ASSIGNMENT-4

Dinesh Yajjala
EE21B042

November 6, 2023

1 Part-A: HAZARDS

1.1 RISC-V Assembly Code

The following RISC-V assembly code calculates the dot product of two vectors of size 10:

```
# Initialize loop counter to 10
addi x5, x0, 10

# Initialize addresses of vectors in x20 and x21
addi x20, x0, 1024
addi x21, x0, 1134

# Initialize result to 0
addi x10, x0, 0

# Initialize vector1
addi x1, x0, 1
sb x1, 0(x20)
addi x1, x0, 2
sb x1, 1(x20)
addi x1, x0, 3
sb x1, 2(x20)
addi x1, x0, 4
sb x1, 3(x20)
addi x1, x0, 5
sb x1, 4(x20)
addi x1, x0, 6
sb x1, 5(x20)
addi x1, x0, 7
sb x1, 6(x20)
addi x1, x0, 8
sb x1, 7(x20)
```

```
addi x1, x0, 9
sb x1, 8(x20)
addi x1, x0, 10
sb x1, 9(x20)

# Initialize vector2
addi x2, x0, 10
sb x2, 0(x21)
addi x2, x0, 9
sb x2, 1(x21)
addi x2, x0, 8
sb x2, 2(x21)
addi x2, x0, 7
sb x2, 3(x21)
addi x2, x0, 6
sb x2, 4(x21)
addi x2, x0, 5
sb x2, 5(x21)
addi x2, x0, 4
sb x2, 6(x21)
addi x2, x0, 3
sb x2, 7(x21)
addi x2, x0, 2
sb x2, 8(x21)
addi x2, x0, 1
sb x2, 9(x21)

loop:
    # Load elements from vectors
    lbu x6, 0(x20)
    lbu x7, 0(x21)

    # Multiply elements and add to result
    mul x8, x6, x7
    add x10, x10, x8

    # Increment addresses
    addi x20, x20, 1
    addi x21, x21, 1

    # Decrement loop counter and branch if not zero
    addi x5, x5, -1
    bne x5, x0, loop
```

1.2 Data and Control Hazards

1.2.1 Data Hazards

1. Read-After-Write (RAW) Hazard: The instruction ‘lbu x7, 0(x21)‘ reads from register ‘x21‘ after the instruction ‘addi x21, x21, 1‘ writes to ‘x21‘.

2. Read-After-Write (RAW) Hazard: The instruction ‘bne x5, x0, loop‘ reads from register ‘x5‘ after the instruction ‘addi x5, x5, -1‘ writes to ‘x5‘.

1.2.2 Control Hazards

No control hazards were identified in the code.

1.3 Throughput with and without Forwarding

The throughput of the code was calculated with and without forwarding:

1. With forwarding activated:

$$\begin{aligned} \text{Number of instructions} &= 133 \\ \text{Number of clock cycles taken} &= 157 \\ \text{Throughput} &= \frac{133}{157} \\ &= 0.847 \end{aligned}$$

2. Without forwarding activated:

$$\begin{aligned} \text{Number of instructions} &= 133 \\ \text{Number of clock cycles taken} &= 237 \\ \text{Throughput} &= \frac{133}{237} \\ &= 0.561 \end{aligned}$$

1.4 Throughput with and without Flushing

The throughput of the code was calculated with and without flushing. The number of clock cycles does not change with and without flushing in the code because there are no branch or jump instructions that would cause a pipeline flush. Flushing is a technique used to handle control hazards, which occur when the pipeline makes incorrect assumptions about the execution path of a program. This can happen in situations where the execution path depends on a branch or jump instruction. If the pipeline predicts the execution path incorrectly, it needs to flush, or discard, the instructions that were fetched incorrectly. In the provided code, the only control flow instruction is the ‘bne‘ instruction at the end of the loop. However, this instruction does not cause a control hazard because the pipeline can accurately predict the execution path of the program.

The ‘bne‘ instruction always branches back to the start of the loop until the loop counter reaches zero, at which point the program ends. Therefore, the pipeline never needs to flush any instructions, and the number of clock cycles remains the same whether flushing is enabled or not.

1.5 Comparison of Results

The results show that enabling forwarding can significantly improve the throughput of the code by reducing the number of clock cycles required to execute the instructions. This is because forwarding can resolve data hazards by allowing the result of an instruction to be forwarded to subsequent instructions before the result has been written back to the register file. On the other hand, flushing does not affect the throughput of the code because there are no control hazards that would require the pipeline to be flushed.

2 Part-B: BRANCH PREDICTION

2.1 Branch Prediction Strategies

In this report, we analyze the performance of a program when run on the RISCV simulator using different branch prediction strategies. The strategies used are Always Taken (AT), Always Not Taken (NT), Back Taken Forward Not Taken (BTFNT), and Branch Prediction Buffer (BPB).

Strategy	Instructions	Cycles	Avg Cycles per Instruction	Branch Prediction Accuracy	Control Hazards	Data Hazards	Memory Hazards
AT	152	262	1.7237	0.4706	26	76	1
NT	140	228	1.6286	0.5833	23	73	1
BTFNT	152	250	1.6447	0.7059	22	77	1
BPB	152	262	1.7237	0.4706	26	76	1

Table 1: Performance of the program with different branch prediction strategies

2.2 Discussion

The results show that the BTFNT strategy achieved the highest branch prediction accuracy, while the AT and BPB strategies resulted in the highest number of cycles. The NT strategy had the lowest number of cycles, indicating a more efficient execution. However, it also had a lower branch prediction accuracy compared to BTFNT. The number of control hazards and data hazards varied slightly between the strategies, but the number of memory hazards remained constant.