Lab3 Report

Liron Cohen 207481268

Yuval Mor 209011543

Question 1 - Harvard Architecture

In Harvard Architecture, the instruction memory and data memory are two different units that use different buses to communicate with the control unit of the CPU.

Advantages:

1. Two separate memories allow for parallel access to both instructions and data. Therefore, it reduces the number of structural hazards, which results in faster execution.
2. Data memory and instruction memory can use different sizes for words/address space.
3. More secure (potentially) – programs cannot overwrite their own or other programs' instructions, maliciously or by mistake.

Disadvantages:

1. Design complexity, area and cost - Designing a CPU with multiple buses and multiple memory components is more complex, requires more space and ultimately more expensive.
2. Reduces memory utilization - instruction memory cannot be used for data and vice versa.

Since we wish to maximize the performance gain from pipelining, we think that Harvard architecture is a good choice.

In addition, for our DMA to work, we must have a different bus for the instruction memory. Therefore, if we use the Von Neuman Architecture would result in the bus being busy every cycle and the DMA will be suffering from starvation.

<u>Question 2 - Structural and Data Hazards</u>

There are three main types of hazards:

1. <u>Structural hazard</u> - occurs when two different instructions require access to the same resource simultaneously. In our case, such hazards happen when there is an ST instruction that immediately followed by a LD instruction (resulting in reading and writing to memory in the same cycle).
   Our solution is to compare the load and store registers of the instructions and stall the LD instruction.

   The original timing diagram that shows the hazard:

Instruction	0	1	2	3	4	5	6	7
ST	F0	F1	D0	D1	E0	E1		
LD		F0	F1	D0	D1	E0	E1	

   After adding a stall:

Instruction	0	1	2	3	4	5	6	7
ST	F0	F1	D0	D1	E0	E1		
LD		F0	F1	D0	D1	stall	E0	E1

2. <u>Data hazard</u> – occurs when there is a dependency between two adjacent instructions. In our single core architecture, this is possible when an instruction reads data from the same location a previous instruction writes to, and the updated value is not yet available at the time of execution (read-after-write hazard).
   Our solution is to check if the source registers are the destination registers of the previous instructions that weren't already updated. If so, we use forwarding from E1 to D1 or E0.

The original timing diagram that shows the hazard:

Instruction	0	1	2	3	4	5	6	7
ADD r5, r3, r4,0	F0	F1	D0	D1	E0	E1		
ADD r6, r6, r5,0		F0	F1	D0	D1	E0	E1	
ADD r7, r6, r5,0			F0	F1	D0	D1	E0	E1

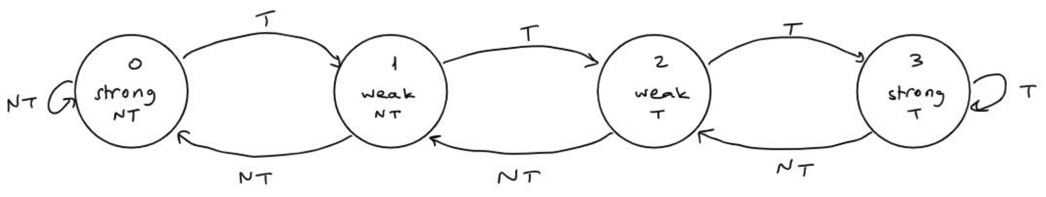
After adding forwarding:

Instruction	0	1	2	3	4	5	6	7
ADD r5, r3, r4,0	F0	F1	D0	D1	E0	E1		
ADD r6, r6, r5,0		F0	F1	D0	D1	E0	E1	
ADD r7, r6, r5,0			F0	F1	D0	D1	E0	E1

3. Control hazard - Occurs due to the usage of branch instructions. When we need to jump to a different instruction, the pipeline can continue executing instructions that won't execute because of the jump.
   Our solution contains branch prediction (detailed in the next question) and flushes the pipeline as necessary.

Question 3 - Branches and branch prediction

For our branch prediction, we used a branch history table of size 10 with a 2-bit predictor. The scheme of the predicator is as followed:



In DEC0 stage - if we had a branch operation, we checked in the history table at index PC % 10, and if the prediction showed that the branch should be taken (value of 2 or 3), we flush the pipeline.

In EXEC1 stage - if we had a branch operation we updated the table according to the state machine above. We also calculated the next PC and flushed if needed.

We will illustrate the flush using an example when the first branch is taken and the prediction says it's taken. The next instruction will be flushed from the pipeline.

Instruction	0	1	2	3	4	5	6	7
JLT r0, r1, r2 12	F0	F1	D0	D1	E0	E1		
ADD r6, r6, r5, 0		~~F0~~	~~F1~~					

Another example is when the first branch is taken and the prediction says it's not taken but we understand in EXEC1 that it is taken:

Instruction	0	1	2	3	4	5	6	7
JLT r0, r1, r2 12	F0	F1	D0	D1	E0	E1		
ADD r6, r6, r5, 0		~~F0~~	~~F1~~	~~D0~~	~~D1~~	~~E0~~		

<u>Question 4 - Low level simulator pipeline implementation</u>

The required files are attached.

<u>Question 5 - Low level testing</u>

We verified that the outputs are the same as the previous labs.

The required files are attached.

<u>Question 6 - Speedup comparison, next generation improvements</u>

1. Speedup results are given below:

Program	number of instructions	number of un-pipelined cycles (lab 2)	number of pipelined cycles (lab 3)	un-pipelined CPI	pipelined CPI
sqrtq	92	552	317	6	3.445
example	61	366	141	6	2.311
add	15	90	25	6	1.667

2. We can see from the table that the new CPI is not 1. We expected it to be higher than 1 because of the hazards that we have in the pipeline (see the answer to question 2 above). When there are hazards, stalling and flushes are needed, and thus we get a CPI that is higher than 1. In particular, the more taken branch instructions we have, the more flushes and the CPI is even higher. For example, in our sqrtq program, we have a lot of branch instructions, and we can explain the high CPI because of control hazards.
3. Some ways to improve our CPI:
    a. Improving branch prediction method - by improving the branch prediction accuracy, we can reduce the number of pipeline stalls and flushes caused by branches and improve CPI.
    b. Implementing out-of-order execution: Out-of-order execution allows the CPU to execute instructions in a different order than they appear in the instruction stream. This can improve CPI by allowing the CPU to execute instructions that are ready to go before waiting for dependent instructions to complete.
    c. Implementing a method to reduce data dependencies - this way we can minimize the number of data hazards.

<u>Question 7 - DMA testing in a pipeline environment</u>

We copied the validation program from lab 2 and changed it. We added two instructions at lines 25 and 26 to test structural hazards. All the other hazards already existed.

<u>Data hazard</u>- We can see in line 10 that the JLT instruction uses R2 that was written to at the line before. We can see in the cycle trace that the forwarding worked and that R2 was updated in the same cycle.

<u>Structural hazard</u>- we added a LD instruction and a ST in lines 25 and 26. We can see in the cycle trace file that the hazard was handled correctly. Also, we can see at the end of the instruction trace that R3 = 100.

<u>Control hazard</u>- The first branch in our code (line 10) is taken, but our first branch prediction is that it will not be taken. We can see in the cycle trace that there was a flush at cycle 17.

The DMA functionality test is the same as lab 2. We can see in the instruction trace that R2 = 1, so the DMA works correctly.

```
// Liron Cohen 207481268
// Yuval Mor 209011543

// Initialization of the parameters
asm_cmd(ADD, 3, 1, 0, 50); // 0: R3 = 50 SOURCE
asm_cmd(ADD, 4, 1, 0, 60); // 1: R4 = 60 DESTINATION
asm_cmd(ADD, 5, 1, 0, 50); // 2: R5 = 50 LENGTH

// Calling the copy operation
asm_cmd(CPY, 4, 3, 5, 0); // 3: START COPY

// Main program code to run in parallel
asm_cmd(ADD, 2, 1, 0, 50); // 4: R2 = 50
asm_cmd(ADD, 5, 1, 0, 55); // 5: R5 = 55
asm_cmd(ADD, 3, 0, 0, 0); // 6: R3 = 0
asm_cmd(LD,  4, 0, 2, 0); // 7: R4 = MEM[R2]
asm_cmd(ADD, 3, 3, 4, 0); // 8: R3 += R4
asm_cmd(ADD, 2, 2, 1, 1); // 9: R2++
asm_cmd(JLT, 0, 2, 5, 7); // 10: if R2 < R5 jump to line 7 // Data Hazard and Control Hazard

// Polling and checking for copy completion
asm_cmd(POL, 2, 0, 0, 0); // 11: R[2] = DMA_S->REMAIN (number of bytes left to copy)
asm_cmd(JNE, 0, 2, 0, 11); // 12: if R[2] != 0 jump to 11

// Copy validation
asm_cmd(ADD, 2, 1, 0, 1); // 13: R2 = 1 COPY CORRECT
asm_cmd(ADD, 3, 1, 0, 50); // 14: R3 = 50 SOURCE
asm_cmd(ADD, 4, 1, 0, 60); // 15: R4 = 60 DESTINATION
asm_cmd(ADD, 5, 1, 0, 100); // 16: R5 = 100  SOURCE + LENGTH
asm_cmd(LD, 6, 0, 3, 0); // 17: R6 = MEM[R3]
asm_cmd(LD, 7, 0, 4, 0); // 18: R7 = MEM[R4]
asm_cmd(JEQ, 0, 6, 7, 22); // 19: if (R[6] == R[7]) jump to 22
asm_cmd(ADD, 2, 0, 0, 0); // 20: R2 = 0 COPY WRONG
asm_cmd(JEQ, 0, 0, 0, 25); // 21: jump to 25 to check structural hazard
asm_cmd(ADD, 3, 3, 1, 1); // 22: R3++
asm_cmd(ADD, 4, 4, 1, 1); // 23: R4++
asm_cmd(JLT, 0, 3, 5, 17); // 24: if R3 < R5 jump to line 17
asm_cmd(ST, 0, 5, 1, 1000); // 25: MEM[1000] = R5 = 100
asm_cmd(LD, 3, 0, 1, 1000); // 26: R3 = MEM[1000] if RAW succeeded R3 = 100
asm_cmd(HLT, 0, 0, 0, 0); // 27: HALT
```