Spring 2004 CS 152
Final Project

8-Stage Deep-Pipelined MIPS Processor

Members:

Otto Chiu (cs152-ae)
Charles Choi (cs152-bm)
Teddy Lee (cs152-ac)
Man-Kit Leung (cs152-al)
Bruce Wang (cs152-am)

Table Of Contents

0. Abstract
1. Division of Labor
2. Detailed Strategy
   - 2.1 Detailed Strategy
   - 2.2 Stage Summary
   - 2.3 Stage Details
   - 2.4 Forwarding Paths
3. Testing
   - 3.1 General testing.
   - 3.2 Victim Cache test.
   - 3.3 Branch Predictor.
   - 3.4 Signed/Unsigned Multiply/Divide.
4. Results

5. Conclusion
6. Appendix I (Notebooks)
7. Appendix II (Schematics)
8. Appendix III (Verilog Files)
9. Appendiex IV (Testing)
10. Appendiex V (Timing)

0. Abstract

The goal of our final project was to improve the performance of the 5-stage pipelined processor from previous labs. Aiming at this goal, we converted our processor into a 8-stage deep-pipelined one (22 pt.). Since an increase in the number of branch delay slots is an intrinsic drawback to adding more pipeline stages, we decided to add a branch predictor to cut down the number of stalled cycles in most cases (8 pt.). This problem also appears during a `jr` instruction. Thus, we installed a jump-target predictor (8 pt.) to abate the stalls associated with the instruction. In addition, we implemented a write-back cache (7 pt.) and added a victim-cache (4 pt.) to minimize first-level cache miss penalties. Finally, we added to our multiplier/divider the ability to handle signed numbers (4 pt.). Our project implemented a total of 53 pt. out of the required 37.5 pt. for our group.

We implemented our design successfully and thoroughly. We noted significant improvement in performance. The final clock speed for our processor is 27 MHz.

1. Division of Labor

The project specifications allowed us to split up the work by the major components: branch predictor, write-back cache, victim cache, jump-target predictor, and signed multiplier/divider. The entire group was involved in changing and verifying the existing design. Here is a more detailed division of labor:

Otto: Branch predictor, signed `mult/div`
Charles: Branch predictor, memory controller
Teddy: Datapath, `mult/div`, testbenches
Man-Kit: Write-back cache, victim cache
Bruce: Write-back cache, jump-target predictor

2. Detailed Strategy

2.1 Datapath.

We noticed from lab 5 that our critical path involves the memory components. Thus, we knew that we would need to concentrate in splitting the memory paths. From the timing analyzer for lab 5, we saw that it took more than 10ns to perform lookup on memory. In order to achieve the 28ns cycle time requirement, we began by splitting up each memory stage because the timing analysis tool told us that those stages together with forwarding paths took significantly more time than other stages. The idea here is to progressively split up the stages with long critical paths. Since a lot of work is involved in splitting a stage, we cut the critical paths by shifting components across stages whenever possible as an alternative. By doing this, we potentially introduced extra cycle delays, but this is partially remedied by the higher clock speed. We have split up our pipeline into the following stages:

IF	PR	ID	EX	MR	MW	WB	FW

Figure 1: Pipeline Stages

2.2 Stage Summary

IF: instruction fetch
PR: predict and register
- Branch predict
- Jump-target predict
- Register file read/write

ID: decode
- Mult/Div

EX: execute
- Resolve branches

MR: memory read
MW: memory write
WB: write back
FW: forward
- forward WB value to ID stage

We initially split up our memory stages to a tag-lookup stage and a data-lookup stage. However, we soon moved the data-lookup parallel to the tag-lookup and registered the data in the MW stage because we found that the critical path happened with data-lookup + forward logics. By moving data-lookup one stage earlier we can split up data-lookup from the forwarding logics. In addition, the routing time is shortened.

2.3 Stage Details

IF:
The instruction is fetched in this stage. We simultaneously look up the tag file and the