Mark McMurtury and Cody Butler
Lab 4
11/26/19

Description:

We were tasked with creating a cpu module that could perform R-type and I-type mips instructions. The cpu must read in the mips instructions that had been encoded in hexadecimal instructions. The program should implement the add, addu, sub, subu, mult, multu, and, or, xor, nor sll, srl, sra, slt, sltu, mfhi, mflo, and nop R-type instructions. The program should also implement the lui, addi, addiu, andi, ori, xori, and slti I-type instructions.

Mips Code Tested:

```
.text
li      $2,429496730 # regs(2) <= 0.1
li      $3,10        # regs(3) <= 10
li      $4,23456789  # user input (replace with sra)
li      $1, 3

multu   $4,$2
mfhi    $4           # regs(4) <= whole part of product
mflo    $5           # regs(5) <= fractional part of product
multu   $5,$3
mfhi    $5           # regs(5) <= modulo value
sll     $5,$5,28
srl     $6,$5,4
or      $6,$6,$5

srl     $22,$5,0

add $19, $5, $3
sub $7, $3, $2
mult $2, $4
mflo $8
addu $9, $1, $6
subu $10, $4, $4

and $11, $1, $2
xor $20, $2, $5
nor $21, $3, $4
sra $12, $5, 10
slt $13, $2, $3
sltu $14, $5, $6

nop

addi $15, $5, 4
andi $16, $4, 8
xori $17, $2, 10
slti $18, $3, 20
```

Hexadecimal code:

```
3c011999
3422999a
2403000a
3c010165
3424ec15
24010003
00820019
00002010
00002812
00a30019
00002810
00052f00
00053102
00c53025
0005b002
00a39820
00623822
00440018
00004012
00264821
00845023
00225824
0045a026
0064a827
00056283
0043682a
00a6702b
00000000
20af0004
30900008
3851000a
28720014
```

Results and Discussion:

After testing each mips instruction throughly our program sets each result in its own register in memory. Our CPU incorporates a three stage pipeline where it fetches the instruction, decodes and executes the instruction, and writes back the result if prompted to. We tested our cpu using a do file and then compared our results to the assembled mips code.

We used multiple multiplexers to accurately determine where our write back signal was being generated. This was dependent upon the instruction as to where the write back signal was coming from, being lo_EX or hi_EX respectively. We also instantiate a register file and an ALU to determine and set our signals for our CPU. We created an always_comb block with instantiated signals that would be used for base case functions. We created a stall condition that would allow our program to perform the phases of the pipeline in order by stalling a stage if the previous instruction is fetching an instruction, executing/decoding the instruction, or writing back the data. Within each instruction if/if else statement we update our signals that are driven or are prompted by the instruction waiting to be executed.

When testing our code initially, we noticed an error in the program we tested. It was reading from a register that did not have any data written to it. The instruction was trying to perform a logical

right shift by 4 bits, but the register did not have any data written there so the value from the register was a don’t care value. This affected the rest of our testing as it changed our results from our expected outputs. In model sim we generated all 4,096 of our memory registers to compare the results from the simulation to our assembled mips program. Once we changed where the data was being read from for that one instruction, our outputs began matching the assembled code. We also had to ensure our rdrt_EX signal was correct, as our I-type instructions were dependent upon that signal. We had to change the signal value for mfhi, mflo, and our shift instructions to match where the values were being read from, either the rd or rt signal. We tested each instruction we wrote in our CPU to ensure the results were being accurately calculated.