# Assignment 1: Simulator

**Submit Assignment**

**Due** Sep 4 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** c
**Available** Aug 21 at 12am - Sep 7 at 11:59pm 18 days

The objective of this assignment it to implement a simulator for a simplified x86 processor. The basic job of a simulator is to mimic a real processor by keeping track (in software) of the various components of a processor such as registers, memory, and which instruction to execute next. As your simulator program runs, it should "execute" simulated x86 instructions one at a time, while updating software representations of registers and memory.

This assignment is divided into two parts. Student should get started on Part 1 right away. More information about x86 instructions is covered in Lecture 3, which is useful for completing Part 2.

In order to complete such a simulator in our first assignment, the task is made easier by supporting only 32-bit ("l") instructions with a machine code representation of instructions fixed to the 32-bit size.

**Start by downloading simulator.tar.gz**, which is a gzipped tar file that can be unpacked by running the following command on a CADE lab1 machine:

```
tar zxvf simulator.tar.gz
```

This creates a new directory called *simulator* containing these files and directories:

- *simulator.c* — This source code file serves as a starting point for your solution.
- *instruction.h* — This header file provides some useful definitions for representing instructions.
- *tests* — This directory contains simple, moderate, and complex tests for your simulator. The tests themselves are machine code programs that your simulator should be able to input and run.
- *run_tests.sh* — This bash script runs all of the tests and reports the results.
- *assembler* — This file is a pre-compiled executable that can be used to generate a simulator input machine code file given an assembly file. (You do not need to use this executable, but it may be useful for testing if you want to modify or write your own assembly test programs.)

## Compiling the simulator

To compile the simulator, run one of the following commands:

```
gcc -g simulator.c -o simulator
```

The above compiles with debug symbols, useful during development.

```
gcc -O2 simulator.c -o simulator
```

The above compiles with optimization. **The second version is used for grading — make sure your program passes all tests when compiled with optimization.** If your program behaves differently with and without optimization, it most likely means your program has undefined behavior, such as using uninitialized data.

### Running the simulator

To run the simulator, pass it a command-line argument specifying the machine code binary file to run. The *tests* directory contains a number of assembly files with extension *.s* and assembled binary machine code files with extension *.o*. The simulator accepts the *.o* files as input.

For example, you can run the simulator with the following command:

```
./simulator tests/simple/subl.o
```

This causes the simulator to load the *subl.o* binary file, which represents the machine encoding of the instructions in the corresponding *.s* file.

The simulator, as is provided above, only loads the binary file. It is your job to finish the implementation of the simulator according to the specifications described in this assignment so that it executes (simulates) the provided machine code program.

## Part 1 — Decode the Binary Machine Code

The provided *.o* files are binary files representing machine code for our simulator. Each instruction is encoded as four consecutive bytes (32 bits). For example, consider *tests/simple/subl.s* and *tests/simple/subl.o*. Notice that there are four instructions in the assembly file, so the binary *.o* file should contain sixteen bytes (four instructions * four bytes each). Verify that this is true by looking at the file size using the *ls* command:

```
ls -l tests/simple/subl.o
```

You should see "16" somewhere in the output.

Our first task is to decode the raw bytes into usable data structures in our simulator. For Part 1, we decode the bytes. In Part 2, we execute the decoded instructions.

Each 4-byte value in the machine code represents all of the necessary information for one instruction, such as the opcode and inputs/outputs. Our machine code represents all instructions in the form given below. All bits are specified in order of significance.

| **Bits:** | 31 - 27 | 26 - 22 | 21 - 17 | 16 | 15 - 0 |
|---|---|---|---|---|---|
| **Meaning:** | opcode | reg1 | reg2 | unused | immediate |

For example, consider the third instruction in the test file *tests/simple/addl_imm_reg.s*:

```
addl    $-2, %ebx
```

In this particular form of the *addl* instruction, the opcode is 2, register %ebx has an ID of 1, and the immediate bits represent -2. Thus, the instruction is encoded with the following bits:

00010000010000001111111111111110

opcode = 2

reg1 = 1

reg2 = 0 (unused)

immediate = -2 (16-bit signed two's complement)

The "reg2" bits are unused in this instruction since it has only one register. The unused 0 bit between the reg2 and immediate bits is always unused. See the full list of **opcodes** and **register IDs**.

### instruction_t

Consider the provided *instruction.h* header file, which defines a struct called *instruction_t* for representing an instruction. Our task is to create an array of *instruction_t* elements and fill in the fields of each element with the appropriate values based on the raw bits encoded in the input file.

For example, suppose that you create an *instruction_t* variable called *instr*, for representing the example *addl* instruction above. After filling in the fields of *instr*, they should have the following values:

```
instr.opcode = 2

instr.first_register = 1

instr.second_register = 0

instr.immediate = -2
```

Your job is to programmatically extract these values out of the 32 bits that encode the instruction using bitwise shifting and masking operations in C. To print the decoded values and check your work, uncomment line 75 in *simulator.c*. Verify correctness using the **opcode** and **register ID** tables.

### Example run

When the task of decoding instructions is complete for Part 1, run your program on the provided *.o* files. As an example, the output of the simulator on *tests/complex/factorial.o* should be the following:

```
./simulator tests/complex/factorial.o
instructions:
op: 0, reg1: 6, reg2: 0, imm: 8
op: 21, reg1: 5, reg2: 0, imm: 0
op: 16, reg1: 0, reg2: 0, imm: 12
op: 20, reg1: 0, reg2: 0, imm: 0
```

```
op: 2, reg1: 6, reg2: 0, imm: 8
op: 17, reg1: 0, reg2: 0, imm: 0
op: 18, reg1: 1, reg2: 0, imm: 0
op: 5, reg1: 5, reg2: 1, imm: 0
op: 8, reg1: 8, reg2: 0, imm: 2
op: 9, reg1: 8, reg2: 5, imm: 0
op: 10, reg1: 0, reg2: 0, imm: 16
op: 0, reg1: 5, reg2: 0, imm: 1
op: 16, reg1: 0, reg2: 0, imm: -28
op: 3, reg1: 1, reg2: 0, imm: 0
op: 15, reg1: 0, reg2: 0, imm: 4
op: 8, reg1: 0, reg2: 0, imm: 2
op: 19, reg1: 1, reg2: 0, imm: 0
op: 17, reg1: 0, reg2: 0, imm: 0
--------------
```

**Re-comment line 75 before you run tests on Part 2.** The extra output causes the auto-tester to report failure.

## Part 2 — Simulate the Assembly Instructions

With the usable array of *instruction_t* elements created in Part 1, the task in Part 2 is to simulate the execution of each instruction, one by one. Your simulator must represent a program counter, registers, and a small memory stack. Each instruction may modify registers and/or memory, and then control moves to the next instruction (by modifying the program counter).

The simulated machine should adhere to the following specifications:

1. The program counter should start at 0. The first instruction to execute is always the first one in the input file. Four bytes is sufficient to represent the program counter for any possible example.
2. There is a 1024-byte memory stack. Every byte in memory should initially have a value of 0.
3. There are 17 registers (see the **register ID table**). The width of each register is 32 bits. All registers should initially have a value of 0, except %esp. %esp is the stack pointer register, and should initially have a value of 1024. (Note that address 1024 is just beyond the range of valid memory addresses.)
4. The simulator should execute instructions one by one using the behavior in the **instruction definition table**. Unless an instruction specifically modifies the program counter (e.g., *jmp*), the program counter should increase by 4 after executing an instruction.
5. There are two stopping conditions to reach the successful end of the simulated program:
    a. The program_counter reaches 4 bytes past the last instruction address (i.e., the last instruction is executed, and it is not a jump/branch).
    b. The *ret* instruction is executed while the value of %esp is 1024.
6. Upon successful completion of the simulation, the simulator should return 0 exit status.

### Instruction definitions

See the **instruction definition table** for a list of all behaviors you must implement in Part 2. Do not be daunted by the size of this table. Once you have the infrastructure for executing one instruction, many of the remaining instructions execute with minor changes. A few of the instructions are more challenging.

## Additional Rules

In addition to the specifications given above, your solution must adhere to these rules:

1. Your simulator program must be pure C code, and you may not use any inline assembly.
2. You may not include any additional header files.
3. Your simulator may not rely on any undefined C behavior, especially in implementing the *cmpl* instruction.

## Testing

Each of the provided tests (in the *tests* directory) has 3-5 associated files:

- *testname.s* — the assembly code used to produce the *.o* file
- *testname.o* — the binary machine code used as input to the simulator
- *testname.expected* — the exact output expected of the simulator
- *testname.in* (optional) — some tests also have an input file, which is used as the input for tests that use the *readr* instruction.
- *testname.c* (optional) — some tests have an associated C source file, which shows the C code used to compile the assembly, for reference purposes only

The provided *run_tests.sh* bash script runs your simulator on all of the provided tests in the *tests* directory, compares the output to the expected output, and print how many tests passed. **The test script requires that your executable be called "simulator".**

The tests in the *tests/complex* directory all use the *readr* instruction to get input. If you run the simulator manually on these tests, it looks like they are hung until you type in some inputs. See *tests/complex README* for the meaning of the input. The *run_tests.sh* script automatically provides input for these tests.

### Modifying Tests

It may be useful for you to write your own tests or modify existing tests (such as adding *printr* instructions for debugging). If you do this, **do not modify the original file in the *tests* directory.** Instead, copy it elsewhere, then use the provided assembler tool to assemble it into a binary file.

## Assembler

Use of the assembler is not required, but is helpful for debugging so you can write your own test programs.

To run the assembler, first run this command in your CADE terminal:

```
setenv LD_LIBRARY_PATH /usr/local/stow/gcc/amd64_linux26/gcc-4.9.2/lib64
```

You need to run this only once each time you login, then you can run the assembler with:

```
./assembler <input> <output>
```

The executable *assembler* expects two command-line arguments. The first is the name of the input assembly file, and the second is the name of the output binary file it should produce.

### Assembly Format

The assembler is quite fragile, and requires the input assembly to adhere to specific formatting:

- Labels are at the beginning of the line, and end with a colon. Labels can be any number of letters followed by a digit. Labels can optionally start with a '.' character.
- Instruction lines begin with one tab character, followed by the instruction name.
- Following the name is a tab character (except for the "ret" instruction).
- Following the tab character are the arguments.
- Arguments are separated by a comma and a space ", ".
- Immediate arguments start with a dollar sign "$".
- Register arguments start with a percent sign "%".
- The forms of the *movl* instruction that use an offset require the offset right before the opening parenthesis.
- In general, follow the formatting of the provided examples carefully.

## Hints

- Be careful when accessing the stack memory. Its type is a byte array, but we are storing and retrieving 4-byte values from it. Make sure to do proper casting when accessing it with pointers.
- The names of registers are not important. Represent them as an array, and simply index into that array using the register number encoded in the instruction. The only special registers you need to explicitly access by name are %esp and %eflags.
- The provided tests are categorized by simple, moderate, and complex. It is recommended you get all simple tests working first, and so on. The *run_tests* script runs them in roughly increasing order of difficulty. Most tests require support for multiple instructions.
- If you are worried about endianness, you are over-complicating things. Instruction fields are encoded based on order of significance in a 32-bit value, not by order of byte address. This is why the provided starting code reads the raw bytes into 32-bit (4-byte) data types. Furthermore, storing and retrieving 4-byte values in the stack memory works correctly using the default behavior in C.

## Grading

Your simulator will be compiled with -O2 and the provided *instruction.h*. Therefore, your entire solution must be contained in exactly one file: *simulator.c*. If you modify instruction.h, your solution will not compile and will receive no credit.

The moderate tests are worth 2x as much as the simple tests. The complex tests are worth 3x as much as the simple tests.

## Submission

Submit exactly one file *simulator.c* here. **All CS 4400 assignments are graded/evaluated on the CADE lab1 machines. Even if your solution works on some other machine, it will receive no credit if it does not work on CADE lab1.**