SIMCPU: a simulated CPU

CMSC 216.001

Due Date: September 24, 2023, 11:59pm

1 Overview

For this project, each student will design and implement a simulated CPU capable of processing our custom machine language, which was introduced in the 'TRANSLATE' project. This simulated CPU will provide insights into the inner workings of computer hardware and illustrate how assembly instructions get translated into machine-level operations.

1.1 Objectives

- Develop a C program simulating a CPU.
- Understand the foundational machine operations and how they interact with registers and memory.
- Differentiate between the roles of registers and memory in a CPU.
- Work with 2's complement values and handle different operand lengths.

2 Detailed Description

2.1 Introduction

In the digital realm, computers operate by executing a series of basic operations termed instructions. High-level programming languages, such as Java, often rely on bytecodes resembling these machine instructions to represent their programs. In the TRANSLATE project, you worked with an assembly language that closely mirrored machine instructions. Now, in the CPUSIM project, your task is to simulate a CPU that processes this machine language.

Diving deeper into a computer's anatomy, we find registers and memory at its core. These components bear resemblance to the variables we declare in programming languages like C or Java. In fact, these variables eventually find their home in either registers or memory. Memory not only houses data but also the sequence of instructions meant for execution. A computer's primary modus operandi is to fetch an instruction from memory, execute it, and then transition to the subsequent instruction. These instructions often involve operands, which could be values extracted from memory or registers. An instruction might encompass operations like addition, where values from two registers, say `R3` and `R4`, are added and stored in another register, `R5`. Some instructions merely shuffle data between locations. Branch instructions, however, play a crucial role in dictating the program's flow, enabling functionalities like conditional statements and loops.

With the foundation laid in the TRANSLATE project, the CPUSIM project will offer a more in-depth exploration of these concepts, allowing you to simulate how a real CPU might interpret and execute machine instructions.

2.2 The Simulated CPU

This project revolves around a simulated CPU with the following characteristics:

- **Memory and Word Size:** The CPU has a memory capacity of 2^{12} bytes (or 4096 bytes). Each word in this memory is 16 bits in size.
- **Registers:** The CPU is equipped with 16 registers that can each hold a 16-bit value. The registers are labeled `R1` through `R15`. These are general-purpose registers, meaning they can be both read from and written to.
- **Program Counter (`PC`):** The `PC` is a special register that holds the memory address of the next instruction to be executed. Direct reading or writing to the `PC` is not allowed. Its value can only be modified using the Jump (`JMP`) and Branch-if-EQual (`BEQ`) instructions. Any other attempts to modify it will be treated as an Invalid Instruction.
- **Status Flags Register:** This CPU also features a "Status Flags Register" which is used to indicate the result of the last executed Comparison (`CMP`) instruction. If the compared values were equal, the status flag will be set to `1`; otherwise, it will be `0`.
- **Program Memory:** For the purpose of this project, you can assume that the machine code of the program resides in the computer's main memory, starting from address 0x0000. Given that each instruction occupies 2 bytes (or 16 bits) and the memory limit is 4096 bytes, a program can have a maximum of 2048 instructions.

As an illustrative example, consider the following program. It's a simple loop that endlessly increments the value in `R1`:

```
# Initialize R1 with 0
  MOV R1 0x0
# Load R2 with the value 1
  MOV R2 0x1
# Add R1 and R2, store the result in R1
  ADD R1 R1 R2
# Decrement the PC by 4 to loop back to the ADD instruction
  JMP 0xFFFC
```

In the above program, the 'JMP 0xFFFC' instruction causes the program counter to decrement by 4 bytes, which makes it return to the ADD instruction, thus creating an infinite loop.

2.3 Project Task: Simulating the `cpu`

Your main objective in this assignment is to design and implement a program called `cpu` in C. This program will simulate the execution of machine code instructions for our hypothetical 16-bit CPU.

2.3.1 Input Specification

Your simulator should accept both uppercase and lowercase input from `STDIN` (standard input in C). The input stream is composed of a series of 4-digit hexadecimal numbers, each representing a single machine code instruction.

The input sequence concludes with an `<EOF>` (End-Of-File) character. Ensure that your program can handle input without any whitespace or characters other than the valid hexadecimal digits.

2.3.2 Instruction Format

Every 16-bit, 4-digit instruction you'll receive adheres to the following structure:

1. **Opcode:** The initial hexadecimal digit signifies the instruction opcode. The specific opcode values and their associated machine operations are detailed in Section **??**.
2. **Operands:** The subsequent three hexadecimal digits represent the operands for the instruction. Operands can come in one of three formats:
   - **Registers:** Encoded as a single hexadecimal digit. This digit is an unsigned value ranging from 0 to 15, pointing to one of the registers: `R0` to `R15`.
   - **8-bit Constants:** Encoded as a two-digit hexadecimal number. This represents a signed 8-bit value with permissible values from -128 to 127.
   - **12-bit Constants:** Encoded as a three-digit hexadecimal number. It denotes a signed 12-bit value, with a range spanning from -2048 to 2047.

2.4 Processing the Input

The input to your program will be a continuous stream of hexadecimal digits representing opcodes and their associated operands. Each instruction will be represented by four hexadecimal digits, with the first digit being the opcode and the remaining three digits representing the operands. The number of operands can range from 0 to 3, depending on the opcode.

Based on the opcode table in Section **??**:

- An input of `0x0123` would represent an `ADD` instruction with operands $R1$, $R2$, and $R3$.
- An input of `0xA010` would represent a `MOV` instruction with operands $R0$ and the constant $0x10$.

You are expected to interpret the input stream, decode each instruction, and store them consecutively in memory starting from location `0x0000`.

2.4.1 Program Constraints and Considerations

- **Hexadecimal Format:** All integer values you encounter in this project, both for input and output, should be represented in hexadecimal format. For clarity, the hexadecimal value `0x14` corresponds to the decimal number 20.
- **Maximum Program Length:** For the purposes of this assignment, you can make the assumption that the maximum length of the program being simulated is 1024 instructions. This translates to a total of 2048 bytes.
- **End of Input:** The end of the input stream is marked by the `EOF` character. On Unix systems, this character can be generated using the `<cntl-d>` key combination.

- **Termination Instruction:** To halt the simulated CPU and prevent further processing, a special instruction code should be introduced. For this purpose, the instruction code `0xDEAD` will be used. This code is uniquely designed to indicate the end of processing and wasn't generated from the TRANSLATE assignment. It is essential to note that this `0xDEAD` instruction should be automatically appended to the end of the input data by your SIMCPU project code.

2.5 Simulating a CPU

Once the input has been processed and stored in memory, your program will initiate the simulation of the CPU. The Program Counter (`PC`) should begin execution at address `0x0000`. The program then involves manipulating the values in the registers, memory, and even altering the program counter or status register based on the instruction.

Examples of how the simulation might work:

- **Example 1:** For the input `0x0123`, the `ADD` instruction will add the values in registers $R2$ and $R3$ and store the result in register $R1$.
- **Example 2:** For the input `0xA010`, the `MOV` instruction will load the constant $0x10$ into register $R0$.
- **Example 3:** If the input is `0xB200`, it would represent a `LDR` instruction. The value in memory at the address specified by register $R0$ would be loaded into register $R2$.
- **Example 4:** An input of `0xEFFF` would represent a `JMP` instruction with a signed 12-bit offset of $0xFFE$. In this case, the offset would be -2. The program counter would be modified by adding this offset to it.
- **Example 5:** If the input is `0x91A0`, the `CMP` instruction would compare the values in registers $R1$ and $R0$. The status register would be updated based on the result of the comparison.

To ensure that the CPU simulation terminates correctly, a special instruction code `0xDEAD` will be automatically added to the end of the memory. This instruction acts as a halt or end instruction, effectively signaling the CPU to cease further processing. This is essential as without a clear termination point, the CPU simulation could potentially run indefinitely or until an invalid instruction is encountered.

2.6 Program Output

After completing the simulation, your program should display the state of the simulated computer. This includes the current values in all the registers and the contents of memory. The output should be written to `STDOUT` (the default output location in C) in the specified format:

2.6.1 Register Dump

For each register, including the `PC`, display the register name followed by its current value. The values should be displayed in hexadecimal format.

```
R0: <value>
R1: <value>
...
R14: <value>
R15: <value>
```

```
PC: <value>
```

2.6.2 Memory Dump

Display the contents of memory, 16 words (32 bytes) per line. Each line should start with the word-aligned address, followed by the contents of 16 consecutive memory locations in hexadecimal format.

```
<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]
```

Note: Ensure that the memory addresses and values are properly word-aligned. This will aid in readability and help users quickly locate specific memory addresses.

2.7 Assembly Language Program Requirement

As part of your project, you are required to create an assembly language program named `fib.s`. This program should compute the first 25 Fibonacci numbers and store them starting from memory location `0x0040`.

Important: Only submit the assembly language program (`fib.s`). Do not submit any compiled versions of this program.

2.7.1 Fibonacci Series Definition

The Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. The n^{th} Fibonacci number $Fib(n)$ can be defined as:

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

2.7.2 Programming Style

Maintaining good programming practices is crucial. Ensure that your code is well-documented, readable, and adheres to best practices. Your style and documentation will account for 10% of your project grade. For guidance on achieving a commendable programming style, please refer to the style guide provided on the course platform.

2.8 Description of Opcodes

Instruction	Binary opcode	Description
`ADD <R1> <R2> <R3>`	0000	Add, sets $R1 = R2 + R3$
`SUB <R1> <R2> <R3>`	0001	Subtract, sets $R1 = R2 - R3$
`MUL <R1> <R2> <R3>`	0010	Multiply, sets $R1 = R2 \times R3$
`DIV <R1> <R2> <R3>`	0011	Divide, sets $R1 = R2 \div R3$
`AND <R1> <R2> <R3>`	0100	Bitwise AND, sets $R1 = R2 \wedge R3$
`ORR <R1> <R2> <R3>`	0101	Bitwise OR, sets $R1 = R2 \vee R3$
`NOT <R1> <R2>`	0110	Bitwise NOT, sets $R1 = \neg R2$
`SHL <R1> <R2> <R3>`	0111	Shift Left, sets $R1 = R2 \ll R3$
`SHR <R1> <R2> <R3>`	1000	Shift Right, sets $R1 = R2 \gg R3$
`CMP <R1> <R2>`	1001	Compare $R1$ and $R2$, set flags
`MOV <register> <constant>`	1010	Move, loads the 8-bit signed constant into the specified register
`LDR <Rdest> [<Raddr>]`	1011	Load, sets *Rdest* from memory at the address in *Raddr*
`STR <Rsrc> [<Raddr>]`	1100	Store, sets memory at the address in *Raddr* to *Rsrc*
`NOP`	1101	No operation
`JMP <offset>`	1110	Jump to `PC` + `signed 12-bit offset`
`JEQ <offset>`	1111	Jump to `PC` + `signed 12-bit offset` if equal

2.9 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 cpu
```

Replace <*filename.c*> with the filename for your source code. I chose `cpu.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 `cpu` 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.

2.10 Example output

2.10.1 Example 1

```
CPUSIM> ./cpu < 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: 0x0022

0x0000:  A000 A101 A202 A303 A404 A505 A606 A707
0x0010:  A808 A909 AA0A AB0B AC0C AD0D AE0E AF0F
0x0020:  DEAD 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]

0x0FE0:  0000 0000 0000 0000 0000 0000 0000 0000
0x0FF0:  0000 0000 0000 0000 0000 0000 0000 0000
```

2.10.2 Example 2

```
CPUSIM> ./cpu < gfa2.o
register  0: 0x0000
register  1: 0x0000
register  2: 0x0001
register  3: 0x0000
register  4: 0x0000
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: 0x0000
register PC: 0x000E

0x0000:  A101 A201 9010 F004 0112 EFF8 DEAD 0000
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]

0x0FE0:  0000 0000 0000 0000 0000 0000 0000 0000
0x0FF0:  0000 0000 0000 0000 0000 0000 0000 0000
```

3 Submission

Project should be submitted by **September 24, 2023, 11:59pm**. Follow these instructions to turn in your project.

You should submit the following files:

- `cpu.c`
- `cpu.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: `cpu.c`, `fib.s`, `util.c`, and `util.h` for project 2, would look like:

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

Late assignments will not be given credit.

4 Grading

The grading rubric for this assignment is provided below. While this rubric is subject to change based on class performance, it represents the current criteria for assessment:

Component	Value
Correctness	50
Completeness	40
Code Quality	10

Please ensure that your submitted code meets the expectations outlined in this rubric to maximize your grade on this assignment.