Implementation LEGv8: a single cycle processor that executes a subset of ARM v8-64bit ISA

Design

In previous labs, I built the components of a single-cycle ARM Legv8 processor. In this lab, I connected all the modules from before to create a single-cycle implementation of the processor. I run three tests via different instruction memory contents to test the functionality of the processor. See the following figure:

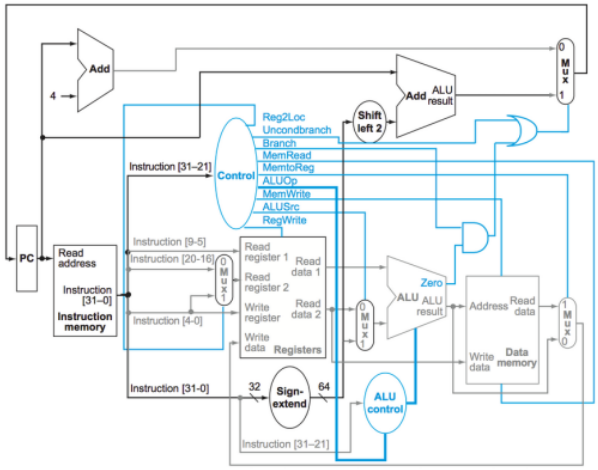

Figure 1: The ARM processor components designed in this lab

singleCycleCPU:

The design for the processor was almost entirely based on the figure above. The key was knowing how much of each component was needed - for example, you need two structural adders. Every signal had to be wired to the correct place with the correct intermediate logic.

Lots of files needed to be updated such as the CPUControl module to handle immediate instructions and the signExtend module to sign extend based on the opcodes of different instruction types. ALUControl has also been modified to accept instructions (add/subtract) for different instruction types.

CPU Control:

The following case was added to deal with immediate instructions:

```
-- If OpCode is R-Format
OutCode <= ("000") when (Opcode(10) = '1' AND Opcode(7 downto 4) = "0101"
AND Opcode(2 downto 0) = "000") else
-- If OpCode is LDUR
("001") when (Opcode(10 downto 0) = "11111000010") else
-- If OpCode is STUR
("010") when (Opcode(10 downto 0) = "11111000000") else
-- If Opcode is CBZ
("011") when (Opcode(10 downto 3) = "10110100") else
-- If Opcode is Unconditional
("100") when (Opcode(10 downto 5) = "000101") else
-- If Opcode is ADDI
("101") when (Opcode(10 downto 1) = "1001000100");
```

signExtend

We sign extend based on different instruction types. Here is a table that shows the opcodes for the different instruction types:

```
-- B    B type  000101
-- AND  Rtype   10001010000
-- ADD  Rtype   10001011000
-- ORR  Rtype   10101010000
-- SUB  Rtype   11001011000
-- LSR  Rtype   11001011000
-- LSL  Rtype   11010011011
-- =>           1----01-0--

-- ADDI Itype   1001000100
-- ANDI Itype   1001001000
-- ORRI Itype   1011001000
-- SUBI Itype   1101000100
-- =>           1--100--00

-- CBZ CBtype   10110100
-- CBNZ CBtype  10110101
-- =>           1011010-

-- STUR Dtype   11111000000
-- LDUR Dtype   11111000010
-- =>           111110000-0
```

ALU Control

The following cases were added to ALUControl to account for instructions of different types:

```
-- R-format add
("0010") when (Opcode = "10001011000") else
-- I-format add
("0010") when (Opcode (10 downto 1) = "1001000100" and ALUOp(1) = '1') else
-- I-format sub
("0110") when (Opcode (10 downto 1) = "1101000100" and ALUOp(1) = '1') else
```

Results and Discussion

The testbench for **singlecyclecpu** 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. The other modules were also tested through the implementation of the singlecyclecpu.

singleCycleCPU:

We perform three different tests that test different components of the Single Cycle CPU. The first one is a computation test shown below:

```
-- ADDI    X10, X11, 1
-- ADDI    X10, X11, 2
-- ADDI    X9, X9, 1
-- SUBI    X9, X9, 1
-- ADD     X10, X9, X11
```

Below is the gtkwave simulation of the assembly code executed above:

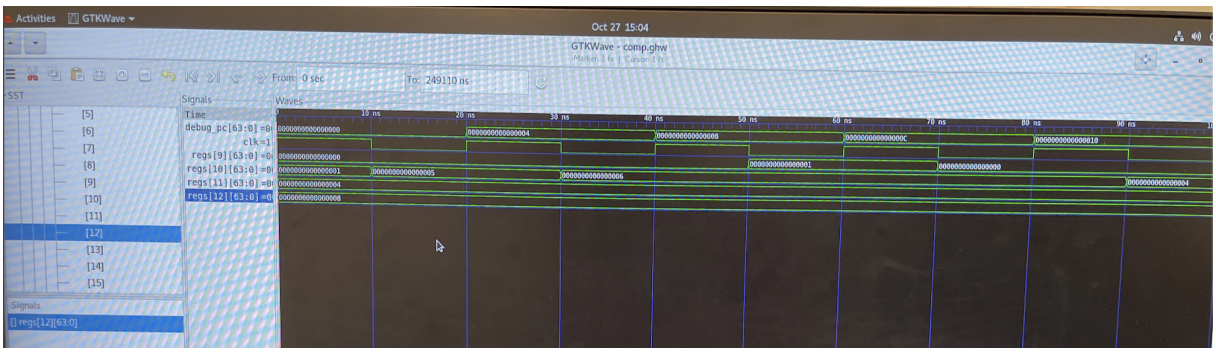

Figure 2: Throughout every clock cycle and step, all the registers have the correct value.

I'm so sorry the picture isn't clear, I'll resubmit with screenshots next time I'm in the lab. Step 1: Add X11 and '1' and store to X10 (thus X10 is value "5"). Step 2: Add X11 and '2' and store to X10 (Thus X10 value is "6"). Step 3: Add '1' to X9 (0 + 1 = 1). Step 4: Subtract '1' from X9 (1 - 1 = 0). Step 5: Add X9 and X11 together and store in X10 (0 + 4 = 4). The registers accurately depicted the steps.

The next test is a Load/Store test:

```
-- STUR X10, [X11, 0]
-- LDUR X10, [X9, 0]
```

Below is the GTKwave simulation of the assembly code executed above:

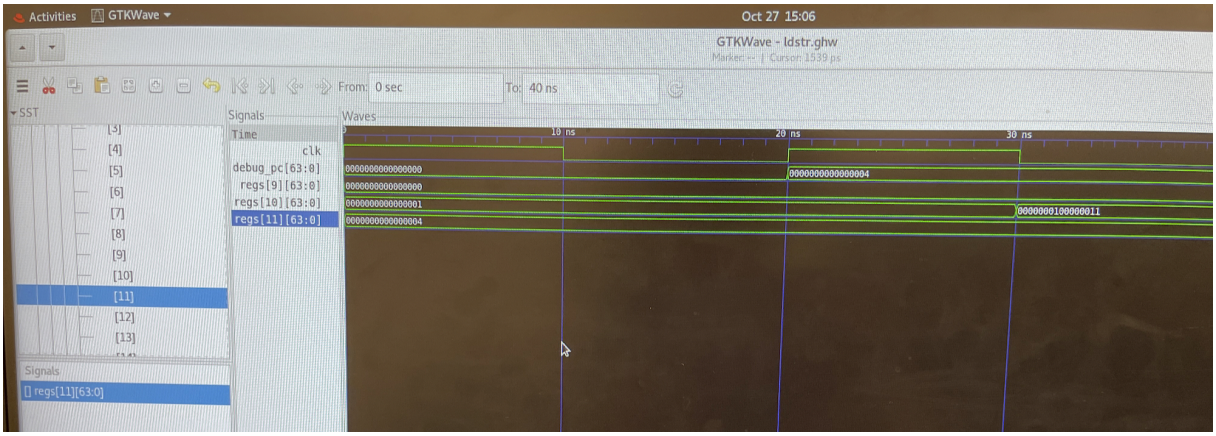

Figure 3: Throughout every clock cycle and step, all the registers have the correct value.

The address of X11 was stored in X10 and X9 address was loaded into X10, thus giving the answer "0000000100000011".

The final test combined all parts and was a full LEGv8 assembly code:

```
-- ADDI    X9, X9, 1
-- STUR    X10, X9, X11
-- LDUR    X12, [X11, 0]
-- CBZ     X9, 2
-- B       3
-- ADD     X9, X10, X11
-- ADDI    X9, X9, 1
-- ADD     X21, X10, X9
```

Below is the GTKwave simulation of the assembly code executed above:

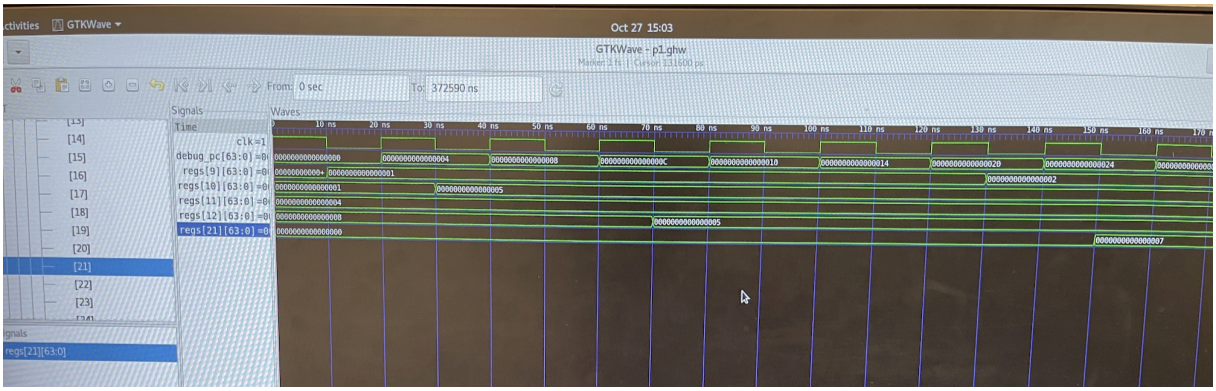

Figure 4: Throughout every clock cycle and step, all the registers have the correct value.

Going down the assembly code given, the registers accurately reproduce the expected results (even skipping instructions).

Question:

Although I do not currently have the ability to double check my answer, I believe that if X9 were -1 instead of 0, that would change the scheme of the program. First of all, the X10 would have a value of 0x04 instead of 0x05 and that would ripple down to change the values in the rest of the program.