CS 382 Computer Architecture
Fall 2023 and Organization

Homework 3

Requirement (READ FIRST!)

You have to **type** the solutions. **Handwritten homework will not be graded and will receive zero credit.** You can annotate this document directly (you might need to know how to insert a picture into a PDF file), or you can submit a separate PDF, with solutions marked with question numbers.

1 (15 points)

When silicon chips are fabricated, defects in materials (*e.g.,* silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get "broken" and always register a logical 0. This is often called a "stuck-at-0" fault. Answer the following questions based on Figure 3.27 in the textbook.

1.1 (5 points)

Which instructions fail to operate correctly if the `Br` wire is stuck at 0?
If the Br wire is stuck at 0, the branching to the next instruction will fail, which means any of the B, BL, and CBZ will fail.

1.2 (5 points)

Which instructions fail to operate correctly if the `ALUsrc` wire is stuck at 0?
Should the ALUsrc wire become consistently fixed at 0, the processor will consistently interpret the second operand as 0 rather than considering their immediate values. Consequently, any instruction relying on immediate, such as SUB, LDR, STR, ANDS, ADDS, SUBF, ADD, ORR, will encounter failures as they necessitate the accurate consideration of immediate values.

1.3 (5 points)

Which instructions fail to operate correctly if the `RegWrite` wire is stuck at 0?
If the RegWrite wire is stuck at 0, it means the CPU cannot write the results of operations back to the register, which will lead to the failure of any instruction that necessitates writing back to RegWrite. This includes instructions like SUB, ADDS, ADD, AND, ORR, SUB, LDR, ANDS, LDR, ADD, SUBS, BL, CBZ, and RET, all of which rely on writing to a register.

2 (20 points)

Consider adding a new instruction, `SWAP Rd, Rn`, to our architecture, which will swap the data stored in `Rd` and `Rn`. Discuss the necessary changes that must be made to the datapath above. There is no need to design a new datapath - just discuss what should be added to the existing one.

- Add another write reg
- Add another regDataW

The introduction of the new SWAP instruction in assembly language necessitates several changes in the processor's design. Each instruction is identified by its unique opcode, and the SWAP instruction requires its own opcode for the control unit to decode and execute it. Specific control signals are generated by the control unit based on this opcode to manage various components in the datapath, allowing for the swap operation between registers. Implementing the SWAP instruction might require additional multiplexors to direct data flow for swapping register contents. The register file must support simultaneous read and write operations for two registers involved in the SWAP instruction. Additionally, the datapath needs modifications to handle the data to be written and the registers to be written for both involved registers. Introducing a new instruction like SWAP may necessitate revisiting the data hazard detection and handling mechanism to address potential hazards like read-after-write, write-after-read, or write-after-write scenarios.

3 (15 points)

Consider the addition of a multiplier to the CPU shown in Figure 3.27 in the textbook. This addition will add 200 ps to the latency of the ALU but will reduce the number of instructions by 20% (because there will no longer be a need to emulate the multiplication instruction). Answer the following questions based on **Chapter 3.3** of the textbook **(no pipelining)** and the following table for the latencies of the stages.

IF	ID	EX	ME	WB
200 ps	250 ps	150 ps	300 ps	200 ps

3.1 (5 points)

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

- The original clock cycle duration is 1100 ps, calculated by adding the time taken for each stage: 200 ps for IF, 250 ps for ID, 150 ps for EX, 300 ps for ME, and 200 ps for WB.

- After improvements, the revised clock cycle duration is 1300 ps, determined by the updated time allocation for each stage: 200 ps for IF, 250 ps for ID, 350 ps for EX, 300 ps for ME, and 200 ps for WB.

3.2 (5 points)

What is the speedup achieved by adding this improvement? *Hint:* *speedup*= 1 *–(new time)/(old time)*

Speedup = 1-(1300*.8/1100) = 0. 0.05454. Therefore, speedup is 5.45%

3.3 (5 points)

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

New clock cycle time: 1100/0.8 = 1375
150 + (1375-1100) = 425 ps is the slowest that the ALU can be while still resulting in improved performance.

4 (20 points)

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Questions in this problem assume that individual stages of the datapath have the latencies shown in Problem 3 above. Answer the following questions.

4.1 (5 points)

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

- In a non-pipelined processor, the clock cycle time is governed by the duration of the lengthiest stage in the pipeline. Considering the provided latencies:
  - IF (Instruction Fetch): 200 ps
  - ID (Instruction Decode): 250 ps
  - EX (Execute): 150 ps
  - ME (Memory Access): 300 ps
  - WB (Write Back): 200 ps
- The longest stage, Memory Access (ME), operates in 300 ps. Hence, in a non-pipelined processor, the clock cycle time equals the duration of this stage, which is 300 ps.
- Conversely, in a pipelined processor, although the clock cycle is often determined by the lengthiest pipeline stage, the presence of parallel operations across stages elevates the overall throughput. Nevertheless, the clock cycle time remains consistent with the non-pipelined processor at 300 ps since it is still constrained by the duration of the longest stage in the pipeline.

4.2 (5 points)

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

- In a non-pipelined processor, each stage of the instruction must be completed before the next begins. So, the total latency is the sum of all stages:
  - Total Latency = IF + ID + EX + ME + WB = 200 ps + 250 ps + 150 ps + 300 ps + 200 ps = 1100 ps
- In a pipelined processor, once the first instruction has passed through a stage, the next instruction can enter that stage in the next cycle. So, the total latency for an individual instruction like LDR is still the sum of all stages, but subsequent instructions will enter the pipeline before it completes, increasing throughput.
  - Total Latency for LDR = IF + ID + EX + ME + WB = 1100 ps
  - same as non-pipelined, but any further instructions are processed more quickly

4.3 (10 points)

- 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?
- To improve performance further, we can split the longest stage into two shorter stages. The longest stage is the ME stage (300 ps).

    - Split the ME stage into two stages of 150 ps each.

- New Clock Cycle Time:
    - The new clock cycle time is determined by the new longest stage.
    - After splitting, the longest stage is now either the ID stage or the two new ME stages, each with 150 ps.
    - Therefore, the new clock cycle time would be 150 ps.

- This reduced cycle time allows the pipelined processor to operate faster, albeit with a slight increase in the number of stages.

5 (20 points)

Consider the following instructions executed in a pipeline with full forwarding support (including WB write/read in the same cycle). Identify the value of which register is forwarded from a stage of an instruction to a stage of a subsequent instruction. (Completely impossible example: “The value of X5 is forwarded from the IF stage of instruction 1 to the WB stage of instruction 2.”) Submit a pipeline execution diagram as shown in the textbook.

```
1 LDR X20, [X19, 0]
2   LDR X21, [X19, 8]
3   ADD X22, X21, X20
4   SUB X23, X23, X22
```

In this context, the forwarding mechanism serves a pivotal role in maintaining the pipeline's efficiency by resolving data dependencies among instructions. For instance, during the EX stage of the ADD instruction (instruction 3), it necessitates the values of X20 and X21, which are being updated in the WB stage of instructions 1 and 2 respectively. Similarly, in the case of the SUB instruction (instruction 4), the value of X22 required during its EX stage is updated in the WB stage of instruction 3. Directly forwarding these values from the WB stage to the EX stage of subsequent instructions mitigates the need for pipeline stalls, significantly amplifying the processor's overall throughput.

	Cycle		1	2	3	4	5	6	7	8
Instructions	LDR X20, [X19, 0]	------>	IF	ID	EX	ME	WB			
	LDR X21, [X19, 8]	------>		IF	ID	EX	ME	WB		
	ADD X22, X21, X20	------>			IF	ID	EX	ME	WB	
	SUB X23, X23, X22	------>				IF	ID	EX	ME	WB

6 (20 points)

Consider the following instructions.

```
1 LDR   X1, [X6, 8]
2 ADD   X0, X1, X0
3 STR   X0, [X10, 4]
4 LDR   X2, [X6, 12]
5 SUB   X3, X0, X2
6 STR   X3, [X8, 24]
7 CBZ   X2, 40
```

6.1 (10 points)

Add `NOP` instruction to the code above so that it will run correctly on a pipeline **without forwarding**, but WB writes/reads in the same cycle. (A pipeline execution diagram is not needed for this problem. Sketching it may help, but it will not be graded.)

- To mitigate the hazard in the ADD X0, X1, X0 instruction caused by data dependency, insert a NOP after LDR X1, [X6, 8].
- Similarly, to ensure the correct execution of SUB X3, X0, and X2 without encountering data dependency issues, place a NOP after LDR X2, [X6, 12].

6.2 (10 points)

Optimize the code execution by re-arranging the instructions to get the same correct result faster.

- To optimize the instruction sequence and minimize interdependencies, prioritize operations that do not rely on immediate results from previous instructions. This approach allows sufficient time for data to be written back. Consider the following rearrangement:
- LDR X1, [X6, 8] (Load data into X1, as subsequent instructions don't rely on its immediate result)
- LDR X2, [X6, 12] (Load data into X2, giving time for X1 to be available)
- SUB X3, X0, X2 (Uses X0 and X2, which is available after the previous load instruction)
- ADD X0, X1, X0 (Depends on the value in X1, ensuring it comes after its corresponding load instruction)
- This rearrangement prioritizes the loading of data into registers first, allowing subsequent instructions that rely on these values to be placed afterward, minimizing dependencies, and reducing the need for NOPs.