CS 2410 Graduate Computer Architecture
Fall 2016

Project 1: Functional Instruction Set Simulator
Assigned: October 3, 2016.
Due: October 27, 2016 by 1:00 PM.

1 Introduction

In this project, you will implement a functional simulator for a simple instruction set architecture (loosely based on MIPS). We'll call the new processor and instruction set, ***X***. Your simulator will be capable of running programs. It will model the semantics of ***X*** instructions. It will read an input program, execute the program, and output basic statistics about the program. You will integrate your simulator with the OCCAM system to setup, run and manage simulations.

2 *X* Programmer's Reference Manual

X is a simplified architecture. The native word size is 16 bits. That is, instructions and data values are 16 bits wide. Two's complement and unsigned data types are allowed for data values. There are eight 16-bit general-purpose registers ($r0 to $r7).

There are separate instruction and data memories. Instruction and data addresses are 16 bits, and both memories are byte addressable. The instruction and data memories are 65,536 bytes each.

2.1 Instructions

X has a small number of instructions, which are described in Table 1. In this table, the operator `**` is exponentiation. The operator `.` (dot) is bit concatenation. The multiply and exponetiation operators only consider the low portion of the computation (i.e., the low 16 bits, rather than the full 32 bits). The function `s_ext()` does sign extension of the immediate to a 16-bit 2's complement number. Sign extension replicates the most significant bit of the input to the function into the high order bits of the output. The function `z_ext()` does zero extension of the immediate to a 16-bit unsigned number. Zero extension fills the high order bits of the output with 0s.

In branches, ***X*** uses absolute addressing to specify a target address rather than PC-relative addressing. The branches also test conditions "equal to zero" (branch zero), "not equal to zero" (branch not zero), "less than zero" (branch negative) and "greater than zero" (branch positive).

The `put` instruction causes the contents of $rs to be output to the standard output. This instruction will assist in debugging.

The `halt` instruction causes the processor to stop and the simulation to terminate. On termination, your program should output the statistics noted below.

2.2 Instruction Format

X has three instruction formats: R, I and IX. R is used for instructions that have only registers, I is used for instructions with an immediate, and IX is used for the jump which has an "extended immediate" (11 bits). Some important points to note are:

- *Rd* is the destination, *Rs* is the first source, and *Rt* is the second source.
- *Imm8* is an 8-bit immediate. The immediate is unsigned in `bn`, `bx`, `bp`, `bz`, and `jal`. For the branches and `jal`, the immediate is shifted left by one (two bytes per 16-bit instruction). It is zero extended to a 16-bit address.

Opcode	Format	Instruction	Definition
00000	R	`add $rd,$rs,$rt`	`$rd` ← `$rs + $rt`
00001	R	`sub $rd,$rs,$rt`	`$rd` ← `$rs - $rt`
00010	R	`and $rd,$rs,$rt`	`$rd` ← `$rs & $rt`
00011	R	`nor $rd,$rs,$rt`	`$rd` ← ¬`($rs | $rt)`
00100	R	`div $rd,$rs,$rt`	`$rd` ← `$rs` ÷ `$rt`
00101	R	`mul $rd,$rs,$rt`	`$rd` ← `low16($rs` × `$rt)`
00110	R	`mod $rd,$rs,$rt`	`$rd` ← `$rs` mod `$rt`
00111	R	`exp $rd,$rs,$rt`	`$rd` ← `low16($rs ** $rt)`
01000	R	`lw $rd,$rs`	`$rd` ← `MEM[word_aligned_address($rs)]`
01001	R	`sw $rt,$rs`	`MEM[word_aligned_address($rs)]` ← `$rt`
10000	I	`liz $rd,imm8`	`$rd` ← `z_ext(imm8)`
10001	I	`lis $rd,imm8`	`$rd` ← `s_ext(imm8)`
10010	I	`lui $rd,imm8`	`$rd` ← `imm8.$rd`$_7$`..$rs`$_0$
10100	I	`bp $rd,imm8`	`PC` ← `($rd > 0 ? z_ext(imm8<<1) : PC + 2)`
10101	I	`bn $rd,imm8`	`PC` ← `($rd < 0 ? z_ext(imm8<<1) : PC + 2)`
10110	I	`bx $rd,imm8`	`PC` ← `($rd` ≠ `0 ? z_ext(imm8<<1) : PC + 2)`
10111	I	`bz $rd,imm8`	`PC` ← `($rd = 0 ? z_ext(imm8<<1) : PC + 2)`
01100	R	`jr $rs`	`PC` ← `$rs`
10011	R	`jalr $rd,$rs`	`$rd` ← `PC + 2; PC` ← `$rs`
11000	IX	`j imm11`	`PC` ← `PC`$_{15}$`..PC`$_{12}$`.(imm11<<1)`
01101	R	`halt`	`stop fetching and terminate simulation`
01110	R	`put $rs`	`output $rs to standard output (stdout)`

Table 1: Instruction definitions

- In the jump `j`, the immediate field, *Imm11*, is 11 bits. The immediate is unsigned and concatenated with the high-order bits of the program counter to form 16-bit instruction address. Similar to branches, the immediate is left shifted by one bit.
- In `liz` and `lui`, the immediate is unsigned. These two instructions can be used together to load a register with an arbitrary 16-bit immediate.
- In `lis`, the immediate is signed. The immediate should be sign extended to 16 bits.

R Format Instruction					
Bit posn	15-11	10-8	7-5	4-2	1-0
Field	*Opcode*	*Rd*	*Rs*	*Rt*	*unused*

I Format Instruction			
Bit posn	15-11	10-8	7-0
Field	*Opcode*	*Rd*	*Imm8*

3 Project Requirements

Your job is to implement a functional simulation of this instruction set. Your simulator will need to implement the semantics of the instructions, model the instruction and data memories, and model

IX Format Instruction		
Bit posn	15-11	10-0
Field	*Opcode*	*Imm11*

the registers. The simulator will allow the latency of some instructions to be configured. A program will be input into the simulator in an encoded form. The simulator will output an instruction trace and some statistics about the simulation.

The simulator is run from the command line. The simulator must be named `xsim`. It takes three arguments, which are the input file (the program to execute), the config file (specifies instruction latencies), and the output statistics file. All of the arguments are required. Here is an example:

```
$xsim inputfile configfile outputstatfile
```

3.1 Configuration

The simulator will allow the latency to be configured for the arithmetic instructions. The latency is how many clock cycles an instruction takes. The configuration file will use JSON format. On each line in the file, there is one ASCII name and one integer value (greater than or equal to 1) as a key-value pair in JSON (see below). The ASCII name is the mnemomic of the instruction to be configured. The integer value is the latency in clock cycles for the instruction. The instructions that can be configured are: `add`, `sub`, `and`, `nor`, `div`, `mul`, `mod`, and `exp`. Any instruction that is not specified in the input file has a default latency of 1 cycle. Here is an example configuration file:

```
{"div":16,
 "mod":8,
 "exp":32,
 "mul":4}
```

This configuration file configures the simulator to use 16 cycles latency for divide, 8 cycle latency for multiply, 32 cycle latency for exponentiation and 4 cycle latency for multply. The other arithmetic instrucitons all take 1 cycle latency.

3.2 Input File

The simulator will accept on the command line an input file. The input file is the first argument on the command line. The input file contains a program to execute. The program is encoded as 16-bit instructions in hexadecimal notation as ASCII text. There is one instruction per line. A line can begin with # to indicate a comment. A comment should be skipped when reading the input file. Here is an example:

```
# September 26, 2016
#
8001
0100
0220
8BFF
6800
```

I suggest that you read and parse the input file to load the contents into an array for the instruction memory. The array can hold short integer values, which are the encoded instructions (converted from ASCII hex to binary).

3.3 Output Statistics File

The simulator will output statistics about the simulation to an output statistics file. The statistics file uses JSON format. It has two major parts: register state and instruction statistics.

Register state: The register state is a list of registers and their values at the end of the simulated program's execution. For each register, the output should list the name of the register and the value in a JSON array section "Registers" (see example below).

Instruction stats: The instruction stats gives information about how many instructions were executed and the total number of clock cycles to execute the program.

For each instruction opcode, the simulator should output how many of the instructions were executed. There should be one per line, with the mnemonic and the number as a key-value pair in JSON. The simulator should also output some statistics: the number of instructions executed (the instruction count) and the total number of simulation cycles. These stats are output one per line, after the instruction counts. For the instructions execution, output the key "instructions" followed by the number. For the cycles, output the key "cycles" followed by the number of cycles. The instruction stats are reported as an array named "Stats" in JSON (see example below)

Example output in JSON:

```
{"registers":[
    {"r0":10,"r1":11,"r2":12,"r3":13,
     "r4":14,"r5":15,"r6":16,"r7":17
    }],
 "stats":[
    {"add":10, "sub":9, "and":8,
     "nor":6,  "div":5, "mul":6,
     "mod":7,  "exp":0, "lw":11,
     "sw":5,   "liz":7, "lis":0,
     "lui":7,  "bp":0,  "bn":10,
     "bx":8,   "bz":0,  "jr":0,
     "jal":0,  "j":0,   "halt":1,
     "put":0,
     "instructions":100,
     "cycles":154
    }]
}
```

3.4 Using the Sandia Structural Simulation Toolkit

If you want, you can build your simulator with the Sandia Structural Simulation Toolkit. Separate instructions and a skeleton C++ program will be given to help you get started. It's very easy!

3.5 Using OCCAM

You will integrate your simulator into OCCAM. This integration will involve writing "wrappers" for the configuration input, building the simulator, running the simulator, and parsing the statistics output. A separate example, with the necessary wrappers, will be provided to help you get started. You can simply change these wrappers to support the configurations above.

The OCCAM integration should allow configuring the simulation (changing the instruction latencies), selecting different input programs, running the program, and viewing the output statistics.

3.6 Programming Language, Libraries, etc

You are welcome to use any programming language that you wish. However, the simulator must be runnable from within OCCAM. Nevertheless, I suggest that you use C++ (or C). We selected JSON for the configuration and output files for two reasons. First, using JSON for these files will make it much easier to integrate your simulator with OCCAM. Second, there are packages (e.g., `jsoncpp`, `rapidjson`) available that you can use to parse JSON. You are welcome to use these libraries to simplify the effort to read/write the files.

4 Turning in the Project

For grading, you will both turn in the project by email (as explained below) and schedule a demo of the project with Bruce Childers. Your project will be graded primarily during the demo with possibly additional follow-up after the demo.

4.1 What to Submit

Prior to scheduling the demo, you must submit all files for the project. You must submit the files:

`xsim.cc` (the source file; you can use C++ with SST)
`all OCCAM related files`
`README.txt` (a help file; see below)

These files should be put into a single archive; you may use either a gzipped tar-ball, or a zip file. Please name your submission with your last name. For instance, `childers-project1.zip` is a good file name for Bruce Childers' submission of project 1. I will use a Mac to unpack your submission. If I cannot unpack the submission, or there are any missing files, I will not grade the project and you may receive a 0.

In the README, put your name and e-mail address at the top of the file. The README file should list what works (i.e., instructions implemented) and any known problems. If you have known problems/issues (bugs!), then you should clearly specify the problems in this file. The explanation and bug list are critical to grading. So, if you're uncertain whether to include something, then err on the side of writing too much and just include it.

Please submit by October 27, 2016 at 1:00 PM before your demonstration.

4.2 Where to Submit

Submit your files to Bruce Childers, email `childers@cs.pitt.edu`).

4.3 Demonstration

You will demo your simulator for grading. I will give you some test files (configurations and programs) to run in the demonstration. These files will *not* be provided prior to the demonstration. I will give them to you on an USB key. You should be able to load the test files into your OCCAM instance (running your simulator), configure the simulator from the OCCAM interface, and run the simulation. You should be comfortable with demonstrating the project in OCCAM, such as configuring the simulator with OCCAM, viewing results in OCCAM, and creating graphs from the results (e.g., plot a histogram of the instruction counts).

All projects will be demonstrated on October 27, 2016. Each demo will be 20-30 minutes. A sign-up sheet with appointment times will be made available prior to the demo date. If you have a class conflcit with the demo day/times, we can make accommodations. If I have not received your file submission by the due day/time, I will not grade the project and you may receive a 0.

Because the demos are intended to be short, please be prepared to quickly and smoothly show what you have done! The time allotted for each demo is strict.

5 Collaboration

In accordance with the policy on individual work, this project is strictly an individual effort. You must not collaborate with a partner. It is your responsibility to secure and back up your files.

5.1 Grading the Project

I will grade the project with multiple configurations and programs covering many (if not all) of ***X*** instructions during the demo. Your ability to easily and smoothly demonstrate the project in OCCAM will also be part of the evaluation. Primarily, the concern is functional correctness and implementation of the project requirements. To ensure that your processor is working as expected, you will find it very useful to write your own test case programs to cover all instructions.