Ben Trantanella

CSC270

3/13/23

Final Project Report: Track A

For my final project, I chose to create a 16-bit CPU single cycle CPU. Beyond the basic ALU operations and load/store, I chose to implement branch-if-equal, jump, set-less-than, and set-greater than, as well as replacing the ALU adder with a carry look-ahead adder that is built from four 4-bit carry look-ahead blocks. The starting point for my project was the 4-bit CPU that we had made in lab 4, with the first step being converting it to a 16-bit CPU. The next step was determining the format of the instruction word and writing the python script assembler.

For my instruction word I used a 27-bit input represented by a 7-digit hex value. The 5 most significant bits represented the operation being performed, with the specific mappings shown below:

Operation Bit Matching

Operation	Bit String
add	00000
addi	10000
and	00001
andi	10001
or	00010
ori	10010
sub	00011
subi	10011
sw	11000

lw	10100
beq	01111
j	01110
slt	01011
sgt	00111

One design decision I made while creating the operation section of my instruction word was to have the most significant bit represent whether an immediate value was being used or not, routing this bit through the control logic to a multiplexor that uses the value from read 2 data when the bit is 0 and the immediate value when the bit is 1. The next design decision that I made was to have the two least significant bits represent the basic ALU operation being performed. Beyond the first 8 operations where that is essentially the entire operation, this is useful because for lw and sw having these bits correspond to the add ALU operation allows us to add the offset to the value in the register, mimicking the address+offset design and allowing us to access more data memory locations than with just the offset. This is also useful because for the operations beq, slt, and sgt, subtracting the two values being compared allows me to determine whether A is equal to, less than, or greater than B based on the output.

The next three bits of the instruction word represented the register being written into for the majority of the operations, or the second register being read from for the sw operation. With the design of only having 8 registers, we only need three bits to reference all of them. Similarly, the following three bits represented the first register being written into. The next 16 bits either represented the immediate value being input, or the second register being written into. In the second case, the three most significant bits were the register mapping, with the final 13 bits being filled with zeros. Within the CPU these were split off from this set of bits and routed into read

address 2, and combined back with the other bits after. Having 16 bits allocated to immediate data allows me to directly insert up to the max value the registers can hold without the need to do any additional math to reach that number. It also allows me to access any data memory location possible in the 16-bit data memory being used. Through this instruction word I am able to fully represent all of the instructions I need. Below are a few examples of what the instruction word in assembly translated to hex would look like.

Example Instruction Word Mapping

Instruction Word	Hexadecimal value
addi $1 $1 1	4090001
j 2	3800002
beq $0 $1 -5	3c1fffb
add $0 $0 $1	0002000
sgt $4 $0 $1	1e02000

The next step of the project was to create the assembler which converts the instructions from assembly code into binary and then into hexadecimal, outputting the result in a .hex file which can be loaded directly into the CPU instruction memory. A description of this along with the design decisions taken and way in which the assembly code is translated to machine code is located in the README.md markdown file.

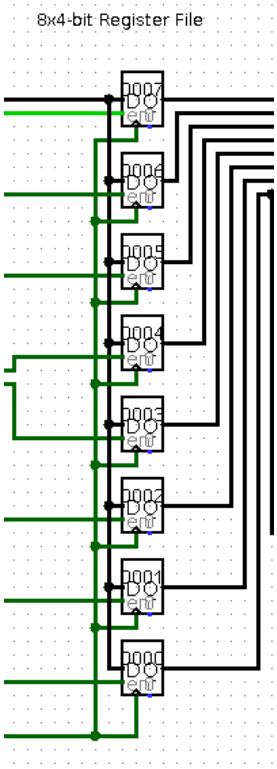
The final step was to implement the features that my CPU needed to perform, specifically branch-if-equal, jump, set-less-than, and set-greater than. First, for branch-if-equal the implementation works by subtracting the two values in the registers being compared, and if the result is zero then the branch offset from the immediate value is added to the current program counter and given to the address field of instruction memory. As explained in the readme, a noop

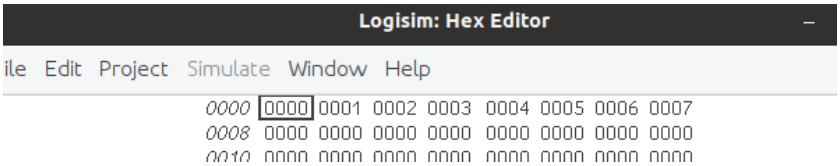
is needed after this instruction, therefore taking two clock cycles to complete. Due to this noop, one other element added to this features implementation was a mux which either outputs -1 or 1 depending on if the offset is negative or positive, adding this to the desired new program counter. Next for the jump instruction, all that is done is taking the jump offset value and adding it to the current program counter, sending this to the instruction memory address. No additional +1 or -1 is needed for this because it executes in one clock cycle and does not need a noop. Finally for the slt and sgt, once again the two values being compared are subtracted from each other. For slt, the value 1 is written to the destination register if the result is negative, with this being checked by looking at the most significant digit of the result - 1 is negative, 0 is positive. Similarly for sgt the value 1 is written to the destination register if the result is positive. The last feature implemented was the carry look-ahead adder. I chose to build this with four 4-bit carry-look-ahead blocks, as shown in the textbook.

Below are the results from the test files given as well as the test files that I wrote. Each file contains a description of the desired result as well as a rough outline of what is being done throughout the assembly code, and the short descriptions beside each picture are verification that the test result is as desired.

Provided Tests:

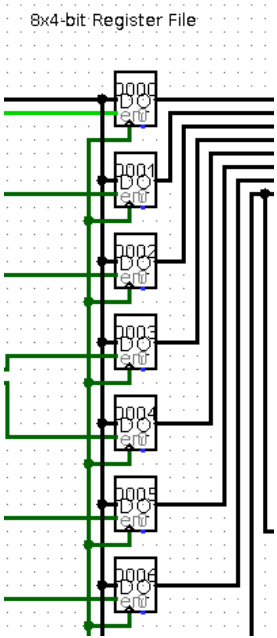
Basic-LW-SW-rev.asm:





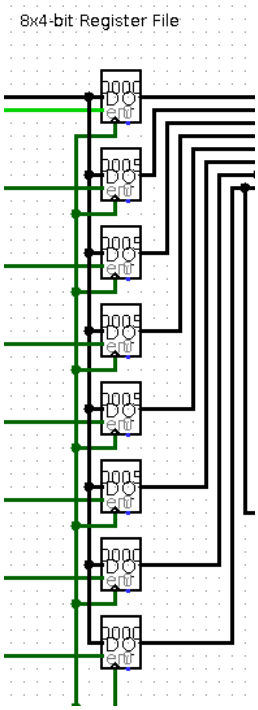
Values in descending order from 7 to 0 in registers, values in ascending order from 0 to 7 in data memory.

Basic-R-Type-rev2.asm:



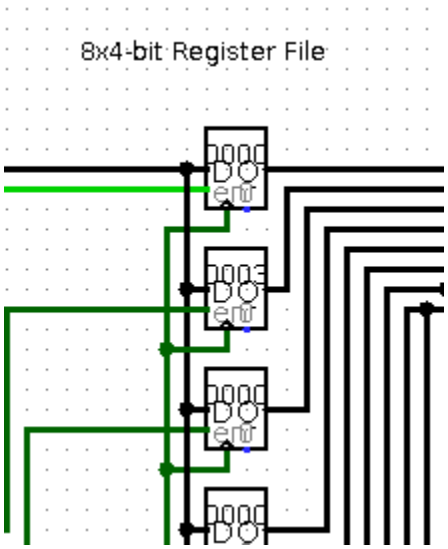
Values 0 through 7 in their respective registers.

Testbranchbackwards.asm:



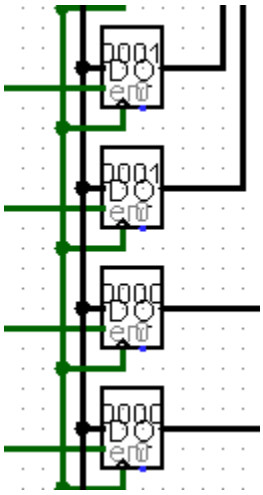
Value 5 in register one through five

testjump.asm:



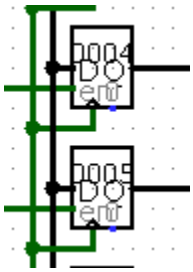
Value 3 in Register 1

test-sgt-slt.asm:



Values 1 in registers 3 and 4, values 0 in registers 5 and 6

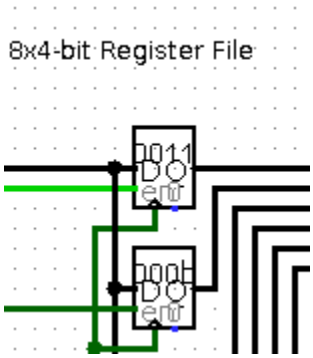
testbranch.asm:



Value 4 in register 4, value 5 in register 5

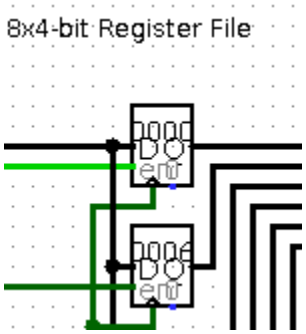
Personal Tests:

pBranchTest.asm:



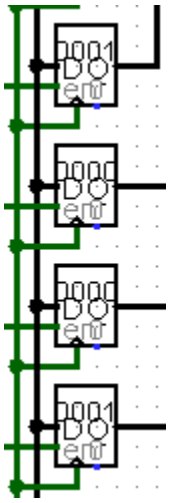
Value 17 (0x11) in register 0, value 11 (0xb) in register 1

pJumpTest.asm:



Value 0 in register 0, value 6 in register 1

pSLT_SGTtest.asm:



Values 1 in registers 2 and 5, values 0 in registers 3 and 4