

CMU 18-447 INTRODUCTION TO COMPUTER ARCHITECTURE, SPRING 2015

LAB 2: SINGLE CYCLE MIPS

Instructor: Prof. Onur Mutlu

TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

Assigned: Wed., 1/21, 2015

Due: **Fri., 2/6, 2015**

Introduction

In Lab 1, you wrote a functional simulator for a MIPS machine. In this lab, you will *implement* a MIPS machine using a hardware description language (System Verilog). The *architecture* of the MIPS machine is very similar to what had been specified in Lab 1. The *microarchitecture* of the MIPS 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 MIPS machine in System Verilog.
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 MIPS machine using a *microcoded* microarchitecture.

Specifications of the MIPS Machine

Architecture

- **Instruction Set.** The machine supports all MIPS instructions specified in Lab 1, *excluding* those related to multiplication and division: DIV, DIVU, MFHI, MFLO, MTHI, MTLO, MULT, MULTU. As shown in the following table, there are 45 MIPS instructions that the machine supports.

ADD	ADDU	ADDI	ADDIU	AND	ANDI	BEQ
BGEZ	BGEZAL	BGTZ	BLEZ	BLTZ	BLTZAL	BNE
DIV	DIVU	J	JAL	JALR	JR	LB
LBU	LH	LHU	LUI	LW	MFHI	MFLO
MTHI	MTLO	MULT	MULTU	NOR	OR	ORI
SB	SH	SLL	SLLV	SLT	SLTI	SLTIU
SLTU	SRA	SRAV	SRL	SRLV	SUB	SUBU
SW	SYSCALL	XOR	XORI			

- **System Call Instruction.** The machine supports a very limited form of the `syscall` instruction as was specified in Lab 1: if register `$r2` has a value of `0xA` when a `syscall` instruction is invoked, it signals the end of the program (and also dumps out the register values). To minimize the confusion, we have the `syscall` instruction already implemented in the supplementary source code that we provide.
- **Exceptions.** The machine does *not* support exceptions. The machine assumes that all instructions execute correctly and does not even bother to check for exceptions. (For example, operands to an `ADDI` instruction are *guaranteed* not to overflow their output. So the machine does not check whether the carry-outs of the most-significant two bits are different from each other.)
- **Branch Delay Slot.** The machine does *not* support a branch delay slot after a jump or branch instruction. (We are being consistent with Lab 1.) This means two things. First, the instruction

at the target-address of the jump/branch instruction is executed *immediately* after the jump/branch instruction (for a branch instruction, this is assuming that the branch condition is true). Second, some of the jump/branch instructions operate differently from their definitions in the *MIPS R4000 User's Manual*, as shown in the following table. Specifically, $PC + 4$ is linked instead of $PC + 8$.

JAL	$R[31] \leftarrow PC + 4$ $PC \leftarrow PC_{31...28} + (\text{Immediate} \ll 2)$
JALR	$R[\text{rd}] \leftarrow PC + 4$ $PC \leftarrow R[\text{rs}]$
B<cond>AL	$R[31] \leftarrow PC + 4$ if (cond) $PC \leftarrow PC + 4 + (\text{Sign_Extended_Immediate} \ll 2)$

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: the program counter and general-purpose registers.
- There is a global wire called the “clock” 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.

Lab Resources

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`
- `update.sh`
- `src/` (**ALL FILES MODIFIABLE**) – You are allowed to add more source code files.
 - `mips_core.v`: The top-level skeleton of the MIPS machine. This is where you will be doing most of the coding.
 - `mips_decode.v`: The skeleton of the instruction decoding unit.
 - `mips_defines.vh`: MIPS-related definitions. This file contains more information than is necessary for this lab.

- `internal_defines.vh`: Where you can add your own definitions.
- `447src/` – Supplementary source code.
 - `regfile.v`: An array of 3-ported registers. (A “file” is an array of registers.)
 - `syscall_unit.v`: The system call unit: exercised only when the `syscall` instruction is invoked to terminate a program.
 - `exception_unit.v`: The exception unit: not exercised.
 - `mips_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/`

The source code is written in *System Verilog*, a very popular hardware description language. If you need a refresher on System Verilog, please refer to the course website for tutorials. 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.)

Software Tools

We provide software tools for compiling, simulating, and synthesizing your MIPS 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
```

The following is the list of software tools that are involved in this lab. For your benefit, we also provide their technical documents on the course website (they have the correct versions that match with those of the binaries). If you have questions about the tools, please consult the technical documents first.

- `ncvlog`: Verilog/System Verilog compiler (Cadence) – `ncvlog.pdf`
- `ncelab`: Verilog/System Verilog elaborator (Cadence) – `ncvlog.pdf`
- `ncsim`: Verilog/System Verilog simulator (Cadence) – `ncvlog.pdf`
- `simvision`: Waveform viewer (Cadence) – `simvision.pdf`, `simviscmdref.pdf`
- `xst`: Verilog/System Verilog synthesizer (Xilinx) – `xst.pdf`

Among these, `simvision` is the only tool that you will directly invoke. All other tools will be *automatically* invoked by the Makefile that we provide. Since `simvision` requires a graphical interface, make sure that you enable X11 forwarding when you remotely log into an ECE server (i.e., `ssh -X ece000.ece.cmu.edu`).¹

¹If you are running Microsoft Windows on your local machine, you need to install an X server that does the actual rendering. Please install both Xming and Xming-fonts from <http://www.straightrunning.com/XmingNotes/>.

Makefile

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

```
$ make
```

The TAs may have to update the supplementary source code in order to fix a bug. Every time you run `make` for anything, the `Makefile` also checks for updates by querying the network file system. If the `Makefile` discovers any updates, it will alert you by flashing an annoying message on your terminal. This message will continue to appear until you run `make update` to actually apply the updates.

```
$ make update
```

Within your current working directory, the updates will patch only the files that you were not allowed to modify. On the other hand, the files/folders you were allowed to modify will not be touched.

Getting Started & Tips

Getting Started

1. In the top-level skeleton (`src/mips_core.sv`), instantiate the general-purpose register file (defined in `447src/regfile.v`) and wire up its inputs and outputs.
2. At this point, you should be able to simulate the test input `447inputs/addiu.s` by executing the following command in your shell:

```
$ make sim INPUT=447inputs/addiu.s
```
3. You should check for correctness by comparing the register dump of the simulation against the reference register dump (`447input/addiu.reg`). You can do this by executing the following command in your shell:

```
$ make verify INPUT=447inputs/addiu.s
```
4. Now that you have a feeling of how things work, study the provided source code. Before you begin to implement your MIPS 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!*

Tips

- Read this handout in detail. Ask questions to the TAs using the Q&A forum (link available on the course website).
- When you encounter a technical problem, please read the logs/reports generated by the software tools in the `outputs/` folder.
- Be cautious about how System Verilog handles signed/unsigned numbers.
- The dual-ported memory module (`447src/mips_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_mip_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.*

Submission

Block Diagram

Please submit a *hardcopy* of a *computer-drawn* diagram of your single-cycle MIPS 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).

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., 2/18. 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.

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/
```

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.

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.

Extra Credit

For extra credit on this assignment, you may implement a *microcoded* microarchitecture for the MIPS 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.