Remy Ren

Using five late tokens on this assignment.

Lab Intro and Objectives

The main objective of this lab is to take the single cycle CPU we have made in the last lab and smash it together with some registers and additional logic to create a 5-stage pipeline CPU! These pipeline stages are: IF/ID (fetch and decode), ID/EX (finish decode and execute), EX/MEM (finish execution and begin memory), and MEM/WB (finish memory operations and write back to registers)!

Brief Component Design Descriptions

All components from previous lab entries will not be given a brief design description, just new ones. There are a heck of a lot of components and it'd kinda just fill space when you can just look back.

1. IF_ID, ID_EX, EX_MEM, MEM_WB: All using structural modelling. They are all basically registers which require a clock signal, passing their inputs to their outputs on the rising edge of the clock.

2. pipelinedcpu0: Uses structural modeling, combines all previous components and the single cycle CPU logic. The new logic is derived primarily from Figure 4.50 in the textbook, and partially from the Green Card for referencing exact instructions.

   It requires a clock and reset signal, but as for output it only has various debug signals.

3. sscpu_testbench: Uses structural modeling. Takes in IMEM from the assignment and allows programs to be run from there.

A Brief Explainer on My Testbenches

All testbenches are modeled using structural as they require their to be tested components in their architecture. In this case, as pipecpu0_tb is a "black box" in terms of its outputs, only giving debug outputs. It runs a program from IMEM, which then allows me to run GTKWave to be able to see signals within the CPU for debugging.

Results and Waveforms

Test Program 1: p0, testing new instructions

```
AND X9, X12, X10        10001010000010100000000110001001
LSR X10, X10, 1         11010011010000000000010101001010
LSL X11, X11, 1         11010011011000000000010101101011
ANDI X12, X12, 15       10010010000000000011110110001100
ORR X21, X19, X20       10101010000101000000001001110101
ORRI X22, X22, 15       10110010000000000011111011010110
NOP                     00000000000000000000000000000000
NOP                     00000000000000000000000000000000
NOP                     00000000000000000000000000000000
```

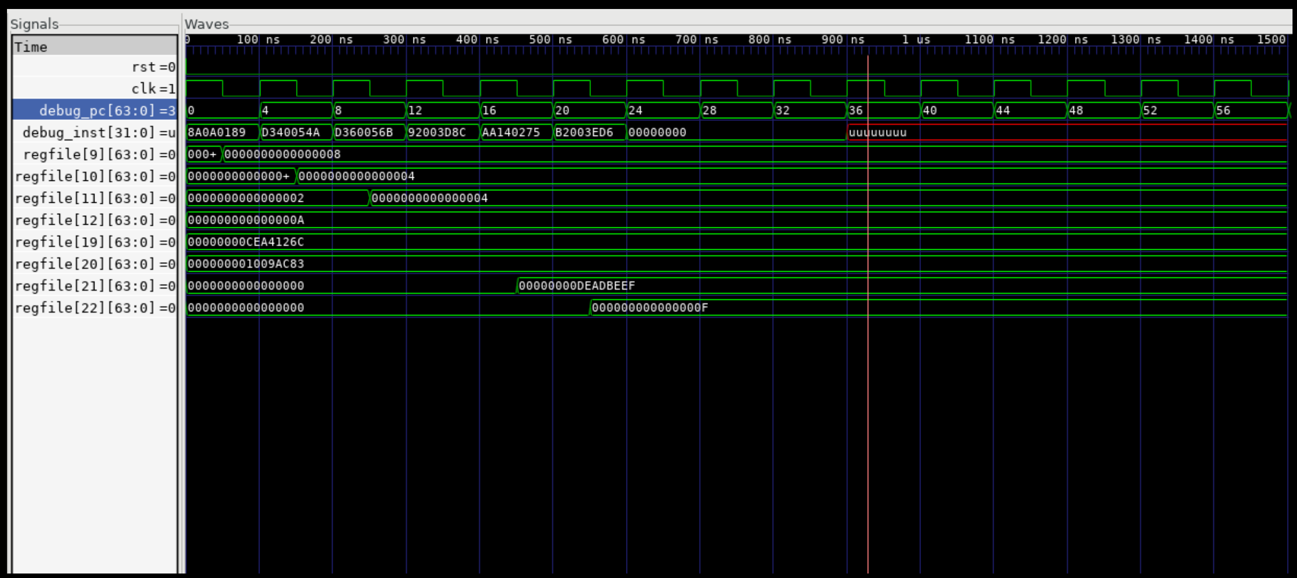

Figure 1: Testing new instructions AND, LSR, LSL, ANDI, ORR, ORRI, and NOP in GTKWave

Registers (all 64-bit, unrepesented zeros for readability) are initialized to:

- X9 = 0x10

- X10 = 0x08
- X11 = 0x02
- X12 = 0x0A
- X19 = 0xCEA4126C
- X20 = 0x1009AC83
- X21 = 0x00
- X22 = 0x00

A breakdown of the new instruction tests in **Figure 1:**

1. X9 = X12 AND X10 = 0x0A AND 0x08 = **0x08**. We can see this in regfile[9] on the first falling edge.
2. X10 = X10 (0x08) shift right by 1 bit, so **X10 = 0x04**. We can see this in regfile[10] on the second falling edge.
3. X11 = X11 (0x02) shift left by 1 bit, so **X11 = 0x04**. We can see this in regfile[11] on the third falling edge.
4. X12 = X12 (0x0A) AND d"15" (0x0F) = 0x0A, no change. We can see this on the fourth clock cycle and the fact that X12 is always 0x0A throughout.
5. X21 = X19 (0xCEA4126C) OR X20 (0x1009AC83) = 0xDEADBEEF *hahaha* We can see DEADBEEF in regfile[21] on the fifth falling edge.
6. X22 = X22 (0x00) OR d"15" (0x0F) = 0x0F, we can see this on the sixth falling edge in regfile[22].
7. 3 NOPs in a row. We can see that on the seventh clock cycle (and next two), debug_pc continues to increment by 4, but nothing is affected indicating a proper NOP. Afterwards we can see debug_inst is undefined as there are no instructions left to execute in IMEM.

As all waveforms and registers correspond with the expected results of the computation test, it is a success! In addition, p0 contains all independent instructions, so no hazards occurred.

Test Program 2: p1, testing pipelining

```
ADD X11, X9, X10          10001011000010100000000100101011 8B0A012B
STUR X11, XZR,0           11111000000000000000001111101011 F80003EB
SUB X12, X9, X10          11001011000010100000000100101100 CB0A012C
STUR  X11, [XZR,0]        11111000000000000000001111101011 F80003EB
STUR  X12, [X12,8]        11111000000000001000000110001100 F800818C
STUR  X12, [X12,8]        11111000000000001000000110001100 F800818C
ORR X21, X19, X20         10101010000101000000001001110101 AA140275
NOP
NOP
STUR X21, [XZR,0]         11111000000000000000001111110101 F80003F5
NOP
NOP
NOP
NOP
LSR X21, X19, X20         11010011010101000000001001110101 D3540275
```

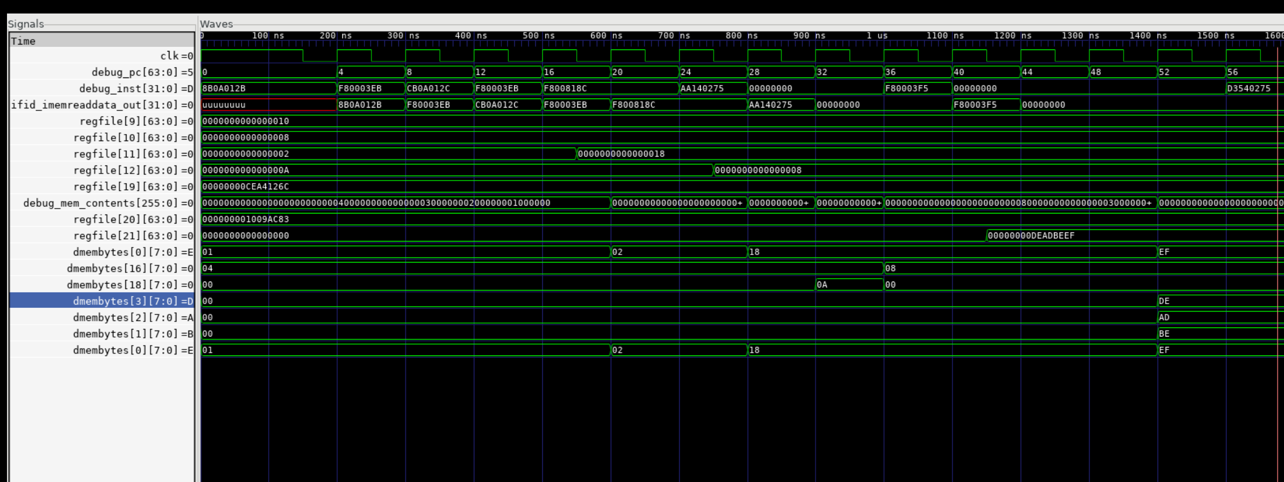

Figure 2: Pipelined CPU Test in GTKWave

Registers are initialized same as in p0, see above. Here is a breakdown of the pipelined CPU test in **Figure 2:**

1. X11 = X9 (0x10) + X10 (0x08) = 0x18. We see this during cycle 5 (debug_pc = 16) in regfile[11].

2. Store the contents of X11 in address XZR + 0 = 0x00. As this pipelined CPU has no forwarding or hazard detection, this instruction fails as the execution of this is on the fourth cycle (debug_pc = 12), yet X11 is not written back to until the middle of the fifth cycle (debug_pc = 16). As a result, DMEM, or dmembytes[0][7:0] incorrectly gets 0x02 on the fifth cycle.

3. X12 = X9 (0x10) - X10 (0x08) = 0x08. We see this occur on the 7th cycle (debug_pc = 24), where regfile[12] gets 0x08.

4. Store contents of X11 into address 0x00. Occurs on the 8th cycle (debug_pc = 28), where dmembytes[0][7:0] gets 0x18. This is a correct value, after X11 had its write back occur.

5. Store contents of X12 into address X12 (0x0A) + 0x08 = 0x12 = 0d18. Incorrect value (I'm sure about this) and address (Not so sure about this?) is loaded here as this instruction's ID/EX (decoding) occurs on the sixth cycle (debug_pc = 20) where X12 has not yet been written back to with its proper value of 0x08. We see the result of this occur on the ninth cycle (debug_pc = 32) where dmembytes[18][7:0] gets 0x0A.

   (a) Not really sure why the cycle after, dmembytes[18][7:0]=0x00... Is this an issue?

6. Store contents of X12 into address X12 (0x18) + 0x08 = 0x20 = 0d16. Correct values is loaded here as this instruction's ID/EX occurs on the seventh cycle (debug_pc = 24), after X12 already had its write back occur. We see the result of this occur on the 10th cycle (debug_pc = 36) where dmembytes[16][7:0] gets 0x0A.

7. X21 = X19 (0xCEA4126C) OR X20 (0x1009AC83) = 0xDEADBEEF. We can see this during the 11th cycle (debug_pc = 40), when regfile[21][63:0] gets this value.

8. 2x NOP instructions. We can see this properly occur on the 8th cycle (debug_pc = 28) where instruction memory is all zeros for two cycles, as well as the ifid (IF/ID Instr. Mem.) being all zeros. This is to stall

for the next non-NOP instruction which is a STUR, which is required if we want STUR to load the correct contents from X21.

9. Store contents of X21 into address 0x00. We can see this occur at the bottom of the scope, where dmembytes[3 downto 0] = 0xDEADBEEF on cycle 14 (debug_pc = 52)!

10. 4x NOP instructions. We can see this occur as like before, starting at the 11th cycle (debug_pc = 40) instruction memory and pipeline instruction memory is all zeros.

11. X21 = Shift X19 (0xCEA4126C) right by X20=0x1009AC83=0d269069443 bits on the 15th cycle (debug_pc = 56). However, this instruction fails to run. This is the result of it being the last instruction and IMEM having a max number of 64 readable bytes. Once we go beyond that, which is after the IF/ID stage, due to our pipeline CPU design, we attempt to fetch beyond the max, throwing an ERROR in IMEM, and stopping our CPU runtime as intended.

As all waveforms and registers correspond with the expected results of the computation test, it is a success!