BLG322E Computer Architecture Homework 2 Report

Mustafa Can Çalışkan
150200097

April 13, 2025

Contents

1 Introduction

This report presents the solutions to the second assignment for the BLG 322E Computer Architecture course.

In Question 1, I analyze the execution of a RISC processor with a 5-stage pipeline that includes: Instruction Fetch (IF), Instruction Decode/Register Read (DR), Execute (EX), Memory (ME), and Write Back (WB). The processor has limited forwarding capabilities and specific handling for register file access hazards. The analysis includes creating space-time diagrams, calculating penalty cycles, and optimizing the execution through software-based solutions.

Question 2 examines branch prediction mechanisms, specifically comparing different strategies: one-bit dynamic prediction and two-bit dynamic prediction with varying initial states. The goal is to determine the accuracy of each prediction method by counting correct predictions and mispredictions.

2 Question 1

In this question, I analyze the execution of a program on a RISC processor with a 5-stage pipeline. The processor has the following characteristics:

- A single forwarding connection between the ME stage output and ALU inputs
- Register file access hazard is fixed (writing in first half, reading in second half of cycle)
- Branch target calculation and decisions occur in the EX stage

2.1 a) Space-Time Diagram Analysis

The space-time diagram for the execution of the given program with software-based NOOP instructions to resolve all data and branch conflicts can be found in the file `space_time_diagrams.xlsx`, in the sheet titled "State-Time Diagram of a)".

The instruction sequence with inserted NOOPs is as follows:

```
XOR R2, R2, R1
NOOP                // Inserted to resolve data hazard with R1
ADD R1, $09, R3
ADD R1, $03, R4
NOOP                // Inserted to resolve data hazard with R4
ADD R5, R4, R5
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
BNZ LOOP
```

```
NOOP                // Inserted due to branch delay slot
NOOP                // Inserted due to branch delay slot
ADD R5, R4, R5      // Loop iteration 2 begins
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
BNZ LOOP
NOOP                // Inserted due to branch delay slot
NOOP                // Inserted due to branch delay slot
ADD R5, R4, R5      // Loop iteration 3 begins
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
BNZ LOOP            // Branch not taken, loop exits
ADD R6, $01, R6
BRU FINISH
NOOP                // Inserted due to branch delay slot
LDL $08(R4), R1
```

The NOOPs are inserted for the following reasons:

1. After `XOR R2, R2, R1`: A NOOP is inserted because R1 is modified and immediately used in the next instruction `ADD R1, $09, R3`. Since the processor only has forwarding from ME to EX stage, we need one NOOP to resolve this data hazard.

2. After `ADD R1, $03, R4`: A NOOP is inserted because R4 is modified and then used in `ADD R5, R4, R5`. The processor cannot forward the result in time without a NOOP.

3. After `SUB R3, $03, R6`: A NOOP is inserted because R6 is modified and then used as a base address in `STL $08(R6), R5`. This creates a data hazard that requires a NOOP.

4. After each `BNZ LOOP`: Two NOOPs are inserted because branch target calculation and decisions occur in the EX stage. This causes a two-cycle delay when the branch is taken, requiring two NOOPs to prevent incorrect instruction execution.

5. After `BRU FINISH`: One NOOP is inserted due to the branch delay slot for the unconditional branch.

The total penalty due to these conflicts is 10 NOOPs: 1 after XOR, 1 after ADD R1, 3 NOOPs (one per iteration) after SUB R3, 4 NOOPs (two per taken branch, for 2 taken branches), 1 after BRU FINISH, for a total of 10 cycles of penalty.

2.2 b) Total Penalty Calculation

To calculate the total amount of penalty as a function of iterations n, I need to analyze which NOOPs would appear in a general case.

For each iteration of the loop, the following NOOPs are inserted:

- 1 NOOP after `SUB R3, $03, R6` due to the data hazard with R6
- 2 NOOPs after `BNZ LOOP` due to the branch delay when the branch is taken (only for iterations where the branch is actually taken, which is $n - 1$ times)

Additionally, there are fixed NOOPs that appear regardless of the number of iterations:

- 1 NOOP after `XOR R2, R2, R1` due to the data hazard with R1
- 1 NOOP after `ADD R1, $03, R4` due to the data hazard with R4
- 1 NOOP after `BRU FINISH` due to the unconditional branch delay slot

Therefore, the total penalty as a function of n is:

$$\begin{aligned}\text{Total Penalty} &= \text{NOOPs before loop} + \text{NOOPs in loop} + \text{NOOPs after loop} \quad (1)\\ &= (1+1) + [1 \times n + 2 \times (n-1)] + 1 \quad (2)\\ &= 2 + n + 2n - 2 + 1 \quad (3)\\ &= 3n + 1 \quad (4)\end{aligned}$$

For our specific example with $n = 3$ iterations, the total penalty is $3 \times 3 + 1 = 10$ cycles, which matches our analysis in part (a).

Therefore, for a program with n iterations, the total penalty would be $3n+1$ cycles.

2.3 c) CPI Calculation

To calculate the Cycles Per Instruction (CPI) for the given program, I need to determine the total number of cycles and the total number of instructions executed.

From the space-time diagram in part (a), the program execution took 35 cycles in total. This includes both actual instructions and the NOOP instructions that were inserted to resolve hazards.

The program consisted of 21 actual instructions (excluding NOOPs):

- Initial instructions (before loop): 3 instructions
  - XOR R2, R2, R1
  - ADD R1, $09, R3
  - ADD R1, $03, R4

- Loop body executed 3 times: 15 instructions
  - First iteration: 5 instructions (ADD, SUB, STL, SUB, BNZ)
  - Second iteration: 5 instructions (ADD, SUB, STL, SUB, BNZ)
  - Third iteration: 5 instructions (ADD, SUB, STL, SUB, BNZ)
- Instructions after loop: 3 instructions
  - ADD R6, \$01, R6
  - BRU FINISH
  - LDL \$08(R4), R1
- Total: 3 + 15 + 3 = 21 actual non-NOOP instructions

There were 10 NOOP instructions inserted as penalty cycles, as calculated in parts (a) and (b).

Therefore, the CPI can be calculated as:

$$\begin{aligned} \text{CPI} &= \frac{\text{Total cycles}}{\text{Total instructions executed}} && (5)\\ &= \frac{35}{31} && (6)\\ &= 1.113 && (7) \end{aligned}$$

For this program, the CPI is approximately 1.113, which is higher than the ideal CPI of 1.0 for a pipelined processor with no hazards. This increase in CPI is due to the data and control hazards that required NOOP insertions.

2.4 d) Code Optimization

To minimize the penalty cycles, I've implemented software-based optimizations through instruction reordering. The optimized space-time diagram can be found in the file `space_time_diagrams.xlsx`, in the sheet titled "State-Time Diagram of d)".

The original unoptimized program execution has several NOOPs and follows this sequence:

```
XOR R2, R2, R1
NOOP                // Inserted to resolve data hazard with R1
ADD R1, $09, R3
ADD R1, $03, R4
NOOP                // Inserted to resolve data hazard with R4
ADD R5, R4, R5
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
```

```
BNZ LOOP
NOOP                // Inserted due to branch delay slot
NOOP                // Inserted due to branch delay slot
ADD R5, R4, R5      // Loop iteration 2 begins
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
BNZ LOOP
NOOP                // Inserted due to branch delay slot
NOOP                // Inserted due to branch delay slot
ADD R5, R4, R5      // Loop iteration 3 begins
SUB R3, $03, R6
NOOP                // Inserted to resolve data hazard with R6
STL $08(R6), R5
SUB R4, $01, R4
BNZ LOOP            // Branch not taken, loop exits
ADD R6, $01, R6
BRU FINISH
NOOP                // Inserted due to branch delay slot
LDL $08(R4), R1
```

After applying instruction reordering, the optimized code sequence is:

```
XOR R2, R2, R1
NOOP                // Data hazard with R1 (not eliminated)
ADD R1, $09, R3
ADD R1, $03, R4
NOOP                // Data hazard with R4 (not eliminated)
ADD R5, R4, R5
SUB R3, $03, R6
SUB R4, $01, R4     // Moved before BNZ, eliminating need for NOOP with R6
BNZ LOOP            // Branch handling optimized
STL $08(R6), R5     // Now acts as delay slot instruction instead of NOOP
SUB R3, $03, R6     // Loop iteration 2 begins
ADD R5, R4, R5      // Reordered to optimize pipeline flow
SUB R4, $01, R4
BNZ LOOP            // Branch handling optimized
STL $08(R6), R5     // Again acts as delay slot instruction
SUB R3, $03, R6     // Loop iteration 3 begins
ADD R5, R4, R5      // Same reordering pattern maintained
SUB R4, $01, R4
STL $08(R6), R5     // Acts as instruction before branch
BNZ LOOP            // Final branch
BRU FINISH          // Unconditional branch
ADD R6, $01, R6     // Acts as delay slot instruction
LDL $08(R4), R1
```

The key instruction reordering strategies implemented are:

1. **Reordering within each loop iteration:** The major optimization was moving SUB R4, $01, R4 before BNZ, which eliminated the need for the NOOP after SUB R3, $03, R6. This rearrangement creates sufficient distance between the definition of R6 in SUB R3, $03, R6 and its use in STL $08(R6), R5.

2. **Branch delay slot utilization:** By placing the STL $08(R6), R5 instruction after the BNZ LOOP instruction, it effectively serves as a useful instruction in what would otherwise be a branch delay slot requiring a NOOP. This optimization is applied consistently in all loop iterations.

3. **Final branch optimization:** The ADD R6, $01, R6 instruction was placed after BRU FINISH to serve as a delay slot instruction, eliminating the need for a NOOP after the unconditional branch.

Some data hazards couldn't be eliminated through reordering alone (the NOOPs after XOR R2, R2, R1 and after ADD R1, $03, R4).

Therefore, the total amount of penalty in clock cycles with these new solution is 2.

3 Question 2

3.1 a) One-Bit Dynamic Predictor

In this section, I analyze the behavior of a one-bit dynamic branch predictor with two different branch instructions. The predictor starts in "Not Taken" state and updates after each branch.

The test case features a loop that executes from counter value 40 down to 0, with two branches:

- BRZ L1: Branches when counter is divisible by 4
- BNZ LOOP: Branches when counter is not zero

For BRZ L1, I observed that it's taken exactly when counter values are 40, 36, 32, 28, 24, 20, 16, 12, 8, and 4. This creates a cyclical pattern where one taken branch is followed by three not-taken branches.

When tracking the predictor state through four iterations:

- First iteration: Predicts N (initial state), encounters T $\rightarrow$ incorrect $\rightarrow$ state changes to T
- Second iteration: Predicts T, encounters N $\rightarrow$ incorrect $\rightarrow$ state changes to N
- Third iteration: Predicts N, encounters N $\rightarrow$ correct $\rightarrow$ state remains N

- Fourth iteration: Predicts N, encounters N → correct → state remains N

Each four-iteration block therefore contains 2 incorrect and 2 correct predictions. Since the entire execution contains 10 such blocks, the final tally for BRZ L1 is:

- 20 correct predictions
- 20 incorrect predictions

For BNZ LOOP, the branch behavior is more consistent. It's taken for all iterations except the last one (39 taken, 1 not taken). The predictor performs as follows:

- First iteration: Predicts N, encounters T → incorrect → state changes to T
- Iterations 2-39: Predicts T, encounters T → all correct → state remains T
- Last iteration: Predicts T, encounters N → incorrect → state changes to N

This results in much better accuracy for BNZ LOOP:

- 38 correct predictions
- 2 incorrect predictions

3.2 b) Two-Bit Dynamic Predictor (Weakly Taken)

For this analysis, I implement a two-bit saturating counter predictor with initial state "10" (Weakly Taken). This predictor uses four states with the following meanings:

- 00: Strongly Not Taken (predicts N)
- 01: Weakly Not Taken (predicts N)
- 10: Weakly Taken (predicts T)
- 11: Strongly Taken (predicts T)

The counter increments when a branch is taken (unless already at 11) and decrements when not taken (unless already at 00).

For BRZ L1 with its (T,N,N,N) pattern:

Starting from state 10, the first T outcome is correctly predicted and moves the state to 11. The subsequent N outcomes gradually push the counter toward not-taken states, with occasional Ts bringing it partially back up.

I tracked the state transitions through several iterations:

- Initial state: 10 (Weakly Taken)

- Counter=40: Predicts T, actual T → correct → state 11
- Counter=39: Predicts T, actual N → incorrect → state 10
- Counter=38: Predicts T, actual N → incorrect → state 01
- Counter=37: Predicts N, actual N → correct → state 00
- Counter=36: Predicts N, actual T → incorrect → state 01
- Counter=35: Predicts N, actual N → correct → state 00
- Counter=34: Predicts N, actual N → correct → state 00
- Counter=33: Predicts N, actual N → correct → state 00

After analyzing the complete execution sequence, I've determined that this predictor achieves:

- 29 correct predictions
- 11 incorrect predictions

This marks a significant improvement over the one-bit predictor for this branch pattern.

For BNZ LOOP with its largely consistent taken pattern:

- First iteration: Predicts T (state 10), encounters T → correct → advances to state 11
- Iterations 2-39: Predicts T, encounters T → all correct → remains in state 11
- Final iteration: Predicts T, encounters N → incorrect → moves to state 10

The results for BNZ LOOP with this predictor are:

- 39 correct predictions
- 1 incorrect prediction

3.3 c) Two-Bit Dynamic Predictor (Weakly Not Taken)

In this section, I analyze the same branches using a two-bit predictor initially set to "01" (Weakly Not Taken).

For BRZ L1 and its (T,N,N,N) pattern:

Beginning in state 01, the first T outcome is incorrectly predicted. This changes the state to 10 (Weakly Taken). As execution continues, the predictor adjusts through various states based on the pattern.

The detailed state transitions for the first several iterations are:

- Initial state: 01 (Weakly Not Taken)
- Counter=40: Predicts N, actual T → incorrect → state 10
- Counter=39: Predicts T, actual N → incorrect → state 01
- Counter=38: Predicts N, actual N → correct → state 00
- Counter=37: Predicts N, actual N → correct → state 00
- Counter=36: Predicts N, actual T → incorrect → state 01

My complete analysis of all iterations shows:

- 29 correct predictions
- 11 incorrect predictions

This matches the performance of the Weakly Taken initialization, though the path through the prediction states differs.

For BNZ LOOP:

- First iteration: Predicts N (state 01), encounters T → incorrect → changes to state 10
- Second iteration: Predicts T, encounters T → correct → advances to state 11
- Iterations 3-39: Predicts T, encounters T → all correct → remains at state 11
- Final iteration: Predicts T, encounters N → incorrect → moves to state 10

The final results for BNZ LOOP with this predictor are:

- 38 correct predictions
- 2 incorrect predictions

To summarize all three prediction strategies, I've compiled the results in Table 1 below:

Predictor	**BRZ L1**		**BNZ LOOP**	
	Incorrect	Correct	Incorrect	Correct
1-bit (init=N)	20	20	2	38
2-bit (init=10)	11	29	1	39
2-bit (init=01)	11	29	2	38

Table 1: Performance comparison of different branch predictors

My analysis demonstrates that two-bit predictors significantly outperform the one-bit predictor for branches with alternating patterns like BRZ L1. For more consistent patterns like BNZ LOOP, the difference is less pronounced, though the two-bit predictor with Weakly Taken initialization achieves the best overall accuracy.

4 Conclusion

This report analyzed the pipelining of instruction and branch prediction in computer architectures. Pipeline hazards were shown to significantly impact processor performance, with mathematical relationships established between loop iterations and penalty cycles. These fundamental concepts remain essential for understanding modern processor design trade-offs and optimization strategies.