Project: Simulated CPU

CMSC 216.001

Due Date: February 27, 2023, 11:59pm

1 Overview

Each student is asked to implement a simulated CPU for processing a simple machine language.

1.1 Objectives

- Create a "simple" program in C
- Begin to build familiarity with basic machine operations
- Consider the distinction between registers and memory
- Manipulate 2's complement values of different length

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 a simulator to process 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 16-bit word size and a memory that contains $2^1$1 (2048) bytes of memory.

In addition to memory, the computer has 16 registers that can be used to hold 16-bit values, plus a program counter. The `PC` register is the "Program Counter" and always contains the address of the next instruction to execute. The `PC` cannot be read or written like a normal register, but can only be modified using special instructions: Branch (`B`) and Branch-if-EQual (`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.

Additionally, the simulated CPU includes a single "zero-flag" status-bit. This bit tracks the result of the last executed Comparison instruction ()`CMP`). If that comparison was between two equal values, the zero-flag should be set to `1`, otherwise it should be `0`.

For this project, you should assume that the program text (the machine code) is stored in the main memory of the computer starting at address 0x0000. Additionally, you may assume that there is a hard limit of 1024 possible instructions in our programs (since that would entire fill our 2048 bytes of available memory). Assume that each instruction takes up 2 bytes (16-bits) 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:

```
# Put 0 into register 1
  MOV R1 0x0
# put 1 into register 2
  MOV R2 0x1
# R1 = R1 + R2
  ADD R1 R1 R2
# PC = PC - 2
  B 0xFFE
# STOP Execution, this is never reached
  END
```

Note: we branch with a value of -2, This has the effect of moving the PC back 1 instruction, since we want to repeat the instruction just before the branch statement.

2.3 What to do

Create a C program called "`uCPU`" that acts as a simulator for a small machine language. Your program should accept *upper or lowercase* input from `STDIN` (the default input location in C). The input will consist of a series of instructions encoded as 16-bit (4-digit hexadecimal) numbers, terminated by an `<EOF>` The input should not contain any whitespace or characters other than hexadecimal digits.

Each 16-bit, 4-digit instruction has the following format:

1. the first hexadecimal digit, represents the instruction opcode (see section 2.4 below)
2. the remaining three hexadecimal digits represent the operands to the instruction. There are three possible formats for operands:
    - registers - are encoded as a single hexadecimal digit, representing an unsigned value from 0–15 indicating a register: `R0`–`R15`

- 8-bit constants - are encoded as two-digit hexadecimal numbers, representing a signed 8-bit value with a range of -128–127.
- 12-bit constants - are encoded as three-digit hexadecimal numbers, representing a signed 12-bit value with a range of -2048–2047.

You can assume that the maximum program length is 1024 instructions (2048 bytes).

`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`).

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>

<word aligned address>: [mem 0x0000] [mem 0x0001] ... [mem 0x000F]
<word aligned address>: [mem 0x0010] [mem 0x0011] ... [mem 0x001F]
        ⋮
<word aligned address>: [mem 0x07E0] [mem 0x07E1] ... [mem 0x07EF]
<word aligned address>: [mem 0x07F0] [mem 0x07F1] ... [mem 0x07FF]
```

Include with your project a test **assembly language** program `"fib.s"` that stores the first 49 Fibonacci numbers at memory location 0x0040. *Do **not** compile your fib.s program using Project 1!!*

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 10% of your project grade). Refer to the posted style guide for tips on C programming style.

2.4 Description of Opcodes

Instruction	Binary opcode	Description
`ADD <register1> <register2> <register3>`	0000	Add, sets the value of the first operand to the sum of the second and third
`SUB <register1> <register2> <register3>`	0001	Subtract, sets the value of the first operand to the difference of the second and third
`AND <register1> <register2> <register3>`	0010	Bitwise AND, sets the value of the first operand to the bitwise AND of the second and third
`ORR <register1> <register2> <register3>`	0011	Bitwise OR, sets the value of the first operand to the bitwise OR of the second and third
`EOR <register1> <register2> <register3>`	0100	Bitwise XOR, sets the value of the first operand to the bitwise exclusive OR of the second and third
`LSL <register1> <register2> <register3>`	0101	Logical Shift Left, sets the value of the first operand to the value of the second, shifted left by the amount indicated in the third
`LSR <register1> <register2> <register3>`	0110	Logical Shift Right, sets the value of the first operand to the value of the second, shifted right by the amount indicated in the third. Zeros are added to the most significant bits of the first operand as necessary
`ASR <register1> <register2> <register3>`	0111	Arithmetic Shift Right, sets the value of the first operand to the value of the second, shifted right by the amount indicated in the third. either zero or one is added to the most significant bits of the first operand as necessary to maintain the sign of the resulting value
`LDR <register1> [<register2>]`	1000	Load, sets the value of the first operand from the memory location specified by the second
`STR <register1> [<register2>]`	1001	Store, places the value stored in the first operand into a memory location specified by the second
`CMP <register1> <register2>`	1010	Compare, compares the values in the operands and sets the `zero-flag` to 1 if they are equal
`MOV <register1> <constant>`	1011	Move, sets the value in the first operand to the signed 8-bit value from the second operand
`B <constant>`	1100	Branch, changes the Program Counter by adding the signed 12-bit operand to it
`BEQ <constant>`	1101	Branch if Equal, modifies the Program Counter only if the z-flag is currently set to 1
`END`	1110	Terminates CPU execution, denotes the end of a program
`NOP`	1111	No operation, this instruction does nothing

2.5 Compiling your program

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

```
gcc -Wall -pedantic-errors -Werror -std=c99 <filename.c> -o uCPU
```

Replace <*filename.c*> with the filename for your source code. I chose `uCPU.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 `uCPU` All your C programs in this course should be written to the C99 standard, unless otherwise indicated, which means they must compile and run correctly when compiled with the compiler `gcc`, with the options `-Wall, -pedantic-errors, and -Werror -std=c99`. 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> ./uCPU < gfa1.o
register  0: 0x0000
register  1: 0x0001
register  2: 0x0002
register  3: 0x0003
register  4: 0x0004
register  5: 0x0005
register  6: 0x0006
register  7: 0x0007
register  8: 0x0008
register  9: 0x0009
register 10: 0x000A
register 11: 0x000B
register 12: 0x000C
register 13: 0x000D
register 14: 0x000E
register 15: 0x000F
register PC: 0x0020

0x0000:  B000 B101 B202 B303 B404 B505 B606 B707
0x0010:  B808 B909 BA0A BB0B BC0C BD0D BE0E BF0F
0x0020:  E000 0000 0000 0000 0000 0000 0000 0000
0x0030:  0000 0000 0000 0000 0000 0000 0000 0000

      [lines ommited for brevity]
[actual project should have full output]

0x07E0:  0000 0000 0000 0000 0000 0000 0000 0000
0x07F0:  0000 0000 0000 0000 0000 0000 0000 0000
```

2.6.2 Example 2

```
project1> ./uCPU < gfa2.o
register  0: 0x0008
register  1: 0x0003
register  2: 0x000B
register  3: 0x0005
register  4: 0x0000
register  5: 0x000B
register  6: 0x000B
register  7: 0x0003
register  8: 0x0000
register  9: 0x0000
register 10: 0x0000
register 11: 0x0000
register 12: 0x0000
register 13: 0x0000
register 14: 0x0000
register 15: 0x0000
register PC: 0x0010

0x0000:  B008 B103 0201 1301 2401 3501 4601 4702
0x0010:  E000 0000 0000 0000 0000 0000 0000 0000
0x0020:  0000 0000 0000 0000 0000 0000 0000 0000

        [lines ommited for brevity]
[actual project should have full output]

0x07E0:  0000 0000 0000 0000 0000 0000 0000 0000
```

```
0x07F0:  0000 0000 0000 0000 0000 0000 0000 0000
```

2.6.3 Example 3

```
project1> ./asm < public1.o
register  0: 0x0000
register  1: 0x0000
register  2: 0x0000
register  3: 0x0000
register  4: 0x0112
register  5: 0x0000
register  6: 0x0000
register  7: 0x0000
register  8: 0x0000
register  9: 0x0000
register 10: 0x0000
register 11: 0x0000
register 12: 0x0000
register 13: 0x0000
register 14: 0x0000
register 15: 0xFFFF
register PC: 0x0800

0x0000:  0112 84A0 94F0 A0C0 BFFF C100 E000 0000
0x0010:  0000 0000 0000 0000 0000 0000 0000 0000
0x0020:  0000 0000 0000 0000 0000 0000 0000 0000

        [lines ommited for brevity]
[actual project should have full output]

0x07E0:  0000 0000 0000 0000 0000 0000 0000 0000
0x07F0:  0000 0000 0000 0000 0000 0000 0000 0000
```

3 Submission

Project should be submitted by **February 27, 2023, 11:59pm**. Follow these instructions to turn in your project.

You should submit the following files:

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

The following submission directions use the command-line `submit` program on the class server that we will use for all projects this semester:

- Log into the VDI: `http://desktop.montgomerycollege.edu`
- If necessary, transfer your source code onto the VDI

- Use *Bitvise SSH* client to log into `tpaclinux`
- If necessary, use the *Bitvise SFTP transfer* to upload your source code to the server
- Finally, use the command-line `submit` program to turn in your code

An example command line submission of files: `uCPU.c`, `fib.s`, `util.c`, and `util.h` for project 2, would look like:

```
submit proj2 uCPU.c fib.s util.c util.h
```

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