# Project Final Report: Pipeline Mechanics

## Abstract

## Pipelining and Forwarding

In order to improve the performance of a modern processor, designers attempt to leverage instruction level parallelism to increase the number of instructions executed in a given clock cycle or reduce the amount of time between each instruction. Pipelining is a possible solution to this problem, where instructions are broken down into multiple stages. A basic MIPS64 pipeline is shown below, where registers are added to create the various pipeline stages.



Figure 1: Basic MIPS RISC Pipeline [1]

Using pipelining, more than one instruction can be fed into the pipeline allowing at most one instruction to be completed at the end of each clock cycle once the pipeline is filled. However, various structural and data hazards keep this maximum performance from being reached. For instance, consider a set of instructions that suffer from a read after write dependency, where the following instruction depends on an output generated by the previous instruction.

The following code snippet illustrates the problem of a true ALU data dependency.

DADD R1, R2, R3 ;Put something in R1

DADD R4, R1, R5 ;Dependency on R1

Figure 2: R.A.W. ALU dependency

In the standard non-optimized MIPS pipeline, a stall would be needed such that the first instruction can complete before the second can be resumed. To improve the efficiency of the pipeline, a technique known as forwarding was developed. By passing the result of the first instruction back into the EX stage of the pipeline, the second instruction can execute right away, and no stall is required.

The following image shows the standard forwarding layout of a MIPS machine [1].



Figure 3: Forwarding in the MIPS pipeline

### The MIPS64 EDU environment

The MIPS64 EDU environment was the platform of choice to examine the various functional differences of the MIPS64 pipeline [2]. This java based simulator provided a set of tools to examine the pipeline, including where instructions are stalled, the contents of the registers, and the overall structure of memory. Using all of these tools the performance of the pipeline was evaluated as changes were made to its structure.

## Methodology

Since forwarding improves the performance of pipelining by solving problems read after write problems, where is the best place in the pipeline to implement forwarding and for what cases will this improve performance?

When a true data dependency occurs on an ALU instruction, such as the case in figure 2, a forwarding path from the output of the EX stage back into the EX will resolve the problem of needing to stall. Similarly, when a true data dependency occurs on a Load or Store instruction, forwarding from the MEM stage back into the EX or MEM stage will resolve the problem of needing to stall.

### Modifications

In order to determine the impact on performance of each of these forwarding paths, the MIPS64 EDU simulator was modified. These modifications allowed the user to select whether ALU instructions, Load and Store instructions, or both are forwarded through to the next stage. This allowed each paths impact on the pipeline to evaluated.

The following image shows the additional options made available to the user after modifying the simulator.

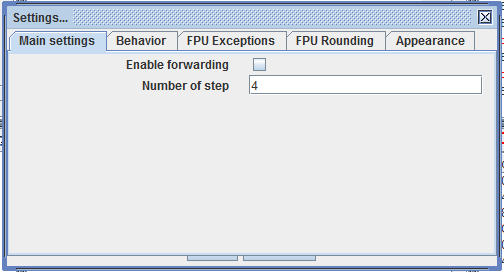
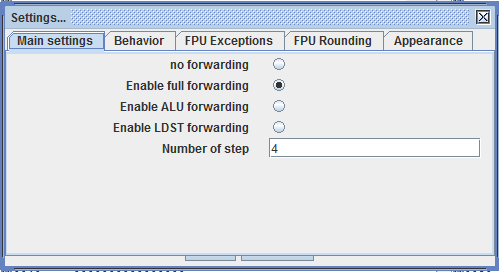
 

Figure 4: Original versus updated eduMIPS64 settings dialogue

#### The EX to EX path

EXPLANATION OF HOW THE CHANGES WERE MADE

#### The MEM to MEM path

EXPLANATAION OF HOW THE CHANGES WERE MADE

### Test Cases

In order to test if the pipeline will stall on a read after write (RAW), the test bench should be able to generate this type of error. After implementing forwarding, these test cases can be run again, and the number of stall cycles compared for each type of pipeline.

Two RAW data hazard test benches were written. The first creates a true data dependency between two registers that can be solved by forwarding to the EX stage. The second creates a dependency between two registers needed for a load and store operation, which can be solved by forwarding from the EX stage to the MEM stage and from the MEM stage to the EX stage. For the second test case, since memory is being used, there will still be at best one RAW stall.

Also, to test how the pipeline performs using a large mix of instructions, a MIPS64 compiler tool chain was used to created assembly language benchmarks from C programs. The Sourcery Code Bench tool chain provides an ANSI compliant C compiler that provides MIPS64 assembly code [3]. These programs are made up of a variety of instructions that simulate the operation of a real program.

### Evaluation

Each benchmark was evaluated for all three possible cases of forwarding, where either no forwarding was used, ALU forwarding was used, Memory forwarding was used, or full forwarding was used. The number of read after write (RAW) stalls, structural stalls, cycles, instruction count, and clock cycles per instruction (CPI) was noted. The code for each test bench is provided for the reader’s convenience in Appendix A.

The following graph shows the read after write stalls and clock cycles per instruction for the first data hazard test bench, see Appendix A item 1 for test bench code.

Based on the graph, we can see how the various forwarding stages affect performance. When there is no forwarding, two read after write stalls occur and has a CPI of 1.727, including the overhead needed to setup the computation. However, if ALU forwarding is enabled, no RAW stalls occur, and the CPI is reduced to 1.545. If only the memory forwarding path is enabled, no performance improvement is observed. Finally, full forwarding improves the performance to the same extent as ALU forwarding.

REASONS FOR THIS?

The following graph shows the read after write stalls and clock cycles per instruction for the second data hazard test bench, see Appendix A item 2 for test bench code.

ANALYSIS OF GRAPH

# Conclusion

# Appendix A: Test Benches

## ALU Data Hazard Test Bench:

;Authors: Dirk Dubois, Chris Morin

;Date: March 28th, 2013

;File: dataHazzardTestBench1

;Data Hazzard Test Bench that shows a true RAW dependencie in the EX stage

.data

;Empty not playing with memory

.code

;Prepare registers

DADDI R2, R2, 5

NOP

NOP

DADDI R3, R3, 5

NOP

NOP

DADDI R5, R5, 5

NOP

NOP

;Do Useful work

;Without forwarding 2 RAW stalls expected due to R1 dependency

;Adding forwarding from EX to back to EX will solve this stall

DADD R1, R2, R3 ;Put something in R1

DADD R4, R1, R5 ;Dependency on R1

SYSCALL 0 ;Exit program

## Load Store Data Hazard Test Bench:

;Authors: Dirk Dubois, Chris Morin

;Date: March 28th, 2013

;File: dataHazzardTestBench2

;Data Hazzard Test Bench that shows a true RAW dependencie in the mem stage

.data

.code

;Prepare registers

DADDI R1, R0, 0 ;Put a location in R1

NOP ;Remove stalls for RAW when setting up the store

NOP

SD R0, 0(R1) ;Store a value of 0 at R1 location

DADDI R2, R2, 3

DADDI R3, R3, 3

;Let the init instructions complete so the pipe is empty

NOP

NOP

NOP

NOP

;Do Useful work

;Without forwarding 4 RAW stalls expected due to R1 dependency

;2 RAW for the LD and 2 RAW for the SD

;Adding forwarding from EX to back to EX will solve this stall

DSUB R1, R2, R3 ;Put something in R1

LD R4, 0(R1) ;Dependence on R1 needs forwarding from EX to MEM

SD R4, 0(R1) ;Dependence on R1 needs forwarding from MEM to EX

SYSCALL 0 ;Exit program

## Mixed Instruction Test

# Works Cited

|  |  |
| --- | --- |
| [1] | D. A. P. John L. Hennessy, Computer Architecture A Quantitative approach, Waltham: Elsevier, 2012. |
| [2] | A. Spadaccini, "EDUMIPS64," 2007. [Online]. Available: http://www.edumips.org/wiki/About. [Accessed April 2013]. |
| [3] | Mentor Graphics, "Sourcery CodeBench Lite Edition," 2013. [Online]. Available: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/. [Accessed April 2013]. |