Christopher Bann
Lab 4

Pipelined processor with no hardware hazard detection

Design

In previous labs, I built a single-cycle ARM Legv8 processor. In this lab, I modified the processor so that it can pipeline instructions. I run two tests via different instruction memory contents to test the functionality of the processor. See the following figure:

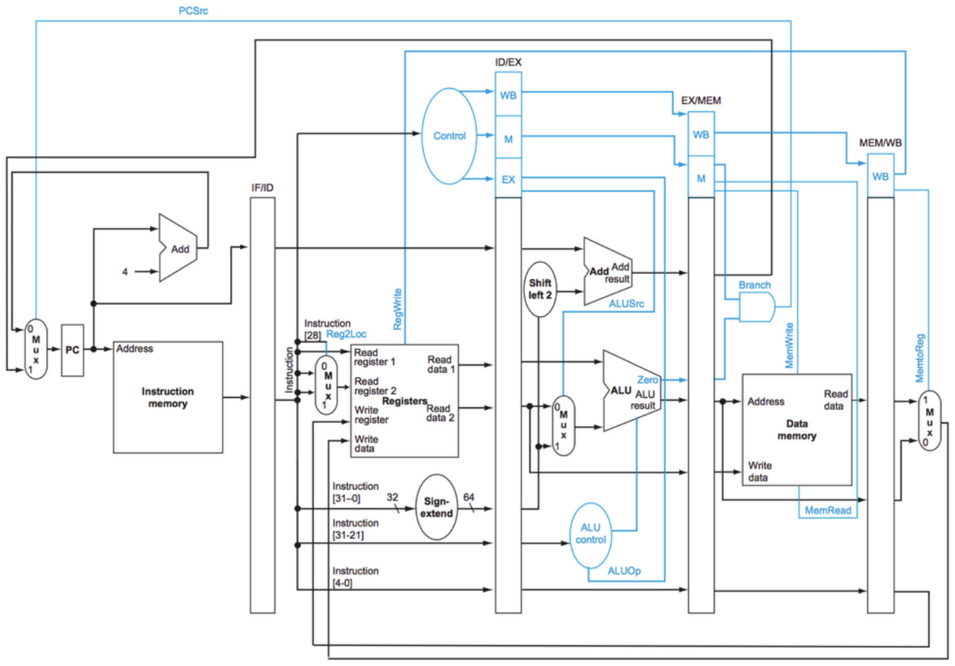

Figure 1: Schematic of Pipelined ARM (LEGv8) processor

pipelinedCPU:

The design for the processor was almost entirely based on the figure above. The key was knowing how each component changed with the addition of our pipelining registers - making sure each wire was connected properly with correct intermediate logic (if needed). Having a more detailed breakdown of each wire going in and out of the pipelining registers helped prevent any misconnections.

Besides the inclusion of new instructions in the alu control file, none of the other files from the previous single cycle cpu needed to be modified as we are only changing how the processor runs.

IF_ID:

The implementation of this register was straightforward as this first component only has two inputs as seen in figure 2. The wiring was done according to IF ID as seen in figure 1. Pc, addr, clk, and rst were the 4 inputs and pc and addr were the two outputs that continued on in the processor.
Below is the schematic of the IF/ID register:

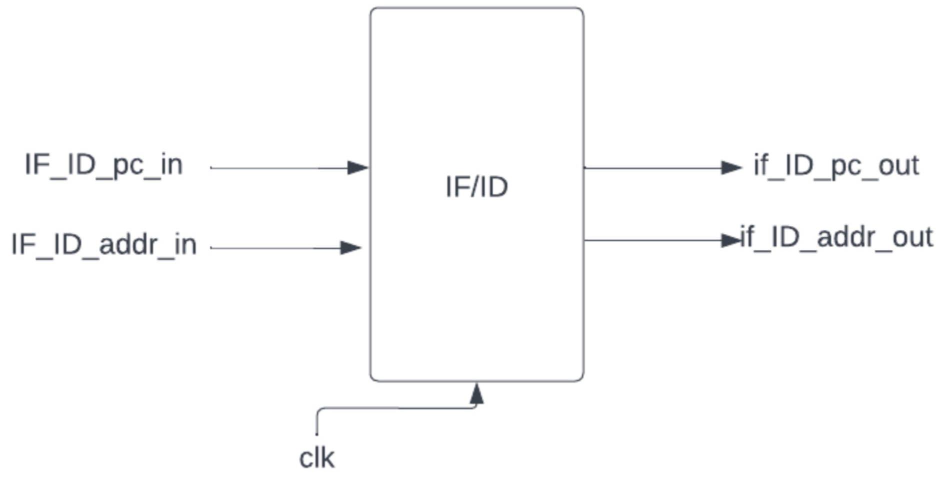

Figure 2: IF/ID Register

ID_EX:

This next implementation is similar to IF/ID as it follows the logic seen in figure 1, with the details shown below in figure 3. The same is replicated for the rest of the registers, thus will not be included in this report (EX/MEM, MEM/WB)
Below is the schematic of the IF/ID register:

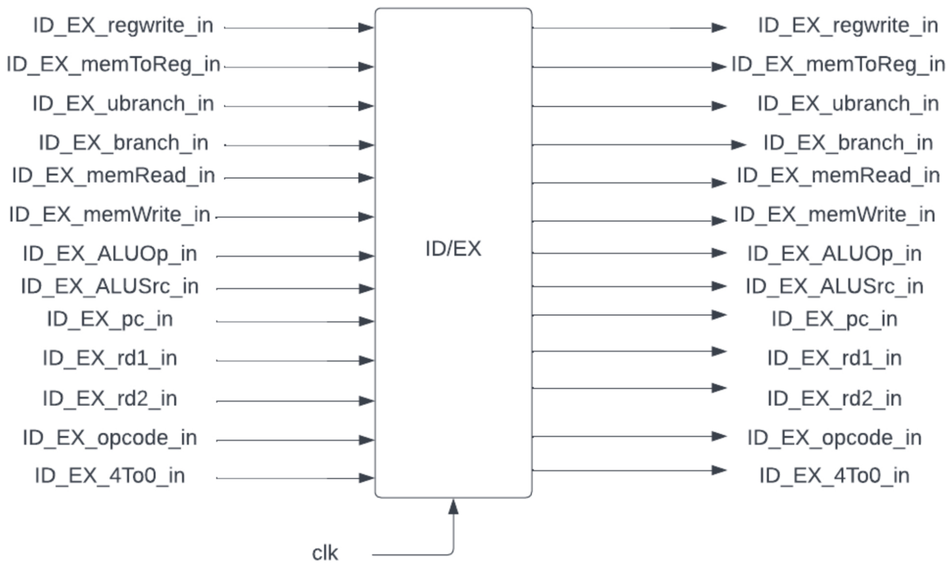

Figure 3: ID/EX Register

Results and Discussion

The testbench for pipelinedCpu was very simple. We have a clock and rest signals that change value over time. I didn't test any other modules as they were tested in previous labs.

pipelinedCPU:

We perform two different tests that test the instruction memory of the pipelined cpu. Here are the two tests performed below:

```
-- AND X9, X12, X10
-- LSR X10, X10, 1
-- LSL X11, X11, 1
-- ANDI X12, X12, 15
-- ORR X21, X19, X20
-- ORRI X22, X22, 15
-- NOP
-- NOP
-- NOP
```

p0 Test

```
-- ADD X11, X9, X10
-- STUR X11 XZR,0
-- SUB X12, X9, X10
-- STUR  X11, [XZR,0]
-- STUR  X12, [X12,8] incorrect value will be loaded
-- STUR  X12, [X12,8] correct value will be loaded
-- ORR X21, X19, X20
-- nop
-- nop
-- STUR X21, [XZR,0] correct value will be loaded
-- nop
-- nop
-- nop
-- nop
-- LSR X21, X19, X20
```

p1 Test

```
x9  = 0x0000000000000010
x10 = 0x0000000000000008
x11 = 0x0000000000000002
x12 = 000000000000000A
```

Initial register values
For simplicity purposes, I will only dive deep into the results of p1 and show the issues of not having hazard detection/fixing.

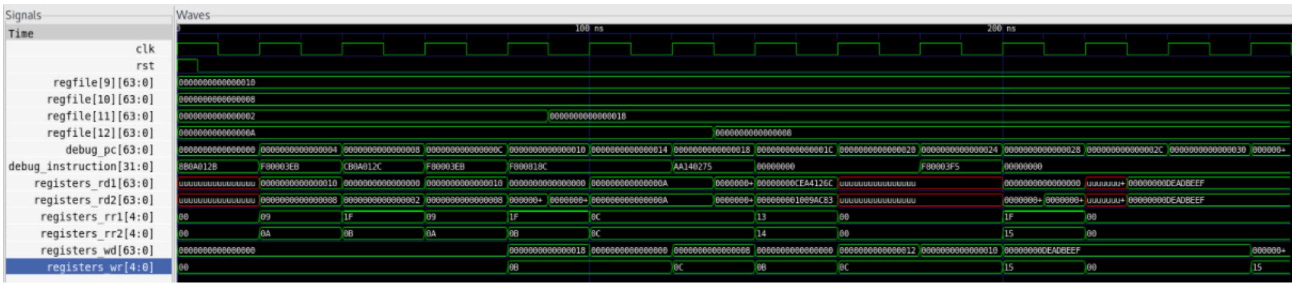

Figure 4: GTKWave of test p1

As you can see from Figure 4, the first two cycles are processed in the register before everything goes haywire and stops working because of hazards. However, if you track the other registers (rd1, rd2, rr1, rr2), you see that they continue on to produce the expected results according to the p1 test set of instructions seen above. It is also possible to observe the set of NoOps in two different sections of the program; again as seen in the instruction set given by p1. Wd and Wr are also writing to the correct registers. Instructions are updating properly and show the NoOps as well. To properly fully test that the pipelined cpu works, hazard management has to be implemented so that we can see the full results in the temp registers over time.