UChicago CMSC-22200 Computer Architecture, Autumn 2016
Lab 1: Instruction-Level ARM Simulator

Instructor: Prof. Yanjing Li
TAs: Xin-Chuan (Ryan) Wu, Maria Hyun, Mohammad Reza Jokar, Gushu Li

Assigned: Thur., 9/29, 2016
Due: **11:59 pm, Thur., 10/06, 2016**

Introduction

For this assignment, you will write a C program which is an instruction-level simulator for a limited subset of the ARMv8 instruction set. This instruction-level simulator will model the behavior of each instruction, and will allow the user to run ARM programs and see their outputs. In later labs, you will use this simulator as a reference to verify that your later labs execute code correctly.

The simulator will process an input file that contains a ARM program. Each line of the input file corresponds to a single ARM instruction written as a hexadecimal string. For example, `0x91000440` is the hexadecimal representation of `addi X0, X2, 1`. We will provide several input files. But you should also create additional input files in order to test your simulator more comprehensively.

The simulator will execute the input program one instruction at a time. After each instruction, the simulator will modify the ARM *architectural state*: values stored in registers and memory. The simulator is partitioned into two main sections: the (1) *shell* and the (2) *simulation routine*. Your job is to implement the simulation routine.

The source code for the lab are provided in course website. In the `src/` directory, we provide two files (`shell.c` and `shell.h`) that already implement the shell. There is a third file (`sim.c`) where you will implement the simulator routine – `this is the only file that you are allowed to change.`

The Shell

The purpose of the shell is to provide the user with commands to control the execution of the simulator. The shell accepts one or more program files as command line arguments and loads them into the memory image. In order to extract information from the simulator, a file named `dumpsim` will be created to hold information requested from the simulator. The shell supports the following commands:

1. `go`: simulate the program until it indicates that the simulator should halt.
2. `run <n>`: simulate the execution of the machine for n instructions.
3. `mdump <low> <high>`: dump the contents of memory, from location low to location high to the screen and the dump file (`dumpsim`).
4. `rdump`: dump the current instruction count, the contents of X0 – X31, , FLAG N, Z, C, V, and the PC to the screen and the dump file (`dumpsim`).
5. `input reg_num reg_val`: set general purpose register `reg_num` to value `reg_val`.
6. `?`: print out a list of all shell commands.
7. `quit`: quit the shell.

The Simulation Routine

The simulation routine carries out the instruction-level simulation of the input ARM program. During the execution of an instruction, the simulator should take the current architectural state and modify it according to the ISA description of the instruction in `https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`. The architectural

state includes the PC, the general purpose registers, FLAGS, instruction format, and the memory image. To simpilify the work in lab 1, we separate 4 1-bit flags into 4 integer variables in our simulator. The state is contained in the following global variables:

```
#define ARM_REGS 32

typedef struct CPU_State {
  uint64_t PC;                /* program counter */
  uint64_t REGS[ARM_REGS];   /* register file. */
  int FLAG_N;        /* flag N */
  int FLAG_Z;        /* flag Z */
  int FLAG_V;        /* flag V */
  int FLAG_C;        /* flag C */
} CPU_State;

/* STATE_CURRENT is the current arch. state */
/* STATE_NEXT is the resulting arch. state after the current instruction is processed */
CPU_State STATE_CURRENT, STATE_NEXT;

int RUN_BIT; /* initialized to 1; need to be changed to 0 if the HLT instruction is encountered */
```

Furthermore, the simulator models the simulated system's memory. You need to use the following functions, which we provide, to access the simulated memory:

```
uint32_t mem_read_32(uint64_t address);
void     mem_write_32(uint64_t address, uint32_t value);
```

Note that in ARM, memory is byte-addressable. Furthermore, we will implement a little-endian architecture. This means that machine words (32 bits) are stored with the least-significant byte at the lowest address, and the most-significant byte at the highest address. To implement loads and stores of 64-bit, 16-bit and 8-bit values, you will need to use these 32-bit memory access primitives (hint: be sure to modify only the appropriate part of a 32-bit word!).

In particular, you should call `mem_read_32` and `mem_write_32` with only 32-bit-aligned addresses (i.e., the bottom two bits of the address should be zero).

The simulator skeleton that we provide includes an empty function named `process_instruction()` in the file `sim.c`. This function is called by the shell to simulate one machine instruction. You have to write the code for `process_instruction()` to simulate the execution of instructions. You can also write additional functions to make the simulation modular. (Keep in mind that you will be using the code that you write in later labs in order to validate your work.) We suggest spending time to make your code easy to read and understand, for your own benefit.

What You Should Do

Your job is to implement the `process_instruction()` function in `sim.c`. The `process_instruction()` function should be able to simulate the instruction-level execution of the following ARM instructions:

ADD	ADDI	ADDIS	ADDS	AND	ANDS
B	BEQ	BNE	BGT	BLT	BGE
BLE	CBNZ	CBZ	EOR	LDUR	LDURB
LDURH	LSL	LSR	MOVZ	ORR	STUR
STURB	STURH	STURW	SUB	SUBI	SUBIS
SUBS	MUL	SDIV	UDIV	HLT	CMP
BL	BR				

Note that for the HLT instruction, 11-bit opcode: 0x6a2, you only need to implement the following behavior: when HLT is executed, then the **go** command should stop its simulation loop and return to the simulator shell's prompt. The `process_instruction()` function that you write should cause the main simulation loop to terminate by setting the global variable RUN_BIT to 0.

The accuracy of your simulator is your main priority. Specifically, make sure the architectural state is correctly updated after the execution of each instruction. We will test your simulator with many input programs (some provided with the problem, some not) in order to ensure that each instruction is simulated correctly.

In order to test that your simulator is working correctly, you should run the input programs we provide you with and also write one or more programs using all of the required ARM instructions that are listed in the table above, and execute them one instruction at a time (run 1). You can use the rdump command to verify that the state of the machine is updated correctly after the execution of each instruction.

While the table appears to have many instructions, there are actually only a few unique instruction behaviors with a number of minor variations. *ARMv8 Reference Data* in the problem package contains the official definition for each instruction in this table (except for HLT, for which we provide a restricted definition above). You may also refer to the LEGv8 reference data, but keep in mind that there are several differences between ARMv8 and LEGv8. For conditional branch instructions (BEQ, BNE, BGT, BLT, BGE,and BLE), they have the same opcode, but you can distinguish them from Rt field. The description of LSR (immediate) and LSL (immediate) opcode in LEGv8 is different from ARMv8. What we want to implement is ARMv8, where the opcode for both LSR and LSL should be: 10 bits (0b'11 0100 1101). If the value in bit 10 to 15 of the machine code is equal to 0x3f, the operation is LSR. Otherwise, it is LSL. Please try to generate more test inputs to figure out how to decode LSR and LSL.

For the purposes of Lab 1, the branch instructions can update NEXT_STATE.PC directly to the branch target when the branch is taken.

Finally, note that your simulator does not have to handle instructions that we do not include in the table above, or any other invalid instructions. We will only test your simulator with valid code that uses the instructions listed above.

Lab Files

From the course website, you will find a source code distribution with five subdirectories `src/`, `inputs/`, `sample_outputs/`, `ref/` and `aarch64-linux-android-4.9/`.

In `src/`, we are providing you with the simulator skeleton as described above. You can compile the simulator with the provided `Makefile`.

In `inputs/`, we have written some input files for you. You should write more input files in order to be confident that your simulator is correct. Also in `inputs/`, you can find a script that will assemble ARM code into the hexadecimal format that the simulator requires. The `README` file describes how to assemble a ARM program with this script and load it into the simulator.

In `sample_outputs/`, we provide the golden sample simdumps of the test input files for your reference to see if your results are as expected.

In `ref/`, LEGv8_Reference_Data.pdf is an LEGv8 Instruction Set reference data, which is similar to ARMv8, but remember implement LSR and LSL in ARMv8 version instead of LEGv8.

In `aarch64-linux-android-4.9/`, we put Google Android compiler toolchain in the folder. As the script in `inputs/` uses the toolchain to assemble ARM code into machine code, please keep the relative path between `aarch64-linux-android-4.9/` and `inputs/` fixed.

Handin

You will be working in groups of 2, and only one copy of the code needs to be submitted for one group, but `please make sure you write down both names and cnetid at the top of the sim.c file.`

You should electronically hand in your code (all files in the `src/` directory) into lab1 folder of your own SVN repository. Contact the TAs if your handin SVN repository does not exist. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in a `inputs/` subdirectory.

Key Reminders

1. Please refer to ARMv8 reference manual Part C, chapter C3 to C6. `https://developer.arm.com/docs/ddi0487/a/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile`. ARM reference manual has both 64- and 32-bit architectures, and we are using 64-bit for our labs. For example, the test input will be `addi X0, X2, 1` instead of `addi W0, W2, 1`.

2. In ARMv8 architecture, stack pointer(SP) and zero register(XZR) share the same register, X31. To simplify the simulation, you only need to simulate register X31 as XZR, which means X31 will always be 0. All the test inputs will use X28 to store a stack pointer.

3. The description of LSR (immediate) and LSL (immediate) opcode in LEGv8 is different from ARMv8. What we want to implement is ARMv8, where the opcode for both LSR and LSL should be: 10 bits (0b'11 0100 1101). If the value in bit 10 to 15 of the machine code is equal to 0x3f, the operation is LSR. Otherwise, it is LSL.

4. You should write more input files in order to be confident that your simulator is correct. Refer to README file in inputs/ subdirectory to know how to assemble a ARM program with script asm2hex and load it into the simulator.

5. You only need to modify sim.c, and you are not allowed to change shell.h and shell.c. You must use mem_read_32 and mem_write_32 functions to implement loading and storing 8-bit, 16-bit, and 64-bit data, and you are not allowed to modified men_read_32 and men_write_32.