TDTS10 Computer Architecture

Lab Assignment 2: Instruction Pipelining

Axel Strid (axest556)
Dennis Zakrisson (denza625)

December 9, 2024

Contents

1 Objective

The purpose of this lab is to understand how instruction pipelines function and how performance can be improved with branch prediction techniques. Specifically, the lab investigates pipeline stages and the effects of different branch predictors on execution performance.

2 Assignments

2.1 Pipeline Basics I

1. Description of what happens in each pipeline stage when the instruction is an `LB` (load byte from memory to a register). The instruction format is: `LB, rt, offset(rs)`.

   LB instructions transfer data from the main memory to a register in the CPU.

   - **rt** is the destination register (where the byte is loaded into)
   - **rs** is the base register (holds the base address for memory access)
   - **offset** is the displacement from the address in rs to compute the final memory address

   - **IF (Instruction Fetch):** The instruction is fetched from the main memory using the program counter (PC), and the PC is updated to point to the next instruction
   - **DI (Instruction Decode):** The instruction is decoded to identify it as a load byte (LB), the base register (rs), the destination register (rt), and the offset are identified.
   - **CO (Calculate Operand Address):** The effective address is calculated by adding the offset to the value in the base register (rs).

     Using the function:

     $$\text{Effective Address} = \text{Value in } rs + \text{offset}$$

     This gives the memory address where the byte will be located in the main memory.
   - **FO (Fetch Operand):** The effective address calculated in the CO stage is used to access the main memory. The byte at the calculated address is fetched from memory.
   - **EI (Execute Instruction):** Process the data, e.g., prepare it for writing to the destination register. The byte fetched from memory is store temporarily, ready to be written into the destination register **rt**.
   - **WO (Write Operand):** The fetched byte is written into the destination register (rt).

2. Description of what happens in each pipeline stage when the instruction is an `ADD` (add the values in two registers and store the sum in another register). The instruction format is: `ADD rd, rs, rt`.

- **rd** is the destination register, where the result (sum) will be stored.
- **rs** is the first source register, which contains one of the values to be added.
- **rt** is the second source register, which contains the other value to be added.

The ADD instruction performs the operation:

$$\text{rd} = \text{rs} + \text{rt}$$

- **IF (Instruction Fetch):** The processor fetches the instruction ADD rd, rs, rt from memory using the program counter (PC). The PC is incremented to point to the next instruction.
- **DI (Instruction Decode):** The fetched instruction is decoded, identifying it as an ADD operation. The source registers (rs and rt) and the destination register (rd) are determined. The rs and rt registers are read from the register file, so their values are ready for the next stage. Control signals for the ALU (Arithmetic Logic Unit) are generated to specify an addition operation.
- **CO (Calculate Operand Address):** Since ADD is an arithmetic operation and not a load/store, there is no need to calculate an effective address (this step is more relevant for memory access instructions like LB). However, the stage is still part of the pipeline.
- **FO (Fetch Operand):** For the ADD instruction, the operands are already available in registers rs and rt after the DI stage. Since there's no memory access involved, this stage is effectively skipped.
- **EI (Execute Instruction):** The actual ADD operation is performed in this stage. The processor adds the values from the rs and rt registers together. The result is stored temporarily, ready to be written back to the destination register rd in the next stage.
- **WO (Write Operand):** The result of the addition (sum of rs and rt) is written into the rd register, which holds the result of the operation.

Conclusion: The `ADD` operation is the only instruction of these two pipelines that *changes the value.*

2.2 Pipeline Basics II

1. Timeline diagram of a two-stage pipeline (fetch, execute) of 4 instructions. A single instruction takes 6 time units as well. We assume that fetch takes 3 time units, and execute 3 time units. Since fetching and execution are two seperate operations, they can happen simultaneously. This means that the next instruction can start its fetch process as soon as the current instruction's fetch process is over. Thereby the second instruction starts its fetch process when the current instruction starts its execute process.

Time Units	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Instruction 1	F	F	F	E	E	E									
Instruction 2				F	F	F	E	E	E						
Instruction 3							F	F	F	E	E	E			
Instruction 4										F	F	F	E	E	E

Diagram representing the flow of instructions through the pipeline

Answer: 4 instructions take 15 time units.

2. The 2nd instruction is a **conditional jump**. Assume a **taken predictor** is implemented and it makes a wrong prediction. This means that the prediction was **never taken**.

Time Units	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Instruction 1	F	F	F	E	E	E												
Instruction 2				F	F	F	FL	FL	FL									
Instruction 3										F	F	F	E	E	E			
Instruction 4													F	F	F	E	E	E

Diagram representing the flow of instructions through the pipeline with conditional jump and the taken predictor being incorrect

Conclusion: We see that if our taken predictor makes a wrong prediction we have to flush out the system and wait to fetch the correct information. This delays the process of our instructions.

2.3 Branch Prediction

1. Report the accuracy of each predictor and perform a comparative evaluation based on the results obtained from the simulation.

Branch Predictors

- **Taken:** The simplest branch predictor that always predicts a branch will be taken, regardless of the actual branch behavior. This predictor is fast but generally inaccurate.
- **Bimodal:** A two-bit counter-based predictor where each branch is mapped to a table of counters. The table entries are used to predict whether a branch is likely to be taken or not based on the counter's state. It provides better accuracy than the Taken predictor but is still relatively simple.
- **2-Level (Local):** This predictor uses both global and local branch history to predict the outcome of branches. It utilizes a history table and a pattern history table to track and predict branch behavior, resulting in more accurate predictions compared to Bimodal. This predictor can be more complex but generally improves prediction accuracy.
- **Perfect:** An idealized branch predictor that always makes the correct prediction. This predictor is used as a benchmark for comparing the accuracy of other branch prediction schemes, but it is not practical in real systems.

Metrics for Accuracy Comparison

- **Address Prediction Accuracy:**

$$\text{Address Prediction Accuracy} = \frac{\text{bpred\_PREDICTOR.addr\_hits}}{\text{bpred\_PREDICTOR.lookups}}$$

Indicates how well the predictor correctly predicts the branch address. Higher values indicate better accuracy.

- **Direction Prediction Accuracy:**

$$\text{Direction Prediction Accuracy} = \frac{\text{bpred\_PREDICTOR.dir\_hits}}{\text{bpred\_PREDICTOR.lookups}}$$

  Measures how often the predictor correctly predicts whether a branch is taken or not. Higher values mean greater accuracy.

- **Miss Rate:**

$$\text{Miss Rate} = \frac{\text{bpred\_PREDICTOR.misses}}{\text{bpred\_PREDICTOR.lookups}}$$

  Represents the fraction of incorrect predictions. Lower values indicate better accuracy.

Table of Accuracy Metrics

Branch Predictor	Address Prediction Accuracy	Direction Prediction Accuracy	Miss Rate
Taken	0.138 (13.8%)	0.138 (13.8%)	0.301 (30.1%)
Bimodal	0.635 (63.5%)	0.651 (65.1%)	0.119 (11.9%)
2-Level	0.514 (51.4%)	0.531 (53.1%)	0.173 (17.3%)
Perfect	1.000 (100%)	1.000 (100%)	0.000 (0.0%)

Table 1: Accuracy Comparison for Branch Predictors

Analysis of Branch Predictors

The **Taken** predictor, which always predicts branches as taken, performs the worst due to its simplicity and lack of history tracking, achieving only 13.8% accuracy in both address and direction predictions. In contrast, the **Bimodal** predictor, which uses a 2-bit counter table to track branch history, significantly improves performance with 63.5% address prediction accuracy and 65.1% direction accuracy. This history-based approach allows it to adapt to branch patterns and make more accurate predictions.

The **2-Level** predictor, using both global and local branch histories, offers even more sophisticated prediction but performs slightly worse than **Bimodal** in this case (51.4% address and 53.1% direction accuracy). The additional complexity of the 2-level approach may not have aligned well with the branch patterns in this simulation, reducing its effectiveness compared to the simpler Bimodal predictor.

Thus, while more complex predictors like **2-Level** theoretically offer better accuracy, their performance depends on the branch behavior in the workload, and in this case, the **Bimodal** predictor provided the best results.

2. For each branch predictor, compute the speed-up relative to the least performant branch predictor (Hint: use the metric "sim-cycle").

Speed-Up Relative to the Least Performant Predictor

To compute the speed-up for each branch predictor, we use the following formula:

$$\text{Speed-Up} = \frac{\text{sim\_cycle of least performant predictor}}{\text{sim\_cycle of current predictor}}$$

First, we identify the least performant branch predictor by selecting the one with the highest `sim_cycle` value. In this case, the **Taken** predictor is the least performant, with 82,098,705 cycles.

Next, we compute the speed-up for each branch predictor relative to the Taken predictor:

- **Taken**:
$$\text{Speed-Up} = \frac{82,098,705}{82,098,705} = 1.0$$
- **Bimodal**:
$$\text{Speed-Up} = \frac{82,098,705}{55,359,135} = 1.483$$
- **2-Level**:
$$\text{Speed-Up} = \frac{82,098,705}{59,718,731} = 1.374$$
- **Perfect**:
$$\text{Speed-Up} = \frac{82,098,705}{46,157,736} = 1.778$$

Thus, the **Perfect** predictor achieves the highest speed-up, being 1.778 times faster than the Taken predictor, followed by the **Bimodal** and **2-Level** predictors with speed-ups of 1.483 and 1.374, respectively.