constrained market**. For **embedded systems** or **mobile devices** (like smartphones), a 5.1% cost increase for a 3.4% performance gain is a poor trade-off. The higher cost and likely higher power consumption of the larger register file would be unacceptable.

Problem 4

ld is the instruction with the longest latency on the CPU from Section 4.4 (in RISC-V text). If we modified ld and sd so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in rs1 before calling ld/sd), then no instruction would use both the ALU and Data memory. This would allow us to reduce the clock cycle time. However, it would also increase the number of instructions, because many ld and sd instructions would need to be replaced with ld/add or sd/add combinations.

4.1 What would the new clock cycle time be?

The original critical path was the `ld` instruction at **925 ps**, which included the path: `RegFile Read` → `ALU` → `D-Mem` → `RegFile Setup`.

In the new design, the ALU is no longer on the `ld` or `sd` paths. We must find the new longest path.

- **R-type:** 700 ps (from Problem 2)
- **beq:** 705 ps (from Problem 2)
- **New ld:** The path is now PC Read → I-Mem → RegFile Read (for `rs1` address) → D-Mem → MemtoReg Mux → RegFile Setup.
  - Latency = 30 (PC Read) + 250 (I-Mem) + 150 (RegFile) + 250 (D-Mem) + 25 (Mux) + 20 (RegFile Setup) = **725 ps**
- **New sd:** The path is PC Read → I-Mem → RegFile Read. The address (`rs1`) and data (`rs2`) must be stable at the D-Mem inputs.
  - Latency = 30 (PC Read) + 250 (I-Mem) + 150 (RegFile) = **430 ps**

Comparing the new latencies, the longest path is the modified `ld` instruction.

Answer: The new clock cycle time would be **725 ps**.

4.2 Would a program with the instruction mix presented in Problem 2 run faster or slower on this new CPU? By how much? (For simplicity, assume every ld and sd instruction is replaced with a sequence of two instructions.)

- **Old CPU Time:** $T_{\text{old}} = 925$ ps and $IC_{\text{old}} = I$. CPU $\text{Time}_{\text{old}} = I \times 1 \times 925 \text{ ps} = 925 \times I$
- **New CPU Time:** $T_{\text{new}} = 725$ ps. The instruction mix is 52% (R/I), 25% (ld), 11% (sd), 12% (beq). The `ld` and `sd` instructions (36% of the total) are all doubled. $IC_{\text{new}} = (0.52 + 0.12) \times I + 2 \times (0.25 + 0.11) \times I$
  $IC_{\text{new}} = (0.64) \times I + 2 \times (0.36) \times I = (0.64 + 0.72) \times I = 1.36 \times I$

  CPU $\text{Time}_{\text{new}} = IC_{\text{new}} \times T_{\text{new}} = (1.36 \times I) \times 725 \text{ ps} = 986 \times I$

Answer: The new CPU is **slower**. Speedup $= \frac{\text{CPU Time}_{\text{old}}}{\text{CPU Time}_{\text{new}}} = \frac{925 \times I}{986 \times I} \approx 0.938$. It is $986/925 \approx 1.066$, or **6.6% slower**.

4.3 What is the primary factor that influences whether a program will run faster or slower on the new CPU?

Answer: The primary factor is the **trade-off between the percentage decrease in clock cycle time and the percentage increase in instruction count**.

In this case, the clock cycle time decreased by: $(\frac{925-725}{925}) \approx 21.6\%$ But the instruction count increased by 36%. Since the instruction count penalty (36%) was greater than the clock speed gain (21.6%), the processor's overall performance was worse.

4.4 Do you consider the original CPU (as shown in Figure 4.21 of RISC-V text) a better overall design; or do you consider the new CPU a better overall design? Why?

Answer: The **original CPU is a better overall design**.

Why: The original design follows the RISC principle of "making the common case fast." Load and store instructions with an immediate offset are an extremely common case in compiled code. The original design handles this common case in a single instruction. Although this lengthens the clock cycle for *all* instructions, our calculation in 4.2 shows that this trade-off results in better overall performance. The new design simplifies the hardware but complicates the common case, forcing the compiler to issue more instructions, which ultimately results in slower execution.

Problem 5

(a) Examine the difficulty of adding a proposed `lwi.d rd, rs1, rs2` ("Load With Increment") instruction to RISC-V. Interpretation: Reg[rd]=Mem[Reg[rs1]+Reg[rs2]]

5.a1 Which new functional blocks (if any) do we need for this instruction?

Answer: None. The instruction requires a Register File (to read `rs1` and `rs2`), an ALU (to add them), and a Data Memory (to read from the resulting address). All these functional blocks already exist in the single-cycle datapath.

5.a2 Which existing functional blocks (if any) require modification?

Answer: The Main Control unit (and possibly the ALU Control unit). The datapath is already capable of executing this instruction, but the Control unit must be modified to recognize the new instruction's opcode/funct fields. It would need to generate the following combination of control signals:

- **RegWrite** = 1 (to write to `rd`)
- **ALUSrc** = 0 (to select `Reg[rs2]` as the second ALU input)
- **MemtoReg** = 1 (to select the Data Memory output for write-back)
- **MemRead** = 1 (to read from Data Memory)
- **MemWrite** = 0
- **ALUOp** = 10 (to signal an R-type operation)
- **Branch** = 0

5.a3 Which new data paths (if any) do we need for this instruction?

Answer: None. The existing datapath is sufficient.

- The path from the Register File's two read ports (`rs1`, `rs2`) to the ALU inputs exists (this is the R-type path).
- The path from the ALU result to the Data Memory address port exists (this is the `ld`/`sd` path).
- The path from the Data Memory read output to the Register File write port exists (this is the `ld` path).

Problem 6: Pipelining

6.1 What is the clock cycle time in a pipelined and non-pipelined processor?

- **Non-Pipelined:** In a non-pipelined (single-cycle) processor, the clock cycle must be long enough for a single instruction to complete all 5 stages.

$$T = \text{IF} + \text{ID} + \text{EX} + \text{MEM} + \text{WB}$$

$$T = 250\text{ ps} + 350\text{ ps} + 150\text{ ps} + 300\text{ ps} + 200\text{ ps} = \mathbf{1250}\text{ ps}$$

- **Pipelined:** In a pipelined processor, the clock cycle is determined by the latency of the **slowest** stage.

$$T = \max(\text{IF}, \text{ID}, \text{EX}, \text{MEM}, \text{WB})$$

$$T = \max(250, 350, 150, 300, 200) = \mathbf{350}\text{ ps}$$

IF	ID	EX	MEM	WB
250 ps	350 ps	150 ps	300 ps	200 ps

Also, assume that instructions executed by the processor are broken dow

ALU/Logic	Jump/Branch	Load	Store
45%	20%	20%	15%

6.2 What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor?

- **Non-Pipelined:** The latency of any instruction is the full clock cycle.

$$\text{Latency} = \mathbf{1250}\ \text{ps}$$

- **Pipelined:** A load instruction must pass through all 5 pipeline stages. The total latency is the number of stages multiplied by the clock period.

$$\text{Latency} = 5\ \text{stages} \times 350\ \text{ps/stage} = \mathbf{1750}\ \text{ps}$$

6.3 If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

- **Stage to Split:** To improve performance, you must split the bottleneck, which is the **slowest stage**. The slowest stage is **ID (Instruction Decode)** at 350 ps.
- **New Clock Cycle:** Splitting the ID stage (350 ps) creates two new stages, ID1 and ID2, each with a latency of $350/2 = 175$ ps. The new set of stage latencies is:

$$\{\text{IF} : 250, \text{ID1} : 175, \text{ID2} : 175, \text{EX} : 150, \text{MEM} : 300, \text{WB} : 200\}$$

  The new clock cycle time is the maximum of these new latencies.

$$T = \max(250, 175, 175, 150, 300, 200) = \mathbf{300}\ \text{ps}$$

6.4 Assuming there are no stalls or hazards, what is the utilization of the data memory?

The data memory is used during the **MEM** stage. According to the instruction mix, only **Load (ld)** and **Store (sd)** instructions access data memory.

$$\text{Utilization} = \text{Fraction}_{\text{ld}} + \text{Fraction}_{\text{sd}}$$

$$\text{Utilization} = 20\% + 15\% = \mathbf{35}\%$$

6.5 Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

The write-register port is used during the **WB (Write-Back)** stage. Only instructions that write a result back to the register file use this port.

- **ALU/Logic** instructions write their result. (45%)
- **Jump/Branch** instructions do not write to the register file. (0%)
- **Load** instructions write the loaded data. (20%)
- **Store** instructions write to data memory, not the register file. (0%)

$$\text{Utilization} = \text{Fraction}_{\text{ALU}} + \text{Fraction}_{\text{ld}}$$

$$\text{Utilization} = 45\% + 20\% = \mathbf{65}\%$$

Problem 7

What is the minimum number of cycles needed to completely execute n instructions on a CPU with a k stage pipeline? Justify your formula.

Answer

The minimum number of cycles is $\boldsymbol{k + (n - 1)}$.

Justification

- The first instruction must pass through all k stages of the pipeline to complete. This takes k clock cycles.
- While the first instruction is moving through the pipeline, all subsequent instructions are fetched and enter the pipeline one after another.
- After the first instruction completes (at the end of cycle k), the pipeline is full. A new instruction will complete on every subsequent clock cycle.
- Since there are $n - 1$ instructions remaining after the first one, it will take an additional $n - 1$ cycles for them all to complete.
- Therefore, the total time is k (for the first instruction) + $(n - 1)$ (for the remaining $n - 1$ instructions).

Problem 8

(a) Assume that $x11$ is initialized to 11 and $x12$ is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers $x13$ and $x14$ be?

```
addi x11, x12, 5
add  x13, x11, x12
addi x14, x11, 15
```

Answer

The final values would be $x13 = 33$ and $x14 = 26$.

Justification

This is a Read-After-Write (RAW) hazard with no forwarding. The new value for $x11$ is written during the Write-Back (WB) stage of the first instruction, but the subsequent instructions read from the register file during their Instruction Decode (ID) stage, which happens much earlier.

- **Cycle 3:** The `add x13, x11, x12` instruction is in its ID stage and reads its registers. At this time, the `addi x11...` instruction is in its EX stage. The register file has not been updated yet, so `add` reads the *original* value, $x11 = 11$. It calculates $x13 = 11 + 22 = 33$.
- **Cycle 4:** The `addi x14, x11, 15` instruction is in its ID stage. At this time, the `addi x11...` instruction is in its MEM stage. The register file is still not updated, so `addi x14` also reads the *original* value, $x11 = 11$. It calculates $x14 = 11 + 15 = 26$.
- **Cycle 5:** The `addi x11...` instruction is in its WB stage and finally writes the value $22 + 5 = 27$ into $x11$, but this is too late for the other instructions.

(b) Assume that $x11$ is initialized to 11 and $x12$ is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register $x15$ be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details.

```
addi x11, x12, 5
add  x13, x11, x12
addi x14, x11, 15
add  x15, x11, x11
```

Answer

The final value of $x15$ would be **54**.

Justification

This question adds a new timing rule: WB (write) happens at the beginning of a cycle, and ID (read) happens at the end. We must trace the pipeline to see when the `add x15...` instruction is in its ID stage.

- **Cycle 1:** `addi x11...` (IF)
- **Cycle 2:** `add x13...` (IF), `addi x11...` (ID)
- **Cycle 3:** `addi x14...` (IF), `add x13...` (ID), `addi x11...` (EX)
- **Cycle 4:** `add x15...` (IF), `addi x14...` (ID), `add x13...` (EX), `addi x11...` (MEM)
- **Cycle 5:** `add x15...` (ID), `addi x14...` (EX), `add x13...` (MEM), `addi x11...` (WB)

During **Cycle 5**:

1. The `addi x11...` instruction is in its **WB** stage. At the *beginning* of the cycle, it writes its result ($22 + 5 = \mathbf{27}$) into register $x11$.
2. The `add x15...` instruction is in its **ID** stage. At the *end* of the cycle, it reads register $x11$.
3. Because the write occurred at the beginning of the cycle, the ID stage reads the new, updated value of $x11 = 27$.
4. The instruction then calculates $x15 = 27 + 27 = 54$.

(c) Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards.

```
addi x11, x12, 5
add  x13, x11, x12
addi x14, x11, 15
add  x15, x13, x12
```

Answer

```
addi x11, x12, 5
nop
nop
nop
add  x13, x11, x12
addi x14, x11, 15
nop
nop
add  x15, x13, x12
```

Justification

In a pipeline with no hazard handling (and not using the special timing rule from 8b), a value written in the WB stage (Cycle 5) is not available to be read by an ID stage until Cycle 6.

- **First Hazard:** `add x13...` and `addi x14...` both read $x11$, which is written by `addi x11...`.
  - `addi x11...` is in WB in Cycle 5.
  - The ID stage of a dependent instruction must be delayed until Cycle 6.
  - The `add x13...` instruction's ID stage would normally be in Cycle 2. To delay it to Cycle 6, we must insert $6 - 2 = 4$ cycles, which means 3 `NOP` instructions.
- **Second Hazard:** `add x15...` reads $x13$, which is written by `add x13...`.
  - The `add x13...` instruction is now in ID in Cycle 6, and its WB stage will be in Cycle 9.
  - The `addi x14...` instruction is in ID in Cycle 7.
  - The `add x15...` instruction is in ID in Cycle 8. This is a problem, as it needs the value from `x13` which is not written until Cycle 9.
  - We must delay the `add x15...` ID stage from Cycle 8 until Cycle 10 (the cycle after C9's WB).
  - This requires 2 `NOP` instructions to be inserted between `addi x14` and `add x15`.