Arin Ravindran && Jonathan Pruchansky
I pledge my Honor that I have abided by the Stevens Honor System.
SHIFTYCPU - Instruction Manual
All implementations designed for potential EC are marked with yellow

Job Description

Register File Implementation - Jonathan Pruchansky

1.) Used the Lab 08 Register File
2.) Sent out an output of a Control Signal StoreRegData signal and two Data Signals

ALU Implementation - Arin Ravindran

1.) Used a negator to turn the second input into a negative number in case of subtraction
2.) Used a Multiplexer to determine addition or subtraction using the ALUOP control signal
3.) Sent both outputs through the adder for the final ALU output

RAM and PC Implementation - Jonathan Pruchansky

1.) Created two rams in the circuit, one for the Instructions (2 Bytes Addressed) and the other for Data (Byte Addressed)
2.) Wired the Program counter with an automatic Adder for our single-cycle circuit

Immediate Value Implementation - Arin Ravindran

1.) Wired the Immediate Values (Last 7 bits) along with RegOutput 2 to a Multiplexer
2.) Used a control signal (Bit between last Register and Immediate/Throwaway Bits) to determine which value to pass to the ALU, either the Register value or the Immediate
3.) Able to use MOV commands aswell as Addition or Subtraction

Control Unit Implementation - Arin Ravindran

1.) Used Logic Gates to generate 5 Control Signals based on instruction and register design

Assembler Implementation - Jonathan Pruchanksy

1.) Coded the Assembler in Python

Splitting and Opcode Design - Jonathan Pruchansky

1.) Splits the instruction data into 16-bits

Assembler

1.) Github Repository Link: https://github.com/tigritik/Shifty-CPU

2.) Assembler Information
    a.) The assembler is written in Python; we ran it using a Python IDE, but anything with Python works
    b.) We wrote our sample program as sample_program.txt in the GitHub. This is a TXT file that assembled using the Python script
    c.) The assemble() function reads the sample_program.txt file, processes the instructions, and generates machine code (in binary form) for each instruction and data section (if any).
    d.) The binary_to_hex() function takes the binary machine code string generated by the assembler and converts it into hexadecimal. The result is written to a file called a.out.
    e.) The data_to_hex() function handles the data section and writes it to a.data.

Instructions

1.) Enter instructions in the TXT file (no spaces after instructions) and Run the Python File
2.) Load the a.data file into the data ram and the a.out Instruction ram

Note: If running the provided sample program, no change is needed to the Python script.

Program

1.) Our program calculates the Triangular Number of a preset data value, in our case, 5. The Triangular number of 5 is 15, which, at the end of the program, will be stored in the address 0x05 in RAM.

```
MOV b 0 0          // Initializes the b register to 0, this is the summation
MOV c 0 0          // Initializes the c register to 0, this is counter
LDR a c 0          // Loads in tthe value of byte (a)
Add c c 1          // Increments C by 1, c is now 1
Add b b c          // B + C, sum is now 1
Add c c 1          // Increments C by 1, c is now 2
Add b b c          //  B + C, sum is now 3
Add c c 1          // Increments C by 1, c is now 3
Add b b c          // B + C, sum is now 6
Add c c 1          // Increments C by 1, c is now 4
Add b b c          // B + C, sum is now 10
Add c c 1          // Increments C by 1, c is now 5
Add b b c          // B + C, sum is now 15
Str b d c          // Stores the sum in B into the c (5th) place in RAM
data
```

byte 5 //used in program for store address
String abc //cool extra feature (can process string and other data types)
Byte 64 //our assembler can process many types of data

Architecture

CPU Breakdown

1.) 2 Rams | One for the Instructions and one for the Data
    a.) Instruction RAM
        i.) Made to handle 2 Bytes of Instruction
    b.) Data RAM
        i.) Made to handle 1 Byte of Data
2.) Register File
    a.) 4 General Purpose Registers
        i.) Registers A B C and a Zero Register (D)
3.) One ALU and one Control Unit
    a.) ALU
        i.) ALU can handle Addition and Subtraction
            (1) Takes a ALUOP Control signal and 2 Data Inputs
            (2) Uses a Multiplexer, an Adder, and a negator
            (3) Sends Input #1 to the adder
            (4) Branches the second input into a negative and positive
            (5) Uses the ALUOP control signal to determine whether to use positive (Addition) or negative (Subtraction)
            (6) Sends both numbers to adder and returns
    b.) Control Unit
        i.) Takes the opcode as input and puts it through the gates
        ii.) The NOT gates are designed to flip certain bits in order to reach the desired added output
        iii.) Generates Control Signals
            (1) RegW - Indication if we are writing to a Register
            (2) MemW - Indication if we are writing to Memory
            (3) MemR - Indication if we are reading from Memory
            (4) ALUOp - Indication of the ALU operation, 0 for addition, 1 for subtraction
            (5) WriteCntrl - If we are writing from RAM or from ALU output
4.) Splitter System
    a.) Splits 16 bits into 6 segments

i.) Bit 15-14: Opcode - routed to the Control Unit
ii.) Bit 13-12: The Store Register - routed to WriteReg and StoreReg
iii.) Bit 11 -10: The first Read Register - routed to ReadReg 1
iv.) Bit 9-8: Second Read Register - routed to ReadReg2
v.) Bit 7 - Immediate Control Bit, routed to the Immediate Multiplexer
vi.) Bit 6 - 0 - Immediate Data (Filled with numbers incase of no immediate

b.) Sign Extension
i.) Uses a sign extension block to convert the 7 bit immediate to 8 bits as input to the multiplexer

c.) Multiplexer
i.) Check whether to assign value as an immediate or use the register data in RegData 2 output for the

Functions

1.) Addition (ADD) - between two registers or a register and an immediate
2.) Subtraction (SUB) - between two registers or a register and an immediate
3.) Moving (MOV) - moving an immediate into a register, using the zero register
4.) Loading (LDR) - loading from RAM into a register
5.) Storing (STR) - Storing from a register into a ram

OPCODE Breakdown for Control Unit

Opcode	Bits A | B	RegW	MemW	MemR	AluOP	WriteCntrl
LDR	00	1	0	1	0	1
STR	01	0	1	0	0	-
ADD	10	1	0	0	0	0
SUB	11	1	0	0	1	0

MOV Definition

MOV A 5 0 = ADD A D 5 (0 is necessary for proper format with the assembler)

D is the zero register defined in the register file, we add the immediate value to that into the register necessary to simulate MOV in ARM

Immediate is wired to 0, and the register is wired to 1 for the Multiplexer.

Instruction Format:

1.) Opcode - 2 Bits
2.) Register 1 - 2 Bits
3.) Register 2 - 2 Bits
4.) Register 3 - 2 Bits
5.) Immediate Control - 1 Bit (0 is for Immediate use, one is for register use)
6.) Immediate - Signed 7 Bits - Extended to 8 bits using Signed extension for ALU

Register Definitions: A - 11 | B - 10 | C - 01| D - 00 | D is the Zero Register, ABC the other 3

Example OPCODES

1.) ADD A B C - 10 11 10 01 1 0000000 - Last 7 is garbage, we don't use them | 1 for Reg
2.) ADD A B 10 - 10 11 10 00 0 0001010 - Last 7 is 10 in Binary | 0 for Immediate Signal

We chose to make this 16 bits to have space for our instructions, registers (both 2 bits) and the immediate control signal, along with the immediate values in 7 bits, representing numbers up to 256

List of All Allowed Commands:
LDR regLoad regBase regOffset
STR regStore regBase regOffset
ADD regDest reg1 reg2
SUB regDest reg1 reg2

LDR regLoad regBase simm7(offset)
STR regStore regBase simm7(offset)
ADD regDest reg1 simm7(addition with imm)
SUB regDest reg1 simm7(subtraction with imm)

Bonus:
MOV regDest simm7(val to store) 0 == ADD regDest D(0-reg) simm7

To add a Data Segment:
//instruction1
//instruction2
data
string abc (string “abc\0”)
Byte 14 (8bit num)
Char a (ascii val)
Word 503 (16bit num)
Dword 76978 (32 bit num)
Quad 123456789 (64 bit num)