Project: Simulated CPU

CMSC 216.001

Due Date: September 12, 2021, 11:59pm

1 Overview

Each student is asked to implement an assembly language interpreter for simple simulated CPU.

1.1 Objectives

- Create a "simple" program in C
- Begin to build familiarity with Assembly opcodes
- Consider the distinction between registers and memory
- Process input from Standard Input (STDIN)

2 Detailed Description

2.1 Introduction

Programs run on computers by having the hardware (or system software) execute basic operations called instructions. Many languages (such as Java) represent the program to be executed as byte codes which are very similar to machine instructions. In this assignment, you will build an interpreter for a simple machine language.

At their lowest level, computers operate by manipulating information stored in registers and memory. Registers and memory are like variables in C or Java; in fact inside the computer that is how variables get stored. In addition to data, memory also stores the instructions to execute. The basic operation of a computer is to read an instruction from memory, execute it and then move to the instruction stored in the next memory location. Typical instructions will read one or more values (called *operands*) from memory and produce a result into another register or memory location. For example, an instruction might *add* the values stored in two registers, `R3` and `R4` and then store the result in the register `R5`. Other instructions might just move data from one place to another (between registers or between a register and a memory location). A final type of instruction, called a *branch instruction*, is used to change what instruction is executed next (to allow executing if and looping statements).

2.2 the Simulated CPU

The simulated computer has a memory that contains 2^6 (64) words of memory, each 64-bits long.

In addition to memory, the computer has 16 registers that can be used to hold values, and a program counter. One of the registers is special. `R0` is hardwired to 0 and writing to it is legal, but does not change its value, but reading from it *always* returns 0. `PC` is the "Program Counter" and always contains the address of the next instruction to execute. `PC` can be read like a normal register, but can only be modified using special instructions (`B` and `BEQ`), any other attempt to modify it is an Invalid Instruction (including using it as the register value of Branch) `R1-R15` are General Purpose Registers, and can be read or written.

For this project, you should assume that the program text (your code) is in a separate memory from the program data (in other words, your computer is a Harvard Architecture). Assume that each instruction takes up only one memory address in the program memory. You can implement this memory any way you see fit, but you need to be able to execute instructions based on the `PC` and order the instructions were entered. For example, the following program is a simple loop that increments `R1` infinitely:

```
  MOV R1 0x0
  MOV R2 0x1
  ADD R1 R2
  B 0x2
  STOP
```

Note: we jump to instruction address 2, since the add is stored there (the first MOV is at instruction address 0).

2.3 What to do

Create a C program called "`asm`" that acts as an interpreter for a simple assembly language. Your program should accept *upper or lowercase* input from `STDIN` (the default input location in C); the input stream will have the following format:

```
<string opcode> <int oprand1> <int oprand2>
<string opcode> <int oprand1> <int oprand2> <int oprand3>
            ⋮
<string opcode> <int oprand1> <int oprand2>
<string opcode>
<EOF>
```

UPDATE: you can assume that the maximum program length is 1024 instructions.

`EOF` is represented on Unix systems as `<cntl-d>`. All integer values in this project are to be input and output in hexadecimal format (e.g. `0x14` is the hexadecimal representation of the decimal number `20`). The only exception is registers names where are specified as the letter '`R`' followed by a decimal integer (e.g. `R15` is the 16th register since we start counting at `0`). Before exiting your program should dump all registers (including `PC`) and it's memory to `STDOUT` (the default output location in C) in the following format:

```
<register R0>    <value>
<register R1>    <value>
        ⋮
<register R14>   <value>
<register R15>   <value>
```

```
<register PC>    <value>

<byte aligned address>: [mem 0x0] [mem 0x1] ... [mem 0x7]
        ⋮
<byte aligned address>: [mem 0x38] [mem 0x39] ... [mem 0x3F]
```

Include with your project a test input file "`fib.s`" that fills memory with the first 64 Fibonacci numbers. Recall that the n^{th} Fibonacci number $Fib(n)$ is computed by:

$$Fib(n) = \begin{cases} 0 & : n = 0 \\ 1 & : n = 1 \\ Fib(n-1) + Fib(n-2) & : n > 1 \end{cases}$$

Use good C programming style (style and documentation accounts for 20% of your project grade). Refer to the posted style guide for tips on C programming style.

2.4 Description of Opcodes

`MOV <register> <constant>`	Copies the supplied constant into the register location <*register*> Example: `MOV R1 42` - move the value `42` into `R1`
`LDR <register> <memory>`	Copies the value stored in the memory location <*memory*> into register location <*register*>. Example: `LDR R1 42` - load memory location `42` into `R1`
`LDI <register1> <register2>`	Copies the value stored in the memory location indicated by <*register2*> into register location <*register1*> Example: `LDI R1 R2` - load the value currently in the memory location indicated by `R2` into `R1`
`STR <register> <memory>`	Copies the value stored in <*register*> into memory location <*memory*> Example: `STR R1 42` - stores the value currently in `R1` into memory location `42`
`STI <register1> <register2>`	Copies the value stored in <*register1*> into memory location indicated by <*register2*> Example: `STI R1 R2` - stores the value currently in `R1` into memory location pointed to by `R2`
`ADD <register1> <register2> <register3>`	Adds the value stored in <*register2*> to the value stored in <*register3*> and stores the result into <*register1*> Example: `ADD R1 R2 R3` - stores the sum of `R2` and `R3` into `R1`
`MUL <register1> <register2> <register3>`	Multiplies the value stored in <*register2*> to the value stored in <*register3*> and stores the result into <*register1*> Example: `MUL R1 R2 R3` - stores the product of `R2` and `R3` into `R1`
`CMP <register1> <register2>`	Compares the value of <*register1*> and <*register2*>, and sets the `COMPARE_FLAG` if they are equal, resets the `COMPARE_FLAG` if they are different Example: `CMP R1 R2` - sets `COMPARE_FLAG`, if `R1` and `R2` contain the same value
`B <constant>`	sets the value of `PC` (program counter) to <*constant*> Example: `B 42` - branches (jumps) by setting the program counter to memory location `42`
`BEQ <constant>`	If the value of `COMPARE_FLAG` is NOT zero, sets the value of `PC` (program counter) to <CONSTANT> Example: `BEQ 42` - sets the `PC` to the value of `42`, only if `COMPARE_FLAG` DOES NOT contains zero.
`STOP`	Causes the simulated computer to stop processing instructions

2.5 Compiling you program

Please use gcc to compile and submit your program. specifically use the following command to compile your program:

```
gcc -Wall -pedantic-errors -Werror <filename.c> -o asm
```

Replace <*filename.c*> with the filename for your source code. I chose `asm.c` for mine, and I suggest you do the same. We'll explain the other options in class, but the result should be a program called `asm` All your C programs in this course should be written in correct C, which means they must compile and run correctly when compiled with the compiler `gcc`, with the options `-Wall, -pedantic-errors, and -Werror`. Except as noted below, you may use any C language features in your project that have been covered in class, or that are in the chapters covered so far and during the time this project is assigned, so long as your program works successfully using the compiler options mentioned above.

2.6 Example output

2.6.1 Example 1

```
project1> ./asm
mov R1 0x1
str R1 0xf
ldr R2 0xf
str R2 0x2e
mov R1 0xef
str R1 0x25
mov R2 0x42
str R2 0x10
stop
^D
register 0x0: 0x000000000000
register 0x1: 0x0000000000EF
register 0x2: 0x000000000042
register 0x3: 0x000000000000
register 0x4: 0x000000000000
register 0x5: 0x000000000000
register 0x6: 0x000000000000
register 0x7: 0x000000000000
register 0x8: 0x000000000000
register 0x9: 0x000000000000
register 0xA: 0x000000000000
register 0xB: 0x000000000000
register 0xC: 0x000000000000
register 0xD: 0x000000000000
register 0xE: 0x000000000000
register 0xF: 0x000000000000
register  PC: 0x000000000008

0x00:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x08:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000001
0x10:  0x000000000042 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x18:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x20:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x0000000000EF 0x000000000000 0x000000000000
0x28:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000001 0x000000000000
0x30:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x38:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
```

2.6.2 Example 2

```
project1> ./asm
mov R1 0x5
mov R2 -0x1
sti R1 R1
add R1 R1 R2
cmp R1 R0
beq 0x7
b 0x2
stop
^D
register 0x0: 0x000000000000
register 0x1: 0x000000000000
register 0x2: 0xFFFFFFFFFFFFFFFF
register 0x3: 0x000000000003
register 0x4: 0x000000000000
register 0x5: 0x000000000000
register 0x6: 0x000000000000
register 0x7: 0x000000000000
register 0x8: 0x000000000000
register 0x9: 0x000000000000
register 0xA: 0x000000000000
register 0xB: 0x000000000000
register 0xC: 0x000000000000
register 0xD: 0x000000000000
register 0xE: 0x000000000000
register 0xF: 0x000000000000
register  PC: 0x000000000007

0x00:  0x000000000000 0x000000000001 0x000000000002 0x000000000003 0x000000000004 0x000000000005 0x000000000000 0x000000000000
0x08:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x10:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x18:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x20:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x28:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x30:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
0x38:  0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000 0x000000000000
```

2.6.3 Example 3

```
project1> ./asm < fib.s
register 0x0: 0x000000000000
register 0x1: 0x000000000000
register 0x2: 0x05F6C7B064E2
register 0x3: 0x09A661CA20BB
register 0x4: 0x000000000041
register 0x5: 0x000000000001
register 0x6: 0xFFFFFFFFFFFFFFFF
register 0x7: 0x000000000008
register 0x8: 0x000000000000
register 0x9: 0x000000000000
register 0xA: 0x000000000000
register 0xB: 0x000000000000
register 0xC: 0x000000000000
register 0xD: 0x000000000000
register 0xE: 0x000000000000
register 0xF: 0x000000000000
register  PC: 0x00000000000F

0x00:  0x000000000000 0x000000000001 0x000000000001 0x000000000002 0x000000000003 0x000000000005 0x000000000008 0x00000000000D
0x08:  0x000000000015 0x000000000022 0x000000000037 0x000000000059 0x000000000090 0x0000000000E9 0x000000000179 0x000000000262
0x10:  0x0000000003DB 0x00000000063D 0x000000000A18 0x000000001055 0x000000001A6D 0x000000002AC2 0x00000000452F 0x000000006FF1
0x18:  0x00000000B520 0x000000012511 0x00000001DA31 0x00000002FF42 0x00000004D973 0x00000007D8B5 0x0000000CB228 0x000000148ADD
0x20:  0x000000213D05 0x00000035C7E2 0x0000005704E7 0x0000008CCCC9 0x000000E3D1B0 0x000001709E79 0x000002547029 0x000003C50EA2
0x28:  0x000006197ECB 0x000009DE8D6D 0x00000FF80C38 0x000019D699A5 0x000029CEA5DD 0x000043A53F82 0x00006D73E55F 0x0000B11924E1
0x30:  0x00011E8D0A40 0x0001CFA62F21 0x0002EE333961 0x0004BDD96882 0x0007AC0CA1E3 0x000C69E60A65 0x001415F2AC48 0x00207FD8B6AD
0x38:  0x003495CB62F5 0x005515A419A2 0x0089AB6F7C97 0x00DEC1139639 0x01686C8312D0 0x02472D96A909 0x03AF9A19BBD9 0x05F6C7B064E2
```

3 Submission

Project should be submitted via Blackboard by **February 5, 2021, 11:59pm**. Follow these instructions to turn in your project.

You should submit the following files:

- Makefile
- asm.c
- asm.h *(optional)*
- fib.s
- *any other source files your project needs*

The following submission directions use the command-line submit program that we will use for all projects this semester.

1. create a directory for your project:

   ```
   mkdir sim-cpu
   ```

2. create (or copy) all of your source files in this directory. Example: To copy a file called `example.c` into your sim-cpu directory:

   ```
   cp example.c sim-cpu
   ```

3. change parent directory of your project directory:

   ```
       cd sim-cpu/..
   ```

4. Create a tar file named `<user_name>.tgz`, where `<user_name>` is your studentID and `<proj_dir>` is the directory containing the code for your project, by typing:

   ```
   tar -czf <user_name>.tgz sim-cpu
   ```

5. Finally, submit the compressed tar file to Blackboard in accordance with the class policies.

Late assignments will not be given credit.

4 Grading

While this rubric is subject to change based on class performance, the current grading rubric for this assignment is as follows:

component	value
Correctness	50
Completeness	40
Code Quality	10