Project 1

Adriano Maron, Ankita Mohapatra, and Matt Barren

**Introduction**

The processor programmed for this project implements a hybrid of the PowerPC 604 and 620 utilizing a number of the specifications, such as delaying loads to avoid read after write hazards with stores. The program for this processor was developed in Java.

**Assumptions**

Since the programmed processor is not utilizing the same resources as the 604 and 620 PowerPC, there were several assumptions that had to be developed. Additionally, there are certain aspects that were not specified in the documentation of the 604 and 620 PowerPC, and therefore an assumption was made to handle these situations.

* Register File – All registers are initially zero
* Memory Locations – memory is stored in a hash map. Therefore, if a load occurs on a memory location that does not exist it will produce a null value. In this instance, the load value returned will be zero.
* Writeback and Commit – Since write back and commit share the same set of buses, there is a conflict between the use of the bus for each stage. In order to avoid an imbalance, priority alternates between committing values and writing values back. When one stage cannot fill all of the available bus positions, then the other stage is allowed to add values onto the bus.
* Load Store Ordering – Loads and stores effective addresses are calculated in order, to avoid potential read after write hazards.
* Branch History Table - The branch history table does not evict previous branches when a new branch conflicts with an already existing branch. Instead, the BHT retains those branches that were entered into the table when a vacant position was available.

**Implementation**

In Java classes were developed for stages and data structures, such as branch history table and reorder buffer. Below is a list of the different stages and data structures created:

|  |  |
| --- | --- |
| Stages | Data Structures |
| * Fetch * Decode * Issue * Execute * Functional Unit (*the following extend Functional Unit)*  |  |  | | --- | --- | | * + BU   + FPU   + Int0 | * + Int1   + LSU   + MULT |  * Write Result (encapsulates both writeback and commit stages) | * BHT * CDB * Instruction * Instruction Cache * Memory * PC * RegFile * RegStatus * ROBStatus * Station –retains necessary information for an instruction at a given station, such as Vj, Vk, and destination. * Status – retains information about the status of the instruction from when it is issued to when it is committed. |

Since the code is programmed in a linear fashion, the simulation starts with the very last stage, commitment, and continues to call stages until the first stage. This process is one full cycle. The iteration of the program stops when all of the following conditions are met:

* Fetch queue is empty
* Decode queue is empty
* Execute does not have any instructions
* ROB is empty (no ROB units are busy)

These conditions allow for the simulator to continue to commit, write, dispatch, and issue a program as long as there are more instructions to execute.

**Analysis and Results**

*Parameter Settings NW=NB=NC=2*

Setting the issue, buses, and commits to a value of two creates a bottleneck at issue, write back, and commitment because these stages cannot remove the instructions off of the decode and fetch queue at the same rate as their potential of four instructions per a cycle. In addition, the functional units in the execution stage have more values waiting to be sent to write back. This is due to having many more execution stages than could potentially be ready for write back, but the buses can only accept two at a time

A high CPI resulted from the inefficient transfer of instructions from issue to future portions of the pipeline. This can be observed from the values below

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CPI | Total Stalls | ROB Stalls | RS Stalls | Misprediction Rate |
| Benchmark 1 | 1.6756756 | 2 | 2 | 0 | 1 |
| Benchmark 2 | 1.3181819 | 0 | 0 | 0 | 0.42857143 |
| Benchmark 3 | 1.9166666 | 13 | 13 | 0 | 0.42857143 |
| Benchmark 4 | 1.1666666 | 21 | 21 | 0 | 0.75 |
| Benchmark 5 | 1.9714285 | 0 | 0 | 0 | 1 |

*Parameter Settings NW=NR=NC=4, NF=ND=2, NQ=NI=4*

For this setting, the greatest issue is the number of reorder buffers clearly plays as the bottleneck. Limiting the quantity of reorder buffers to four implies that only four instructions can be operated at once. In a CPU with six different functional units, there will always be at least two units that are not being utilized. The CPI was marginally better than the previous setting because the issue, write back, and commit were able to move more instructions through the pipeline in each cycle.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CPI | Total Stalls | ROB Stalls | RS Stalls | Misprediction Rate |
| Benchmark 1 | 1.6 | 28 | 28 | 0 | 1 |
| Benchmark 2 | 1.35 | 74 | 74 | 0 | 0.42857143 |
| Benchmark 3 | 1.3548387 | 31 | 31 | 0 | 0.42857143 |
| Benchmark 4 | 1.3417722 | 43 | 43 | 0 | 0.75 |
| Benchmark 5 | 1.6883117 | 72 | 72 | 0 | 1 |

*Parameter Settings NW=NB=NC=NF=ND=4, NR=16, and NQ=NI=8*

In this setting, the CPU had its best performance because the of having more space in queues and being able to process more instructions per a cycle. No stalls were incurred in the execution stages, but this would most likely change if there were longer operations occurring more frequently, such as division. The reorder buffers were the greatest bottleneck because the number of instructions that could be issued in a given cycle exceeded the number of commitments that were taking place.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | CPI | Total Stalls | ROB Stalls | RS Stalls | Misprediction Rate |
| Benchmark 1 | 1.1836735 | 4 | 4 | 0 | 1 |
| Benchmark 2 | 1.0070423 | 0 | 0 | 0 | 0.42857143 |
| Benchmark 3 | 1.4074074 | 26 | 26 | 0 | 0.5 |
| Benchmark 4 | 0.6875 | 22 | 22 | 0 | 0.75 |
| Benchmark 5 | 1.4255319 | 0 | 0 | 0 | 1 |

*Misprediction Rates*

The mispresdiction rate for the branch history table was greater for instruction sets that had more branches. For example, Benchmark 1 branched three times, which was not enough to allow the 2-bit branch prediction to reach a steady state of predictions. In contrast, Benchmark 2 had more iterations, which allowed the two bit predictor perform more correct predictions.

*Conclusions*

The Tomasulo approach to creating a processor allows for a more efficient execution of programs. This is due to the ability to take advantage of speculation, which allows for instructions to be completed when they are ready with their registers. Speculation also contributes to a lot of additional logic and complexity in order to ensure outputs are equivalent to an in-order processor, while having a more efficient CPI.