Lab 2: Instruction-Level ARM Simulator

Due **Friday 2/19** (midnight)

Instructor: Bingzhe Li
TA: Ricardo Hernandez

1. Introduction

For this lab, 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. These types of queuing-type simulators are important in determining good choices in microarchitecture design and subsequently its architecture.

This lab's objective accomplishes two tasks. First, it introduces you to software and the process in running code as well as the basics of compiling in C and what simulators mean. Second, it will introduce you to the ARM ISA. All computer architects, programmers, and digital designers know how processors work by reading the reference manual. And this lab will certainly introduce you to this.

The simulator will process an input file that contains an ARM program. Each line of the input file corresponds to a single ARM instruction written as a hexadecimal string. For example, `E282100A` is the hexadecimal representation of `add r1, r2, #10`. 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 is on Canvas. You should have seen about this on the date this lab was assigned. 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.

2. Compilation Environment

We will use the C language for our programming language and although many of you may not have had experience in this, it is similar in terms of its usage as Java. Also, this lab gives you the underlying framework to get started as well as the more complicated items in code. You are left to implementing architectural items. A wonderful introduction to the C programming language is available in the text by Y. Patt and S. Patel, which is also available on reserve in the Edmon Low Library [1]. More terse and complete texts are available in other texts including the one of the original texts on the subject by the inventors of C, Brian Kernighan and Dennis Ritchie [2]. There is also a great reference from your ECEN 3233 textbook in Appendix C [3] as well as numerous online resources.

The problem with programming languages, like C and Java, is sometimes you do not know how to compile someone else's program. For Verilog in Lab 1 and future labs, we solve this with the DO file. With the DO file you can basically compile and simulate any program from another user (i.e., `vsim -do file.do`). Most programming languages solve this similarly with a `Makefile`. A Makefile is a file that tells your system how to compile your files for a specific programming language as well as provide any needed command-line arguments necessary to have the program compile correctly. In fact, you can actually use a `Makefile` with any type of compiled or interpreted language, even Verilog. For this lab, a `Makfile` is provided in the repository and all you have to do to compile the program is type `make` provided the files are located in your subdirectory that you are working in. To compile the initial skeleton, go to the subdirectory where you files exist and type `make`.

There are many C compilers available, but you will need the GCC or clang compiler to compile the simulator and the ARM GNU compiler to compile your own ARM programs for the simulator. You can download this through the links [4] for Cygwin and [5] for ARM Toolchain. Although there are many ways to interact with the compiler, I would recommend using the command line interface. Its easier to use the

Command Line or Power Shell through Windows 10. There are other methods, but this is probably the most versatile.

We discussed in the first class lecture how to get access to GCC or Clang on a variety of systems but in case you still need to set them up:

- **Windows** - You will need to install Cygwin (`https://www.cygwin.com/`) or MinGW (`http://www.mingw.org/`) to gain access to GCC. Cygwin requires some extra steps during installation to make sure GCC and make get installed. During installation, you need to install several packages as well. 1. gcc-core; 2. make (GNU version of the 'make' utility; 3. gcc-debuginfo. A tutorial can be found in `https://stackoverflow.com/questions/47215330/how-do-i-install-gcc-on-cygwin/`. MinGW will require some extra work to get Make working so I would only recommend this if you have used it before.
- **Mac** - Installing Xcode from the AppStore will also install Clang for use in the terminal. Clang is a different C compiler compared to GCC but they both achieve the same objective for what we are doing. In addition, Clang will automatically be aliased to GCC so you can use `gcc` on the terminal and it will still work.
- **Linux** - Odds are GCC and/or Clang is already installed on your system but if it is not, you can easily install it using your distro's package manager.

3. ARMsim

The goal of this lab is to get you to think about how to model the ARM ISA. I have tried to give you a good framework for the simulator written in C. The ARMsim simulator is broken into two main C files: `shell.c` and `sim.c`. The first file handles the simulation from the user's point of view (i.e., the interface). The second file is the main simulator that makes sure instructions are modeled corrected. Each instruction is decoded and then `isa.h` handles the emulation in a function.

3.1 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:

- 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`.)
- run <n>: simulate the execution of the machine for n instructions.
- mdump <low> <high>: dump the contents of memory, from location low to location high to the screen and the dump file (dumpsim).
- rdump: dump the current instruction count, the contents of R0 – R14, R15 (PC), and the CPSR to the screen and the dump file (dumpsim).
- input `reg_num reg_val`: set general purpose register `reg_num` to value `reg_val`.
- ?: print out a list of all shell commands.
- quit: quit the shell.

3.2 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; /* current program status register */
} 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 the ARM architecture, memory is byte-addressable and 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, 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 a function named `process_instruction()` in the file `sim.c`. This function is called by the shell to simulate one machine instruction. You can also write additional functions to make the simulation modular (Keep in mind that you will probably 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.

3.3 What do you need to 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	ASR	B	BIC
BL	CMN	CMP	EOR	LDR	LDRB
LSL	LSR	MOV	MVN	ORR	ROR
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.

It is important to 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 or mdump 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. The 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. However, as noted below, adding more instructions will give you some bonus points.

3.4 Some Guidance

In order to give you some guidance and help you get started, since for many this will be something different, much of the simulator is given to you. Also, dealing with a larger program might be difficult, so we tried to avoid frustration with writing code from scratch. Most of the main functionality within the simulator works well and should be functional (i.e., `sim.c` and `shell.c`). However, if you notice within `sim.c` I put an example of the instruction call for you. For example, here is the call for `ADD`:

```
if(!strcmp(d_opcode,"0100")) {
  printf("--- This is an ADD instruction. \n");
  ADD(Rd, Rn, Operand2, I, S, CC);
  return 0;
}
```

These calls will be processed from `isa.h` and you can replicate the piece of code to simulate the other instructions. One free instruction (i.e., `ADD`) is provided within the code.

The ARM instruction set allows the condition field to be used with conditionally executing instructions. Typically, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfills the conditions encoded by the field, the instruction is executed, otherwise it is ignored. Because all instructions check this condition code, it is recommended you handle the condition code check inside the `decode_and_execute()` function.

Remember you can make additional functions to help make your code be more modular and break apart large segments into easier to understand (and test) parts.

3.5 Lab Files

In the source code file, you will find a source code distribution with three subdirectories arm2hex/, src/ and inputs/. 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. However, you should write more input files in order to be confident that your simulator is correct and that you simulate all the instructions you are required to implement. The README file in the arm2hex as well as information later described in this document describe how to assemble an ARM program with this script.

3.6 Compiling Code

As stated previously, several examples are given in the `input\` directory. Both assembly and hexadecimal versions are given. The `arm2hex\` folder contains a Python script that will compile ARM assembly programs into their proper one-instruction-per-line hex formats for the simulator. The format for this script is as follows:

```
Linux/Mac:
./arm2hex <path to ARM assembler> <input ASM file> <output HEX file>

Windows:
python arm2hex <path to ARM assembler> <input ASM file> <output HEX file>
```

If the ARM toolchain binaries are in your PATH variable, you can simply refer to the name of the ARM assembler and the script call will look like this:

```
Linux/Mac:
./arm2hex arm-none-eabi-as file.s file.x

Windows:
python arm2hex arm-none-eabi-as file.s file.x
```

If the toolchain binaries are not in your PATH variable, you'll need to specify the location of the assembler manually like so:

```
Linux/Mac:
./arm2hex /home/user/bin/arm-none-eabi-as file.s file.x

Windows:
python arm2hex 'C:\Program Files(x86)\GNU Tools Arm Embedded\9 2019-q4-major\bin\
    arm-none-eabi-as.exe' file.s file.x
```

GNU tools are important to the use of computers and have been instrumental in great advances in microarchitecture. As discussed in class, we will also use GNU tools for our work in this class. All of the ARM tools will have a program prefix of `arm-none-eabi-`. That is, if you want to the GNU binutils program `gcc`, you would type `arm-none-eabi-gcc`.

3.7 Resources

If you have not done so already, we recommend that you become familiar with the ARM ISA and how the GNU compiler functions. The ARM instruction set architecture is defined in the referenced manual that we have provided on the course website. However, the best reference is the list of ARM instructions in Appendix B in your ECEN 3233 textbook [3]. This Appendix is great, because it lists all the instructions in a concise manner and also provides a pseudo-language to understand each instruction's operation called Register Transfer Language (RTL). A copy of this section of the book will be posted on Canvas in case you do not have a copy of this book or do not have your copy nearby.

4. Handin

You should upload **a copy of your code** (compressed Zip file including all code) and **your lab report (PDF file)** to Canvas. Your code should be readable, well-documented, **compilable** and **executable**. In addition, please turn in additional test cases that you used in the `inputs/` directory. Make sure that you test your simulator extensively with a suite of test cases so that you are confident that you have implemented all instructions correctly. This is for your benefit in later labs when you can use this simulator to verify the behavior of your hardware.

If you get stuck understanding the code or figuring out what is wrong, it might help to either use a debugger or print out some variables. C has an excellent debugger called `gdb` and although it is a little hard to first get started with, it is extremely powerful. You can also use an Integrated Development Enviroment (IDE) that includes a user-friendly debugging interface such as JetBrains CLion or Eclipse with C extensions, but you will be on your own setting that up.

It is highly recommended you try to tackle things early and get most of the work done during the first week in lab. It is also important to split the work among your team to help get larger things done. As I will say for this and remaining labs that much of the work in these labs is involved. Therefore, do not procrastinate and get to it. This lab also has many practical and real world implications and can help you understand how to use C and programming languages.

In order to grade the lab, we will use some unknown assembly code that you will not see. To get the maximum grade for this lab, your simulator must be able to simulate all the instructions asked in Section 3.3. Your job is to really try to make sure your simulator covers all possible instructions that this unknown code may utilize.

4.1 Extra Credit

There are many opportunities for improving the simulator. Adding extra features or more instruction support. These extra items can earn you extra points that may help you later in the semester. Remember, if you try any extra credit or add any extra functionality that it should be **documented in your lab report** as well as upload zip file to Canvas. As with any extra credit, this should only be attempted provided you have extra time with this course as well as your other courses.

References

[1] Y. N. Patt and S. J. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. New York, NY, USA: McGraw-Hill, Inc., 2003.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1978.

[3] S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2015.

[4] "Cygwin." `https://www.cygwin.com/`.

[5] "Gnu arm embedded toolchain." `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads`.