General Information:

CPU Name: Please Give Us 110+

Names: Justin Gajewski, Tyler Hackett

I pledge my honor that I have abided by the Stevens Honor System.

Please consider us for extra credit for an LED display showing calculation results and our assembler recognizing .data and .text segments

Justin:

- Developed our assembler in Python
  - Reads input file of assembly code, returns image file of hexadecimal, and addresses.
  - Supports assembly instructions (ADD, SUB, LOAD, STORE, MOVE), registers (X0, X1, X2, X3), and immediate integer values
- Created input assembly file and output image files for testing and general usage
- Created and contributed to the instruction manual

Tyler:

- Created CPU in the Logisim .circ file
- Contributed to everything CPU arch-related in the instruction manual, completed the architecture description table

How To:

Please refer to Architecture Desc (page 3) for information on instructions and registers

Use the assembler:

1) Open the “project2” folder in your IDE of choice
2) Create a .text and a .data segment, respectively, in assembly.txt[1]
3) Experiment with combining instructions creatively to make your own functioning program while following the general format for instructions, being mindful of how many references and what kind of references each instruction requires
4) Run the backend.py file; a message saying “Success!” will be printed in the terminal unless an error is encountered[2], in which case please review and make the necessary adjustments to your input
5) Ensure that instrMem.txt and dataMem.txt are populated; these will contain the hexadecimal encoding for your .text and .data segments, respectively

Use the CPU:

6) Open 382Project2.circ from the “project2” folder in Logisim Evolution
7) Locate the Data Mem (RAM), right-click and select “Load Image…”, select dataMem.txt from within the same folder, and click “Open”
8) Locate the Instr Mem (ROM), right-click and select “Load Image…”, select instrMem.txt from within the same folder, and click “Open”
9) From the top of your application, locate “Simulate” and ensure that “Auto-Tick Frequency” is set to 0.5 Hz
10) Enable auto-ticking by clicking on “Auto-Tick Enabled” (right above “Auto-Tick Frequency”)
11) Pat yourself on the back as you watch the LED register display populate with your instructions

The assembly.txt file already contains an example program for your convenience. Use this file to input and create your own program.

[1] - refer to Extra Credit (page 6) for information on .text and .data segments

[2] - possible errors include invalid register reference, invalid immediate value input, invalid instruction

Architecture Desc:

Instruction Name	Binary Encoding	Purpose
ADD Rd Rn Rm	“110” + binary encoding of Rd + binary encoding of Rn + binary encoding of Rm	Adds values from 2 registers and stores the sum in a designated register
SUB Rd Rn Rm	“111” + binary encoding of Rd + binary encoding of Rn + binary encoding of Rm	Subtracts values from 2 registers and stores the difference in a designated register
LOAD Rd Rn	“101” + binary encoding of Rd + binary encoding of Rn	Loads a value from memory into a register with an offset
STORE Rd Rn	“001” + binary encoding of Rd + binary encoding of Rn	Stores a value from a register into memory with an offset
MOVE Rd imm	“100” + binary encoding of Rd + binary encoding of immediate integer value	Moves an immediate integer value ($0 < x < 31$) into a register

Register Reference Name	Binary Encoding
X0	“00”
X1	“01”
X2	“10”
X3	“11”

Encoding for each instruction is 16 bits

How Does Each Component Work?

Assembler:

Input Parsing: reads the input assembly.txt file line by line

Instruction Coding: each instruction is analyzed and converted into its corresponding binary representation, adhering to the predefined opcodes (also converts immediate integer values into binary values), then reversed and converted into its hexadecimal equivalent

Error Handling: error checks to ensure that the correct registers are referenced, immediate integer values fall within the allowed range, and instruction format conforms to specifications; otherwise, throw an error

Output Generation: prints hexadecimal values into their respective image file (intrMem.txt, dataMem.txt)

CPU:

- Architecture
  - Single-Cycle Datapath
  - We have 4 general purpose registers (X0, X1, X2, X3)
- Functions
  - ADD, SUB, MOVE, LOAD, STORE
- Memory
  - ram.txt is our instruction memory
  - Data.txt is our data memory
- Bit Allocation (16 total)
  - Bit: control signal for RegWrite
    - 1 for LOAD, MOVE, ADD, SUB
    - 0 for STORE
  - Bit 2: control signal for Reg2Loc
    - 1 reads from Reg2 in bits 7 and 8 for ADD and SUB
    - 0 reads from bits 4 and 5 for LOAD, STORE, and MOVE
  - Bit 3: control signal for the ALU
    - 1 shows STORE, LOAD, and SUB
    - 0 shows MOVE and ADD
  - Bits 4 and 5: Destination Register
  - Bits 6 and 7: First Source Register
  - Bits 8 and 9: Second Source Register
    - Used for ADD and SUB
  - Bit 10: Operation ALU control signal
    - 1 for STORE, LOAD, and MOVE
    - 0 for ADD and SUB
  - Bit 11: MemtoReg control signal
    - 1 shows load instructions
    - 0 shows all other instructions

- Bit 12-16: Immediate Values
  - Used in MOVE

Extra Credit:

.data and .text segments:

Our assembler supports .text and .data segments, allowing users to define instruction and data memory regions. These segments are essential for separating the program's operational logic from its associated data.

.text segment: This section contains the program's executable instructions. The assembler processes these instructions into hexadecimal machine code, storing them in instrMem.txt for use by the CPU (ensure that instructions are formatted correctly →, e.g., no punctuation between inputs, only spaces between inputs, newline between instructions, etc.).

.data segment: This section holds user-defined data (defined integer values with spaces between) that the program can access during execution. The assembler encodes this data and stores it in dataMem.txt, enabling a smooth data retrieval process and easy manipulation for the CPU.

LED display:

Each LED is connected to the output bus of the register file (X0-X3), which takes the binary output of each register. When a value is written to a register, it is immediately routed through the output bus to the LEDs. Each LED corresponds to a bit in the binary value: a 1 lights up the LED, while a 0 keeps it off.