March 15, 2013 ECSE 425: Computer Organization and Architecture Dirk Dubois 260371224
Group 11 Christopher Morin 260344722

Project Final Report: Pipeline Mechanics

Abstract

In order to determine the effect of forwarding on the standard MIPS64 pipeline, the EduMIPS64 simulator was altered to perform forwarding of only ALU instructions, only Memory instructions, or both. These modifications were implemented by adjusting write back permissions to the simulators shared register file with semaphores. The modified pipelines were then evaluated on their performance using a set of standardized benchmarks. After comparing the various forwarding paths, it is clear that full forwarding provided the best performance increase than ALU or Memory paths alone.

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.

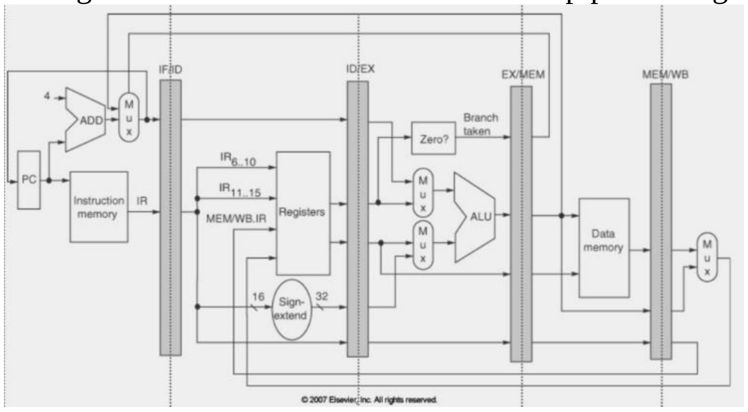


Figure 1: Basic MIPS RISC Pipeline [1]

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].

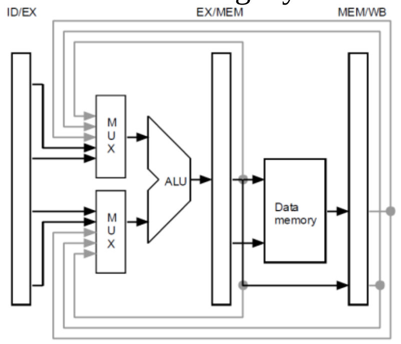


Figure 3: Forwarding in the MIPS pipeline

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

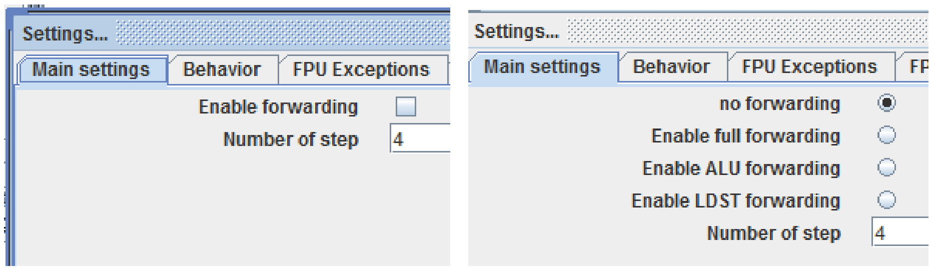
Since forwarding improves the performance of pipelining by solving problems read after write problems, where is the best place in the pipeline to implement forwarding and for what cases will this improve performance?

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.



Enable forwarding ☐
Number of step 4

no forwarding ●
Enable full forwarding ○
Enable ALU forwarding ○
Enable LDST forwarding ○
Number of step 4

Figure 4: Original versus updated eduMIPS64 settings dialogue

Forwarding implementation

The eduMIPS64 simulator implements forwarding differently than how it would be implemented in a real chip. Instead of keeping track of which values need to be forwarded based on following instructions, eduMIPS writes the value back to the register file right away when the result is ready. It uses a semaphore attached to each register that is locked when a register is waiting to be written back to.

This solution is elegant, but provides challenges when attempting to modify the forwarding paths. Different kludges were used on each of the two instruction types to selectively enable forwarding. Both forwarding types used an additional Boolean that enables or disables it, which was set by the Config GUI. In both cases, it was necessary to inhibit the write back (WB) stage from occurring if forwarding was enabled for that instruction type. Instructions of the appropriate type were already written back to the register file and doing it again in the WB stage could overwrite another instruction's result. Also, releasing an already free semaphore throws an exception in java.

ALU type forwarding

ALU forwarding was implemented by examining each instruction exiting the EX stage. If the instruction was an ALU type, the results of the operation were written back to the register file and the associated semaphore was released.

LDST type forwarding

Because store values don't have any results to write back to the register file, only Loading type instructions needed to be considered. EduMIPS uses a class hierarchy to represent instruction types. Since all loading instructions work similarly, their behavior is described in a shared superclass. This class was modified, such that write back was performed in the MEM stage if the forwarding Boolean was disabled. As in the ALU case, the associated semaphore was released when results were written back to the register file.

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.

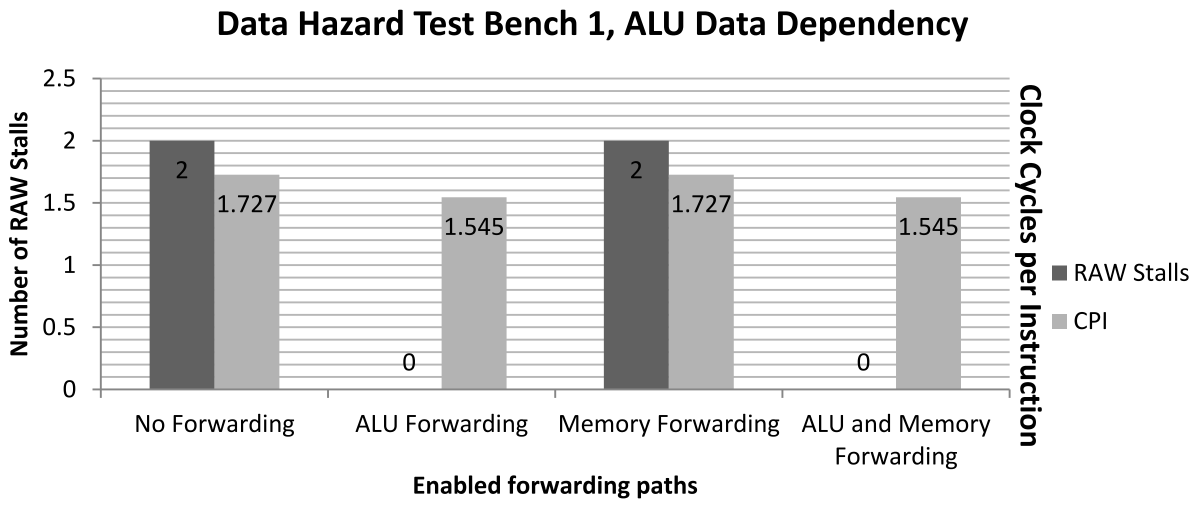
Two RAW data hazard test benches were written. The first creates a true data dependency between two registers that can be solved by forwarding to the EX stage. The second creates a dependency between two registers needed for a load and store operation, which can be solved by forwarding from the EX stage 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 bench mark that performs a simple sorting algorithm was used. Provided with the MIPS simulator, this program uses a mix of ALU and Memory instructions, as well as loops. This program simulates the operation of a real program and can be used to compare the global performance of forwarding.

Evaluation

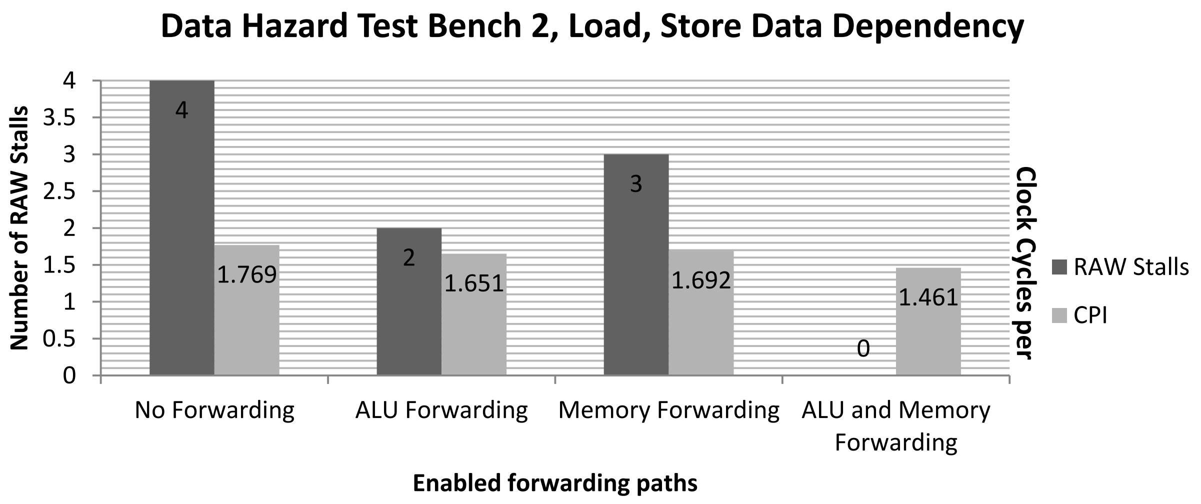
Each benchmark was evaluated for all three possible cases of forwarding, where either no forwarding was used, ALU forwarding was used, Memory forwarding was used, or full forwarding was used. The number of read after write (RAW) stalls, structural stalls, cycles, instruction count, and clock cycles per 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.

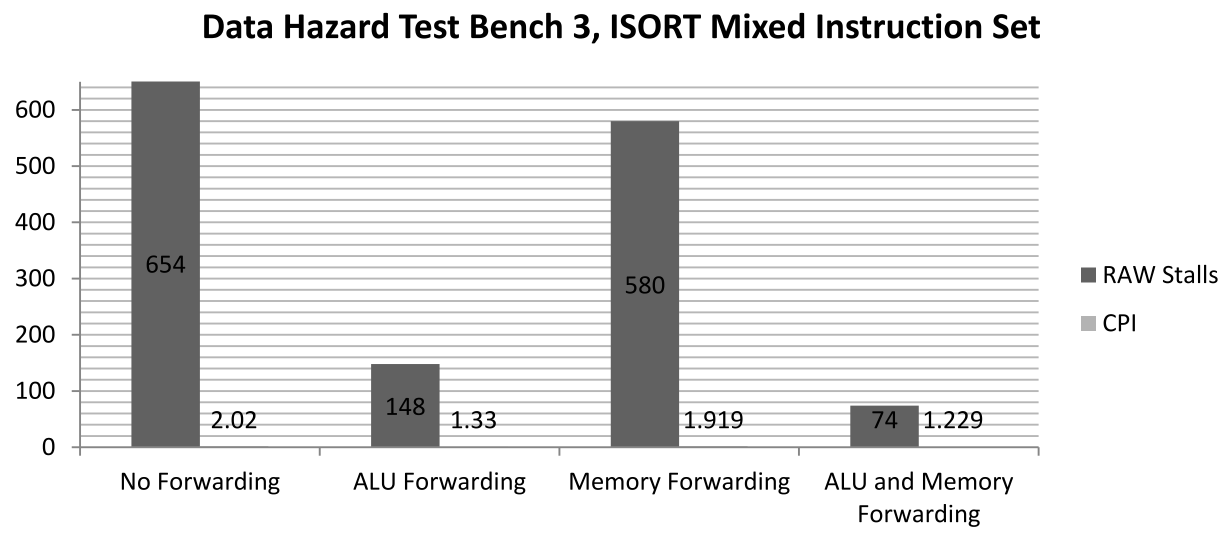


The graph above shows how the various forwarding stages affect the performance of the benchmark by comparing RAW stalls and CPI. When there is no forwarding, two read after write stalls occur and has a CPI of 1.727, including the overhead needed to setup the computation. However, if ALU forwarding is 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.

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.



When there is no forwarding, four read after write stalls occur and has a CPI of 1.769, including the overhead needed to setup the computation. However, if ALU forwarding is enabled, two RAW stalls occur, and the CPI is reduced to 1.651. If only the memory forwarding path is enabled, three RAW stalls occur and the CPI is 1.692. Finally, full forwarding improves the performance to no RAW stalls and the lowest CPI of 1.461.
The following graph shows the read after write stalls and clock cycles per instruction for the third data hazard test bench, see Appendix A item 3 for test bench code.



When there is no forwarding, 654 read after write stalls occur and has a CPI of 2.02. However, if ALU forwarding is enabled, 148 RAW stalls occur, and the CPI is reduced to 1.33. If only the memory forwarding path is enabled, 580 RAW stalls occur and the CPI is 1.919. Finally, full forwarding improves the performance to 74 RAW stalls and the lowest CPI of 1.229.

Analysis and Conclusion

Having run the various benchmark tests, it is clear that enabling full forwarding of both ALU and MEM instruction types improves the overall performance of the system. By reducing the number of RAW stalls, the overall CPI was reduced.

Since the eduMIPS simulator forwards based on instruction, the separated forwarding paths does not perfectly replicate the style of forwarding implemented in an actual MIPS pipeline. Although this solution of forwarding based on instruction provides insight into performance gains based on the mixture of instructions, it does not accurately show the data dependencies between MEM and ALU instructions. However, it is clear from the results, that forwarding any type of instruction provides significant performance gains. Test bench 3 highlights this with a drastic reduction in RAW stalls when ALU instructions are forwarded.

Overall, the modifications to the simulator provide useful insight to how instruction mixes and forwarding can improve pipelines functionality. However, larger test benches with specific instruction mixes could have been written and tested. Also, future expansion of this project could include editing the write back structure of the simulator so the true MIPS forwarding architecture can be implemented.

Appendix A: Test Benches

1. ALU Data Hazard Test Bench:

```
;Authors: Dirk Dubois, Chris Morin
;Date: March 28th, 2013
;File: dataHazzardTestBench1
;Data Hazard Test Bench that shows a true RAW dependencies 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
```

2. Load Store Data Hazard Test Bench:

```
;Authors: Dirk Dubois, Chris Morin
;Date: March 28th, 2013
;File: dataHazzardTestBench2
;Data Hazard Test Bench that shows a true RAW dependencies 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
```

3. Mixed Instruction Test

```
    .data
vet:     .word 14,8,12,4,7,13,5,11,9,1,2,6,15,3,10
len:     .word 15

    .text
    DADDI R1, R0, 8         ;indice j = 1 (*8)
    LD R2, len(R0)          ; R2 = len
    DSLL R2, R2, 3          ; R2 = R2 * 8
for:     LD R3, vet(R1)          ; R3 = vet[R1] = vet[j]
    DADDI R4, R1, -8  ; R4 = R1 -8 (i = j-1)
while:   SLTI R5, R4, 0          ; R5 =(R4 < 0) (cioè i <= 0)
    BNE R5,R0, endw         ; if R5 != 0 goto endw (perchè i < 0)
    LD R6, vet(R4)          ; R6 = vet[R4] = vet[i]
    SLT R7, R3, R6          ; R7 = (R3 < R6) = (vet[j] < vet[i])
    BEQ R7,R0, endw         ; if R7 == 0 goto endw
    DADDI R4, R4, 8         ; R4 += 8 ( i++)
    SD R6, vet(R4)    ;vet[R4] = vet[i] = R6 (ma i è stato incrementato)
    DADDI R4, R4, -16 ;R4 = R4 - 16 ( i -= 2)
    J while                 ; goto while
endw:    DADDI R4, R4, 8 ; R4 += 8
    SD R3, vet(R4)          ;vet[R4] = vet[i] = R3 = vet[j]
    DADDI R4, R4, -8  ; ...resettiamo i
    DADDI R1, R1, 8         ;indice j++
    BEQ R1, R2, end         ;il ciclo è finito?
    J for             ;un'altro ciclo
end:     HALT
```

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].