ELE 241 Microprocessors & Applications, Spring 2018
Lab 1: Instruction-Level ARM Simulator

Eng. Mo'men Abdelhady
Assigned: Wednesday, 2/28, 2018
Due: **Friday, 3/10, 2018**

Introduction

For this assignment, you will write a C program which is an instruction-level simulator for a limited subset of the ARM 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, `E0910003` is the hexadecimal representation of `ADDS r0, r1, r3`. 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 C:\Users\Momen\Desktop\labs\lab1. 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. (As we define below, this is when a SWI instruction is executed with a value of 0x0A.)
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 R0 – R14, R15 (PC), and the CPSR 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 the *ARM Architecture Reference Manual* that is provided on the course website. The architectural state includes the general purpose registers, the CPSR, and the memory image. The state is contained in the following global variables:

```
#define ARM_REGS 16

typedef struct CPU_State {
  uint32_t REGS[ARM_REGS];   /* register file. */
  uint32_t CPSR;
} CPU_State;

CPU_State STATE_CURRENT, STATE_NEXT;
int RUN_BIT;
```

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

```
uint32_t mem_read_32(uint32_t address);
void     mem_write_32(uint32_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 8-bit values (bytes), 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).

In addition, there are many addressing modes for memory operations in ARM. You can find them within the reference manual, and you must only be able to process those using an immediate or register offset.

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:

ADC	ADD	AND	B	BIC	BL
CMN	CMP	EOR	LDR	LDRB	MLA
MOV	MUL	MVN	ORR	RSB	RSC
SBC	STR	STRB	SUB	TEQ	TST
SWI					

For implementing these instructions, your tasks will be the following. First, implement each of these instructions as specified within the *ARM Architecture Reference Manual* accurately and completely. Each instruction should be compatible with conditional execution as described by the ARM manual. However, you only need to implement a subset of conditions: EQ, NE, GE, GT, LT, LE, and AL.

In addition, for the Data Processing Instructions (again defined in the reference manual), you must implement the S suffix for the instructions. The S suffix (ADDS vs. ADD) allows the instruction to set the CPSR's condition flag bits upon the execution of the instruction. Although the CPSR has more functionality than just the condition flags, you will only need to implement this functionality of the CPSR. You must also implement both the immediate and register operations for each Data Processing instruction. Finally, you should implement the barrel shifting and register rotating functionality defined in the reference manual as well.

Note that for the SWI instruction, you only need to implement the following behavior: if the bottom byte of the instruction's value is 0x0A (decimal 10) when SWI is executed, then the **go** command should stop its simulation loop and return to the simulator shell's prompt. If the bottom byte is any other value, the instruction should have no effect. No registers are modified in either case, except that R15 (PC) is incremented to the next instruction as usual. The `process instruction()` function that you write should cause the main simulation loop to terminate by setting the global variable RUN BIT to 0. Also of note, you should not worry about implementing any mode changes or register switches on a SWI. Thus, you must only worry about one set of registers.

NOTE: ARM assumes that the PC value is actually equal to PC+8 when the instruction at PC is being executed. This is because of ARM's pipeline which keeps three instructions in flight at once. However, we do not ask you to keep the incremented PC value. This means that whenever you use an operation that requires the PC value (B, BL), you must use PC+8 as the base offset.

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 handout, 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. You should tackle the instructions in groups: Data Processing, Memory Instructions, Branches, and so on. *Arm Architecture Reference Manual* contains the official definition for each instruction in this table (except for SWI, for which we provide a restricted definition above). Please implement only the 32-bit behavior of the instructions.

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

In C:\Users\Momen\Desktop\labs\lab1, you will find a source code distribution with two subdirectories src/ and inputs/. In src/, we are providing you with the simulator skeleton as described above. You can compile the simulator with the provided gcc. 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.

Resources

If you have not done so already, we recommend that you play with the Reference assembler/simulator to become familiar with the ARM ISA. The ARM instruction set architecture is defined in the manual that we have provided on the virtual machine. Finally, please don't hesitate to ask for help if you become stuck or if something is unclear! Take advantage of the Workshop, office hours, and labs.

Handin

You should electronically hand in your code (all files in the src/ directory) into momen.abdelhady@yahoo.com. Your code should be readable and well-documented. In addition, please turn in additional test cases that you used in a inputs/ subdirectory. If you feel the need to describe any additional aspects of your design in detail, please include these in a separate README. During the demo, we will ask you questions about your instruction simulator and test it with a number of input programs. Please be sure to allow plenty of time to get checked off (i.e., don’t come in the last 15 minutes of the workshop). Following the workshop check-off, we will test your simulator extensively with a suite of test cases so that we are confident that you have implemented all instructions correctly. This is for your benefit in later labs!