DEVIN MAYA 2/16/24

CSE 120 Homework 4

Q1

Machine A :Non-pipelined, Single-stage datapath, Cycle time: 30ns

Machine B: Pipelined, 6 pipeline stages, Cycle time: 5ns

a)Which machine is better for faster execution of the same program? You can assume that there are no stalls in the program and has a large number of instructions ($I > 1$).

Assuming a large number of instructions and no stalls in the program:

- *Machine A*, being non-pipelined, processes one instruction every 30ns. This sequential processing results in lower throughput, especially noticeable with a large instruction count.
- *Machine B*, with its pipelined architecture, introduces the possibility of executing different stages of multiple instructions simultaneously. After filling the pipeline, it achieves a throughput of one instruction every 5ns, significantly enhancing performance.

Answer: *Machine B* is superior for faster execution, leveraging pipelining to reduce the effective instruction completion time and enhance throughput.

b) Let there be a new Machine C (Machine C: Pipelined, 4 pipeline stages, Cycle time: 8ns)with 4 pipeline stages and a cycle time of 8ns. If you assume that there are no stalls in the program, which machine is the fastest? (2pts)

Comparing the three machines when assuming no program stalls:

When analyzing Machine B and Machine C, both employing pipelining but with different cycle times and pipeline stages:

- *Machine B* offers the lowest cycle time (5ns), allowing for higher throughput, with the capability of delivering one instruction to completion every 5ns post-pipeline priming.
- *Machine C*, despite being pipelined, has a longer cycle time of 8ns compared to Machine B. While it provides a performance improvement over Machine A, it cannot match the throughput of Machine B.

Answer: Given the cycle times and assuming ideal pipelining conditions without stalls, *Machine B* remains the fastest. It benefits from both a high degree of pipelining and a short cycle time, optimizing instruction throughput.

Q2 You are tasked with improving the performance of a functional unit. The computation for the functional unit has 5 steps (A-E). Each step is indivisible. Assume there is no dependency between successive computations.

A) The greatest possible clock rate speedup possible with pipelining is equal to the number of pipeline stages. This is because pipelining allows multiple instructions to be executed simultaneously in different stages of the pipeline, effectively reducing the time taken to execute each instruction.

The principle behind pipelining in computer architecture is to divide a computational task into several sequential stages, allowing each stage to operate in parallel. This division increases the throughput of the system by executing multiple instructions simultaneously at different stages. Given that the functional unit's computation is divided into 5 indivisible steps (A-E) without dependencies between them, the greatest possible clock rate speedup with pipelining is analyzed as follows:

Pipelining enables each of the 5 steps to be handled by a different stage in the pipeline. Once fully operational, a new instruction can enter the pipeline at each clock cycle, theoretically allowing for up to a 5x speedup, corresponding to the number of pipeline stages. This optimal speedup assumes ideal conditions devoid of pipeline stalls, data hazards, or control hazards.

Answer: The greatest possible clock rate speedup with pipelining in this scenario is equal to the number of pipeline stages, which is 5.

B) For maximizing the clock rate, the minimum number of pipeline registers would be 4. These registers would be inserted between each stage of the functional unit (A, B, C, D, E). Using fewer pipeline stages would limit the potential speedup achievable with pipelining, while using more pipeline stages would increase complexity and potentially introduce additional latency.

To achieve the maximum clock rate with pipelining, it is essential to insert pipeline registers between each stage to hold the intermediate results. These registers are pivotal for synchronizing the stages and enabling the simultaneous execution of different instructions at various stages of the pipeline.

Given 5 computation steps (A-E) in the functional unit, the minimum number of pipeline registers required to enable pipelining would be:

- Between A and B
- Between B and C
- Between C and D
- Between D and E

This arrangement results in 4 pipeline registers needed to separate and synchronize each of the 5 stages. Inserting these registers allows each stage to operate independently on different instructions during each clock cycle, effectively maximizing the clock rate by minimizing the idle time of each stage.

Answer: For maximizing the clock rate through pipelining in a functional unit with 5 computation steps, the minimum number of pipeline registers required is 4. These registers, inserted between each stage (A-E), facilitate the simultaneous execution of instructions in a pipelined manner, enhancing overall performance while maintaining computational integrity.

Q3

For the following questions, it will be helpful for you to draw the Pipeline progression as F D X M W (which stand for the different pipeline stages, i.e., Fetch, Decode, eXecute, Memory, Write) for each instruction and trace in which clock cycle the computation for instruction is actually taking place. That way, you will know the correct value of the operands in the computation.

For all 3 parts, assume that before the code block is executed, x11 is initialized to 11 and x12 is initialized to 22. Other registers, if un-initialized can be assumed to be 0.

A)

Suppose you executed the code below on a pipelined system that does not handle data hazards. Also, the register file does not have the feature of reading and writing in the same clock cycle. For example, if the old value of x1 is 10. Now, let's assume that in Clock Cycle 5 (CC5), we are both writing a new value of 20 to x1 and also reading x1 in the same clock cycle. Then, the output value from the read would be 10, NOT 20. This new value of 20 will be available in x1 only from the next clock cycle CC6. In this context, what will be the final values of registers x13, x14, and x15 at the end of this code block?

- I1: addi x13, x12, 8
- I2: add x14, x12, x13
- I3: xor x15, x11, x12
- I4: sub x14, x13, x15
- I5: add x15, x13, x14

Considering the pipelined system without data hazard handling and no simultaneous read-write capability: Given the initial conditions and the specified code block, the final values of the registers are determined as follows:

- **I1:** $x13 = 30$ $(x12 + 8)$
- **I2:** Given that $x13$ is updated in the previous cycle, $x14$'s computation uses the new value of $x13$, resulting in $x14 = x12 + x13 = 22 + 30 = 52$.
- **I3:** $x15 = x11 \oplus x12$. Without the specific XOR result, we can denote the operation as $x15 = 11 \oplus 22$.
- **I4:** $x14$ is recalculated with the new values of $x13$ and $x15$ from I1 and I3.
- **I5:** $x15$ is recalculated with the updated values of $x13$ and $x14$ from I1 and I4.

B) (5pts)

Suppose you executed the code below on a pipelined system that does not handle data hazards. Also, the register file HAS the feature of reading and writing in the same clock cycle (i.e. fast register file (or) half-cycle writeback). For example, if the old value of x1 is 10. Now, let's assume that in Clock Cycle 5 (CC5), we are both writing a new value of 20 to x1 and also reading x1 in the same clock cycle. Then, the output value from the read would be 20 – courtesy of the fast register file (or) half-cycle writeback feature. This new value of 20 will be available in x1 in CC5. In this context, what will be the final values of registers x13, x14, and x15 at the end of this code block?

- I1: addi x13, x12, 8
- I2: add x14, x12, x13
- I3: xor x15, x11, x12
- I4: sub x14, x13, x15
- I5: add x15, x13, x14

With the feature of reading and writing in the same clock cycle:

- The computations follow similar logic as in part A but without the penalty of waiting for the next cycle for updates. Hence, the values read for computations within the same cycle as a write will reflect the most recent write operations.

C) (5pts)

Modify the code below by adding the minimum number of NOP instructions so that it runs correctly on a pipelined system that does not handle data hazards. Also, the register file does not have the feature of reading and writing in the same clock cycle. For example, if the old value of x1 is 10. Now, let's assume that in Clock Cycle 5 (CC5), we are both writing a new value of 20 to x1 and also reading x1 in the same clock cycle. Then, the output value from the read would be 10, NOT 20. This new value of 20 will be available in x1 only from the next clock cycle CC6.

- I1: addi x13, x12, 8
- I2: add x14, x12, x13
- I3: xor x15, x11, x12
- I4: sub x14, x13, x15
- I5: add x15, x13, x14

Given the initial code block and considering a pipelined system that does not handle data hazards and cannot read and write in the same clock cycle, we need to insert NOP (No Operation) instructions to ensure correct execution:

```
I1: addi x13, x12, 8
NOP
NOP
NOP
I2: add x14, x12, x13
NOP
```

```
NOP
NOP
I3: xor x15, x11, x12
NOP
NOP
NOP
I4: sub x14, x13, x15
NOP
NOP
NOP
I5: add x15, x13, x14
```

Explanation: - After **I1**, we insert three NOPs to ensure **x13** is correctly updated before being used in **I2**. - Similarly, after **I2** and **I3**, NOPs are added to account for the write-read delay. - For **I4**, we ensure that both **x13** and **x15** are updated. - Before **I5**, NOPs ensure that **x14** (updated in I4) is correctly read. This arrangement ensures that each instruction's data dependencies are resolved before the instruction is executed, avoiding potential data hazards due to the lack of data forwarding and simultaneous read-write capabilities.

D) (5pts)

Modify the code below by adding the minimum number of NOP instructions so that it runs correctly on a pipelined system that does not handle data hazards. Also, the register file HAS the feature of reading and writing in the same clock cycle. For example, if the old value of x1 is 10. Now, let's assume that in Clock Cycle 5 (CC5), we are both writing a new value of 20 to x1 and also reading x1 in the same clock cycle. Then, the output value from the read would be 10, NOT 20. This new value of 20 will be available in x1 only from the next clock cycle CC6.

- I1: addi x13, x12, 8
- I2: add x14, x12, x13
- I3: xor x15, x11, x12
- I4: sub x14, x13, x15
- I5: add x15, x13, x14

Make sure it is the minimum number of NOPs (might be 0 in between some instructions)! For example, without even thinking twice and not even looking at the code content below, we can state with confidence that adding 10 NOPs in between each instruction will solve all data hazards! The problem is, then, you are heavily penalizing your energy expenditure and throughput in catering

to these excess NOPs in your code. In a system that features reading and writing in the same clock cycle (half-cycle writeback), we can minimize the number of NOPs:

```
I1: addi x13, x12, 8
I2: add x14, x12, x13
I3: xor x15, x11, x12
I4: sub x14, x13, x15
I5: add x15, x13, x14
```

Answer: Since the register file supports simultaneous reading and writing, there is no need for NOP instructions to mitigate data hazards, optimizing for minimal NOP insertion and catering to the pipelined system's specific capabilities.

Q4 (10 pts)

A) (7 pts)

Assume the 5-stage pipeline processor has:Full Forwarding paths Half-cycle write back Stall capability based on non-forwardable data hazard detection Resolves branches in the Decode stage.

For dealing with control hazards on account of branch instructions, we will be dealing with the approach adopted by the textbook. That is, it is a static prediction of NOT TAKEN. So, if the branch is found to be NOT taken in the decode stage (Prediction=Reality), no bubble is introduced into the pipeline on account of the branch instruction. However, if a branch turns out to be taken, we pay the price in the form of an extra clock cycle. If the decode stage does not have the data available yet to determine whether the branch is Taken or Not Taken, it will stall the branch instruction at that stage until the data becomes available from the previous instruction. In this code example, the branch is NOT taken (i.e. x2 != x4), so the next instruction (addi) will be executed after the beq instruction.

Fill in the table to show how the pipeline progresses:

-

1	2	3	4	5	6	7	8	9	10	11	12	13
I1: add x2, x4, x3 I2: ld x4, 0(x1) I3: beq x2, x4, I4 I4: addi x5, x5, 10	F	D	X	M	W							

Hint: In the lecture, I discussed how the IF.Flush input to IF/ID register flushes the pipeline with a NOP at the decode stage if a branch is taken. So, you may be wondering whether you need to indicate a NOP in this tabular view for the pipeline. However, since you are

only indicating which pipeline stage each valid instruction is and not the fate of the NOPs, you won't need to be bothered with indicating the NOPs in the table.

Assuming the 5-stage pipeline processor is equipped with:

- Full Forwarding paths
- Half-cycle write back
- Stall capability based on non-forwardable data hazard detection
- Branch resolution in the Decode stage with a static prediction of NOT TAKEN.

Given the branch is NOT taken, the instruction sequence and their respective pipeline stages are shown below:

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
I1: add x2, x4, x3	F	D	X	M	W							
I2: ld x4, 0(x1)		F	D	X	M	W						
I3: beq x2, x4, I4			F	D	X	M	W					
I4: addi x5, x5, 10				F	D	X	M	W				

The pipeline progresses smoothly with full forwarding and half-cycle write-back capabilities, ensuring no additional stalls for data hazards and executing the branch prediction strategy effectively.

B) (3 pts)

You are considering removing the forwarding paths from your processor because they are too expensive. How will removing the forwarding paths qualitatively affect the 3 terms in the CPU Performance Iron Law equation? For each term, explain why it will increase, stay the same, or decrease. If it depends on other factors, explain those too. You can assume your processor is pipelined with full forwarding (initially) and it is executing a generic workload.

- Instruction count
- CPI
- Clock Frequency

Removing the forwarding paths from the processor would qualitatively affect the CPU Performance Iron Law terms as follows:

- **Instruction count:** Remains the same. The total number of instructions executed is not directly affected by the presence or absence of forwarding paths.

- **CPI (Cycles Per Instruction):** Likely to increase. Without forwarding paths, data hazards that were previously mitigated now cause delays, as dependent instructions must wait for data to be written back to the register file.
- **Clock Frequency:** Could potentially increase due to a simpler pipeline design with reduced complexity. However, any increase in clock frequency may not compensate for the performance loss due to a higher CPI, resulting in a net negative impact on overall performance.

Forwarding paths plays a role in enhancing the efficiency of pipelined processors by reducing the need for stalls and handling data hazards more effectively. Their removal necessitates alternative approaches to manage data dependencies, which could adversely impact processor performance, especially in terms of CPI.