README

Contributors:

Waynar Bocangel Calderon

Alexis Chen

Danica Xiong

Zoom Video Demo Link:

https://ucsd.zoom.us/rec/share/YZbkyfwlbsz6eD0KPfIXSp4iD44PrkMWNCpr0o38PeMHNIYEs88g8qoKX8vPaO4P.sR63niMoKm5NAV7e

Passcode:

+w!*zVi8

How to use our processor:

1. Copy the machine code of specified program (ie. Program 1, 2, or 3) from the machine_code doc. Paste the machine code into the machine_code.txt files in the simulation\modelsim folder, and the cse141LProcessor folder.
2. Run the Processor in Quartus
3. Port it to ModelSim and compile the testbench
4. Simulate our processor
5. Check the output file or console for output.

TestBenches:

We used the 3 test benches given by the Professor to prove our correctness.

Design:

Our ISA is reg-reg architecture and our design is fully synthesizable.

Features that work:

Program 1 encodes a message using LFSR patterns and saves it to memory.

Program 2 decodes the message.

Program 3 formats the message (to remove spaces) and checks for errors.

Basically everything works for every type of input!

Features that don't work:

N/A (Everything works)

Challenges:

We spent a lot of time debugging our assembly into our processor because our instructions are very complicated given the limited bits per instruction. Due to the fact that we have a 2 bit flag (Set #00) to set what types of instructions we're executing, it was very easy to get instructions mixed up. (Ie. "set #2, lsa" is add and "set #0, lsa" is load). We solved this challenge by carefully comparing our assembly code with the waveform to make sure that the values in the registers match. Very often, errors in our program were caused by setting the wrong flag for instructions.

In addition, since we only have 8 registers and our branching instructions use 2 registers to compare, we are limited to writing simple loops for our program. Originally when we were writing the programs in ARM, we had loops for setting LFSR and nested loops. However, translating those loops into our assembly instructions complicates the codes since we need to constantly move registers around and make sure that we have the correct values in certain registers for condition checks. As a result, we simplified our programs and implemented the LFSR in the hardware.

Furthermore, while using relative branching, we had to make sure to keep our operations under 32 lines backward and 31 lines forward since we only had 6 bits dedicated to branches.

Regarding hardware, we spent a lot of time importing the look up tables for the LFSR, and implementing the CTRL part (since we had a lot of instructions and had to check for both OP code and set flag). It took a while for us to get the LFSR to work because it was challenging to understand, and even harder to implement. We had to

figure out how to set the LFSR tap and make the LFSR state change without the clock. It took a long time to debug because it was implemented in hardware and had weird bugs.