ELEC3608: Pipelined Cycle Processor Part 4

Lawrence Wakefield

10/10/15

Contents

1 Introduction

In designing a pipelined processor we enable multiple instructions to execute in the pipeline. As a result, the peak clock frequency is much higher than that of the single cycle processor hence increasing the throughput and performance of the processor.

Designing a pipelined processor introduces further complications, specifically *hazards*. *Hazards* occur when consecutive instructions rely on the completion of prior instructions. If consecutive instructions (with hazards) are in the pipeline at the same time, then a later instruction may attempt to access data that hasn't yet been written.

In the previous part, we resolved hazards by inserting `nop` instructions to wait for hazards to be resolved. In this part, we attempt to resolve hazards dynamically as well as introducing the forwarding of data.

The code is available on GitHub.

2 Dynamically Handling Branch and Jump Instructions

2.1 Jump Instructions

Jump instructions are not conditional and will always execute. In the last part of the assignment, we noted that a *control hazard* occurs when a jump instruction is followed by another instruction. This control hazard means that the following instruction will enter the pipeline and execute when instead we just wish to jump. To resolve this hazard in the previous part, we just inserted a single `nop` instruction.

In this part, we resolve a control hazard on a jump instruction by injecting a `nop` instruction after a jump instruction.

2.2 Branch Instructions

Branch hazards are resolved in a similar way to jump instructions, by injecting three `nop` instructions after a branch instructions.

2.3 Summary

These solutions do not improve the performance. However they do improve the programming, by allowing the programmer to not have to think about resolving hazards.

Note that a branch instruction is conditional, meaning that it does not always make a branch. With this knowledge, an optional improvement might be to attempt to predict the output of the branch. This would mean we only need to inject a single `nop` instruction instead of three. If a prediction is incorrect, then any changes will be rolled back.

3 Arithmetic Instructions

Arithmetic instructions often rely on the result of previous instructions. In this part of the assignment we are able to resolve these hazards using forwarding of data. Forwarding from the MEM stage to the EX stage is as follows:

1. Check if the instruction in the MEM stage is arithmetic
2. If the instruction is arithmetic, check whether it's destination matches one of our inputs.
3. If the destination matches one of our inputs, replace the input to the ALU, with the result in the MEM stage.

With the same process, we can forward data from the WB stage. However, we will do *only after* attempting to forward from the MEM stage, as the data in the MEM stage is newer than the WB stage.

If no forwarding occurs from either the MEM or WB stage, then no hazard exists and we can continue execution without delay.

Previously a data hazard on arithmetic instructions could incur up to a total of a three cycle delay. With forwarding of data, we have removed any necessary delays.

4 Load Instructions

Similarly to an arithmetic instructions, we often want to access the result of a `lw` instruction immediately after it has been loaded. In the last part of the assignment, we were able to resolve this hazard by inserting three `nop` instructions. Again, we are able to achieve a performance increase using data forwarding:

1. Check if the instruction in the WB stage is `lw`
2. If the instruction is `lw`, check whether it's destination matches one of our inputs.
3. If the destination matches one of our inputs, replace the input to the ALU, with the result from the memory in the WB stage.

However note the following example:

```
lw $t1, $0($t2)
add $t3, $t1, $t1
```

Step	IF	ID	EX	MEM	WB
1	lw				
2	add	lw			
3		add	lw		
4			add	lw	
5				add	lw

In the above example, the `lw` instruction has not been able to load data before the `add` instruction enters the EX stage. Data cannot be forwarded back in time in step 4. As a result, a hazard still exists. We resolve this as follows:

```
lw $t1, $0($t2)
nop
add $t3, $t1, $t1
```

Step	IF	ID	EX	MEM	WB
1	lw				
2	nop	lw			
3	add	nop	lw		
4		add	nop	lw	
5			add	nop	lw
6				add	nop

Now in step 5, we are able to forward data.

5 Performance Results

Letting the Fibonacci program run, it is now able to complete in a total of 1134554 cycles. At a clock cycle of 1ns(1GHz), this would suggest the program would execute in ~0.001134554 seconds. We can compare this to the Fibonacci program from part 3 of this assignment, which completed the program in 1925949 cycles.

Given these results, we can see that our forwarded pipelined processor executes the Fibonacci program in ~58.9% of the time that the non-forwarded processor, providing a satisfying performance increase.

6 Improvements

There are still improvements to be made, especially on dynamically handling hazards. Most notably, dynamic detection of hazards on `lw` instructions is suggested. This would remove the need to insert `nop` instructions after `lw` instructions with hazards. Another notable improvement would be branch prediction, which would offer a further performance increase.

7 Appendix

7.1 Modified Fibonacci Program

```
# Patterson and Hennessy, 2nd ed. modified for single cycle processor in Chisel.
            .text
            .globl      main
main:       nop
            nop
            subu  $sp, $sp, 32
            sw          $ra, 20($sp)
            sw          $fp, 16($sp)
            addu  $fp, $sp, 28
            li          $s0, 0
l1:         li          $s1, 20
            move  $a0, $s0
            jal         fib
            move  $t9, $v0
            addi  $s0, $s0, 1
            sltu  $t0, $s0, $s1
            bne         $t0, $zero, l1
            move  $v0, $a1
            lw          $ra, 20($sp)
            lw          $fp, 16($sp)
            addu  $sp, $sp, 32
            jr          $ra
fib:  subu  $sp, $sp, 32
            sw          $ra, 20($sp)
            sw          $fp, 16($sp)
            addu  $fp, $sp, 28
            sw          $a0, 0($fp)
            lw          $v0, 0($fp)
            nop
            bne         $v0, $zero, $L2
            li          $v0, 0
            j           $L1
```

```
$L2:  li        $t0, 1
            sltu  $t1, $t0, $v0
            bne         $t1, $zero, $L3
            j           $L1
$L3:  lw        $v1, 0($fp)
            nop
            subu  $v0, $v1, 1
            move  $a0, $v0
            jal         fib
            sw          $v0, 4($fp)
            lw          $v1, 0($fp)
            nop
            subu  $v0, $v1, 2
            move  $a0, $v0
            jal         fib
            lw          $v1, 4($fp)
            nop
            addu  $v0, $v0, $v1
$L1:  lw        $ra, 20($sp)
            nop
            nop
            lw          $fp, 16($sp)
            addu  $sp, $sp, 32
            jr          $ra
```