

## CMU 18-447 INTRODUCTION TO COMPUTER ARCHITECTURE, SPRING 2014

## LAB 2: SINGLE CYCLE ARM

Instructor: Prof. Onur Mutlu

TAs: Rachata Ausavarungnirun, Varun Kohli, Paraj Tyle, Xiao Bo Zhao

Assigned: Friday, 1/24, 2014

Due: **Friday, 2/7, 2014**

## Introduction

In Lab 1, you wrote a functional simulator for a ARM machine. In this lab, you will *implement* an ARM machine using a hardware description language (SystemVerilog). The *architecture* of the ARM machine is identical to what had been specified in Lab 1. The *microarchitecture* of the ARM machine is *single-cycle*, a very simple microarchitecture that we covered in class. For full credit, you should achieve the following in this lab.

1. Implement a single-cycle ARM machine in SystemVerilog.
2. Ensure that your implementation is correct by simulating it and verifying the register dumps of the test inputs.
3. Ensure that your implementation is synthesizable (i.e., capable of being translated into logic gates) and satisfies timing constraints.

**Extra credit:** A significant amount of extra credit will be given if you also implement another ARM machine using a *microcoded* microarchitecture.

## 1 Specifications of the ARM Machine

### 1.1 Architecture

- **Instruction Set.** The machine supports all ARM instructions specified in Lab 1. As shown in the following table, there are 25 ARM instructions that the machine supports.

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

- **System Call Instruction.** The machine supports a very limited form of the `SWI` instruction as was specified in Lab 1: if the least significant byte of a `SWI` instruction has a value of `0xA`, it signals the end of the program (and also dumps out the register values). To minimize the confusion, we provided an example implementation of `SWI`.
- **PC prefetch.** Similar to Lab 1, the PC register does not directly reflect the  $PC + 8$  offset specified by ARM. Therefore, if the last instruction executed is at address `0x400020`, the PC dumped will have the value `0x400024` (next instruction to execute) instead of `0x40002C` (next instruction to execute + 8). However, to ensure correctness when using PC, every instruction that uses PC should include the + 8 offset. Therefore, the instruction  $R0 \leftarrow R15 + R1$  is equivalent to  $R0 \leftarrow R15 + R1 + 8$ . The only exception to this rule is when storing PC into the link register `R14`. In this case, you should always store the address of the next instruction.

## 1.2 Microarchitecture

The machine has a single-cycle microarchitecture: *every instruction takes exactly one cycle to execute*. Aside from correctness (as defined by the architectural specifications), this is the only constraint that we are placing on the machine's microarchitecture. As long as these two constraints are satisfied (i.e., correctness and single-cycle), you are free to implement the microarchitecture in anyway you want. To guide you along the way, we provide an abstract description of the single-cycle microarchitecture as we discussed in class.

- The architectural state of the machine (excluding memory) is stored in registers: general-purpose registers and CPSR.
- There is a global wire called the “clock (clk)” that is connected to all the registers.
- When a register sees a rising edge on the clock, the register captures the instantaneous “snapshot” of the values on its input. From then on, the register holds the captured values and feeds them to its output.
- The output from the register(s) are fed into a combinational circuit consisting of logic gates (e.g., ADD). In turn, the output from the logic gates are fed back as input to the register(s).
- At the next rising edge on the clock, the register again captures the values on its input.
- At each rising edge, the execution of an instruction is initiated. At the next rising edge, the values stored in all the registers (i.e., the architectural state) should be updated in such a way that the instruction can be considered to have correctly executed.

## 2 Lab Resources

### 2.1 Source Code

The source code is available at: `/afs/ece/class/ece447/labs/lab2`. Do NOT modify any files or folders unless explicitly specified in the list below.

- `Makefile`
- `src/` (**ALL FILES MODIFIABLE**) – You are allowed to add more source code files.
  - `arm_core.sv`: The top-level skeleton of the ARM machine. This is where you will be doing most of the coding.
  - `arm_decode.sv`: The skeleton of the instruction decoding unit.
  - `arm_defines.vh`: Useful ARM-related definitions.
  - `internal_defines.vh`: Where you can add your own definitions.
- `447src/` – Supplementary source code.
  - `regfile.v`: An array of multi-ported registers.
  - `arm_mem.v`: The memory module.
  - `testbench.v`: The testbench.
- `447inputs/` – Example test inputs.
- `inputs/` (**MODIFIABLE**) – You are allowed to add your own test inputs.
- `outputs/` (**MODIFIABLE**) – You can direct your outputs to here.
- `447ncsim/` – Config. file for the `ncsim` tool.
- `ncsim/` (**MODIFIABLE**) – Ignore. If you really want, you can implement a customized config. file.

- `dc/` (**MODIFIABLE**) – Config. files for the DC tool. You have to add any new source files in here, according to dependencies.
- `447util/` – Testing script for simulations.

The source code is written in *Verilog*, a very popular hardware description language. Since *SystemVerilog* is a superset of *Verilog*, all starter code provided should be compatible with *SystemVerilog* syntax. If you need a refresher on Verilog/SystemVerilog, please refer to the course website for tutorials. We will use the IEEE 1800-2012 standard for SystemVerilog. It is up to you if you want to use *SystemVerilog* constructs or to keep using the *Verilog* subset exclusively. However, your source code must compile, simulate, and synthesize using our Makefile, our software tools, and our supplementary source code. Please stick to the bread-and-butter features of the language, so that you don't experience esoteric incompatibilities. (Direct all questions on this matter to the technical documents of the software tools.)

## 2.2 Software Tools

We provide software tools for compiling, simulating, and synthesizing your ARM machine. In order to use the tools, remotely log into an ECE server (e.g., `ece[000-008].ece.cmu.edu` from off-campus and `ece[009-031].campus.ece.cmu.local` from on-campus) and execute the following command in your bash shell. The command will modify your shell's environment variables to setup the correct AFS paths to the tool binaries as well as the licenses for using them. *We highly recommend doing the lab on the ECE servers.*

```
$ source /afs/ece/class/ece447/bin/setup447
```

## 2.3 Makefile

We provide a `Makefile` that automates the tedious process of compiling, simulating, verifying, and synthesizing your Verilog implementation. Typing `make` without any targets or options will invoke the help screen. **Please read this help screen carefully!**

```
$ make
```

## 3 Getting Started & Tips

### 3.1 Getting Started

1. In the top-level skeleton (`src/arm_core.sv`), instantiate the general-purpose register file (defined in `447src/regfile.v`) and wire up its inputs and outputs.
2. Look at the alu unit, and correctly implement addition.
3. Wire the modules together and make sure PC is tied correctly.
4. At this point, you should be able to simulate a simple addition instruction.
5. Now that you have a feeling of how things work, study the provided source code. Before you begin to implement your ARM machine, *draw a block diagram of how you want to organize it*. The block diagram will help you think about your implementation. As your implementation matures, so will you refine your block diagram. *This block diagram will be collected!*
6. To check the correctness of any test, you can run the following command in your shell:  

```
$ make verify INPUT=447inputs/addiu.s
```

### 3.2 Tips

- Read this handout in detail. Ask questions to the TAs using Piazza.
- When you encounter a technical problem, please read the logs/reports generated by the software tools in the `outputs/` folder.

- Be cautious about how Verilog handles signed/unsigned numbers.
- The dual-ported memory module (`447src/arm_mem.v`) provides 4 write enable signals for each port. Each write enable signal corresponds to a byte in a 4-byte word.
- Memory accesses must be aligned. If you are accessing a byte, any address is allowed. But, if you are accessing a 4-byte word, the two lowest bits of the address must be zero.
- Your implementation should satisfy the timing constraints that the synthesis tool checks for. Notably, your implementation should have a minimum clock cycle of less than 25ns. The synthesis report (`timing_arm_core.rpt`) shows the value of the minimum clock cycle as well as the critical-path through your implementation that determines this value. *Based on the synthesis report, you will analyze and briefly discuss the critical-path of your implementation.*

## 4 Submission

### 4.1 Block Diagram

Please submit a *hardcopy* of a *computer-drawn* diagram of your single-cycle ARM machine. The TAs will collect this diagram during the Lab Sections and you will be responsible for explaining your design decisions to the TAs.

The diagram should be at the same level of detail as you saw in the textbook and lecture notes. All major structures (e.g. registers, muxes) should be drawn, as well as boxes for the various control logic blocks. Label all wires with their names and widths. We suggest using different colors (or line styles) to differentiate control- and data-path wires. Putting in an extra effort to keep this diagram neat and clean will definitely pay off. It is okay to utilize plenty of white space and to span multiple sheets of paper. We recommend using Inkscape (cross-platform), Adobe Illustrator (Windows/Mac), or Microsoft Visio (Windows/Mac).

### 4.2 Lab Section Checkoff

So that the TAs can check you off, please come to any of the lab sections during the week immediately *after* the deadline – i.e., during the week starting from Mon., 2/10. Please come *early* during the lab section.

During the Lab Section, the TAs may ask you:

- to answer questions about your implementation,
- to simulate your implementation using various test inputs (some of which you may have not seen before),
- to synthesize your implementation.

### 4.3 Source Code

Make sure that your source code is readable and documented. Please submit the lab by executing the following commands.

```
$ cp -r src /afs/ece/class/ece447/handin/lab2/andrewID/
$ cp -r inputs /afs/ece/class/ece447/handin/lab2/andrewID/
$ cp -r dc /afs/ece/class/ece447/handin/lab2/andrewID/
```

### 4.4 README

In addition, please submit a `README.txt` file. To create this file, execute the following command.

```
$ cp README.txt /afs/ece/class/ece447/handin/lab2/andrewID/README.txt
```

The `README.txt` file must contain the following two pieces of information.

1. A high-level description of your design.
2. A high-level description of the critical-path of your synthesized implementation.

It may also contain information about any additional aspect of your lab.

#### 4.5 Late Days

We will write-lock the handin directories at midnight on the due date. For late submissions, please send an email to `447-instructors@ece.cmu.edu` with a tarball of what you would have submitted to the handin directory.

```
$ tar cvzf lab2_andrewID.tar.gz src inputs README.txt
```

Remember, you have only 5 late lab days for the entire semester (applies to only lab submissions, not to homeworks, not to anything else). If we receive the tarball within 24 hours, we will deduct 1 late lab day. If we receive the tarball within 48 hours, we will deduct 2 late lab days... During this time, you may send updated versions of the tarball (but try not to send too many). However, once a tarball is received, it will immediately invalidate a previous tarball you may have sent. You may not take it back. We will take your very last tarball to calculate how many late lab days to deduct. If we don't hear from you at all, we won't deduct any late lab days, but you will receive a 0 score for the lab.

### 5 Extra Credit

For extra credit on this assignment, you may implement a *microcoded* microarchitecture for the ARM machine described in this handout. It is up to you as a computer architect to come up with a functional and efficient design. Because this will require a substantial amount of effort, it will be worth up to 4% additional credit *in the course* (i.e., if your grade ended up being 89% at the end of the semester, this could boost it up to 93%). Note that *you do not have to complete the full implementation to receive extra credit* – we will give partial extra credit for the portions of your design that you are able to demonstrate, and we encourage everyone to attempt this extra credit. *All* of the guidelines for Lab 2 specified in this handout also apply to the extra credit, except for the following differences.

- Block diagram: This can be drawn by hand. Please make sure to include the state machine, control logic (microsequencer and control store design), and datapath.
- Submission path: `/afs/ece/class/ece447/handin/lab2extra/andrewID/`
- Tarball (for late submissions): `lab2extra_andrewID.tar.gz`

For late submissions, we must receive the tarballs for both Lab 2 and the extra credit on the same day. If not, we'll deduct the lab late days for whichever tarball we received last.