CS/ECE552 Spring 2019 Project Report

Team 3: Jeremy Intan, Chetankumar Mistry, Nicholas Sielicki

May 5, 2019

Contents

1 Design Overview

1.1 Description

We created a small processor implementing the WISC-SP19 ISA, which itself is a small ISA extension on MIPS, as a part of a course-long project for CS552/ECE552 in the spring of 2019.

It is implemented in a 5-stage pipelined design with full forwarding, where data memory and instruction memory are implemented as two instances of a memory system that has a 2-way set-associative 4KiB cache in front of 128KiB of total memory.

1.2 Pipeline Stages

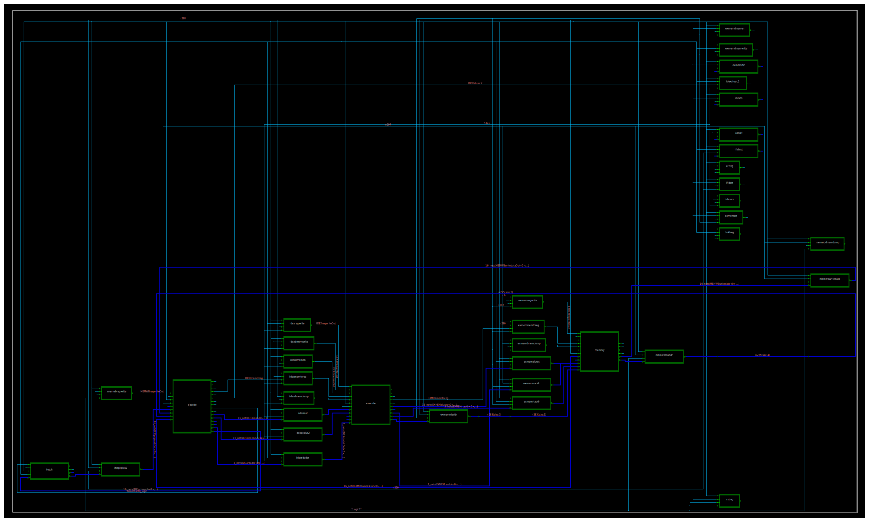

Figure 1: block view with stages clearly visible.

Fetch Contains Instruction Memory, Contains Program Counter, Outputs instructions at a given instruciton address.

Decode Inputs a given instruction and creates output signals which configure further stages for proper operation. Holds the register file. Jump targets and branch decisions are determined within this stage.

Execute Calculates addresses, Generates arithmetic and logical results with provided register contents from decode.

Memory Contains data memory, returns or stores data at a given data address.

Retirement/Writeback Writes back to the decode stage / register file.

1.3 Work Sharing

All team members contributed equally to high-level design. Jeremy had substantial experience with Verilog prior to this course, and generally speaking, contributed the most overall. Chet did much of the frontend and control work, which meant doing the prep work of translating the ISA I-formats to meaningful signals that could be used elsewhere, and that work played a major role in simplifying future design decisions. Nicholas was most active in keeping-up with the project, performing verification checks, and fixing errors when verification arose, and bringing things together.

2 Optimizations and Discussions

2.1 Optimizations Implemented

2.1.1 Carry-Lookahead Adder

A CLA was implemented to attempt to minimize the lengths of critical paths involving adders, though this did not bring us within timing constraints.

2.1.2 Least Recently Used Cache Eviction Strategy

We add an additional bit for each each line between the two caches, to save the last-used cache module. When given the choice of evicting two valid lines, this is used to evict. See Extra Credit document for more details, this has negative and positive implications.

2.1.3 Forwarding Paths

This design includes full forwarding between stages such that stalling is avoided in most cases that do not involve data memory.

Ex-To-Ex For subsequent ALU operations, our design does not need to stall such that the previous instruction progresses through the writeback stage, the result is passed immediately back into the ALU in-place of the register input, and a stall is avoided.

For example:

```
add r0, r1, r2
add r3, r4, r0
```

Ex-To-Id If the contents of a register are needed at decode, such as for the address of a jump target, we avoid the need to wait for the data to progress through memory and writeback, instead stalling just one cycle and forwarding from the ALU back to the decode stage such that the decode cycle can continue sooner. There are two instances where this is potentially necessary:

- First Instance

```
add r0, r1, r2
nop
bnez r0, TARGET
```

- Second Instance

  ```
  add r0, r1, r2
  bnez r0, TARGET
  ```

  The first instance would not need to stall, whereas the second would need to stall, but this is ultimately the same forwarding path.

Mem-To-Ex If a register is to be loaded with memory contents, we are able to forward the result from the memory stage immediately back to the ALU of a subsequent instruction, rather than waiting for the writeback stage to complete. This may either entirely remove the need for a stall, or minimize it by a cycle. In addition, if an ALU result is presently in the memory stage (but is not to be committed to memory), the current input may be forwarded back.

- Stall Once - "True" Mem-to-Ex with no intermediate instruction

  ```
  ld r0, r4, 0
  add r5, r0, r0
  ```

- Stall None - "True" Mem-to-Ex with intermediate instruction

  ```
  ld r0, r4, 0
  nop
  add r5, r0, r0
  ```

- Stall None - "Fake" Mem-to-Ex without memory operation

  ```
  add r1, r3, r7
  nop
  add r5, r1, r1
  ```

RF Bypassing If a value is simultaneously requested as the output of the register file while simultaneously being present at the input of the register file in the same cycle, ie: from the writeback stage, we do not stall while waiting for the result to be written, instead gating around the register file entirely and proceeding immediately.

2.2 Reasonable Potential Improvements

2.2.1 Branch Prediction

1. loop-specific branching strategies We spent some time discussing the idea of a simple branch prediction strategy that would predict based

on the MSB of the immediate, ie: whether or not the jump target was positive or negative. This could be combined with a ring buffer of addresses, that, when the jump target is negative and the address is present within the ring buffer, the branch is predicted taken, and otherwise predicted not-taken. This was not implemented for obvious reasons.

2.2.2 (Cache) Critical Word First

We put a lot of thought and planning into the implementation of a cache replacement strategy that would prioritize evicting and replacing the word in the line that was responsible for the request, such that stall could be left prior to the line being fully finished and consistent, but ultimately scrapped this due to the belief that it would dramatically complicate corner cases, such as returning consistent data for operations like halting. Because of this and the lack of modularity in our cache FSM implementation, we prioritized other improvements.

2.2.3 (Cache) Prefetch On Instruction Memory

Along the same line of `Critical Word First`, we discussed the idea of attempting to saturate the instruction cache with future instructions when given the opportunity, IE: during data memory stalls. This would have been relatively straightforward, but our stalling logic was unscoped throughout the entire heirarchy, and would have needed to be refactored considerably for this to be reasonable.

3 Design Analysis

3.1 Hazards

Most of the time, forwarding prevents the need to stall. However, as listed above, in some instances Read After Write hazards cannot be avoided. However, there are some instances where it is unavoidable.

Hazard	Number of Cycles Stalled
Loads followed by branch	2
ALU followed by branch	1
Loads followed by ALU operation	1
Load followed by an intermediate instruction, followed by a branch	1

An additional consideration deals with stalls and writeback. Writeback is responsible for refreshing IDEX and EXMEM pipeline registers in the case that there is a stall in execute or memory and writeback has written the value to the register file, to keep registers consistent through the stall.

3.2 Cache Design

We designed our cache to try minimize the number of cycles per request. This is 1 cycle for a hit on a read or write (ready in the same cycle as the request is made), and 7 cycles for a "read miss" or "write miss" on a clean and valid line (ready on the 7th cycle after the request is made).

For "read miss"/"write miss" on a dirty valid line, we finish on the 9th cycle after the request is made.

In the case of a hit, the cache will return the value in the same cycle

Note that when traversing words within a given line, we always access memory with an offset ordering of `000`, `010`, `100`, and `110`, (ie: the 0th, 1st, 2nd, 3rd words). When doing line replacement, we always replace all four words in the line to create a valid line, then write the incoming value to create a dirty line, rather than reading just the 3 values from memory that are needed for the line, and using the input value that caused the request for the 4th word. In other words, on evicting writes, we fully replace a line and then subsequently do a write on a clean line, rather than combining the actions.

For miss on a clean line, we spend the next 4 cycle (after the request cycle) to read from memory the content of that line. At the 4th, 5th, 6th, 7th cycle, the data from memory will be ready, so we place it in the cache.

We also use the data from memory to give it to data_{out} if it is a read request (save in buffer for the first 3 read so it's not lost). For a write request, we simply ignore one of the word from memory, and use the data instead

For miss on a dirty line, we still spend the next 4 cycle reading from memory, but we also read from cache at the same time (from offset 000, 010, 100, 110). The data from the cache is saved to registers. At the 4th and 5th cycle, data from memory is ready, but cache is still being use for read, so we saved those to some registers (we also save the output to a different register if it's the requested offset word). The next 4 cycles will be spent on writing the dirty line back to memory (saved to registers in the previous 4 cycle), and saving the new requested line to cache. First, we know in 6th and 7th cycle data from memory is ready, SO like the clean miss, we save to cache and also save to a register if it's the requested line / use data in if it's a write on one of them. The next 2 cycle, we will use the stored word in the register that was saved from 4th and 5th cycle (again using data in if it's a write on one of them).

4 Conclusions and Final Thoughts

4.1 RTL is Opinionated

Much of the difficulty we faced in synthesizing comes from the hierarchy we have within our modules, which is difficult to deconstruct in a meaningful way. In the future, if we could do this project again, we would have made an effort to clean up and refactor modules after they were implemented correctly, rather than assume that the synthesis and compilation software would be able to make short work of our inefficiencies.

In particular, this DesignWare Technical Bulletin was illuminating to us. Small changes in the way that the `RHS` of a Verilog statement is written can have a large impact on the degree of simplification and consolidation that it is able to perform.

4.2 Cache FSM

Our Cache FSM was kept simple in terms of state count, and we employed counting registers to combine states. In the future, we would not have been so lazy, and expanded all states out, for the sake of making it easier to modify in the future. This played a big role in preventing us from being able to implement optimizations we had thought about.

In general, we would have liked to have spent more time in the design phase of the cache, rather than jumping into it before we had details fully fleshed out.