# Inf2C Computer Systems

# Coursework 2

# MIPS Processor Simulator

**Deadline: Fri, 20 Nov, 16:00**

Instructor: Aaron Smith

TA: Siavash Katebzadeh

The aim of this assignment is to write a simulator for a 5-stage multi-cycle MIPS processor. A simulator is nothing more than a functional model of a processor that mimics the behavior of a real processor but is written in a high level language. Your simulator, written in C, will read a memory file consisting of MIPS instructions and data, "execute" the instructions and output the state of the processor and various statistics of the execution. To get you started, you will be provided with a skeleton implementation of the simulator that you will need to extend. You are strongly advised to read up on MIPS processor in the lecture notes and course textbook and to commence work as soon as possible.

This is the second of three assignments for the Inf2C-CS course. It is worth 20% of the overall course mark.

Please bear in mind that the guidelines on academic misconduct from the Undergraduate year 2 student handbook are available on the following link http://web.inf.ed.ac.uk/infweb/student-services/ito/students/year2.

## 1 Overview

In this assignment, you are provided with a skeleton of the simulator, which includes the 5-stage MIPS processor and memory. You will need to extend the provided code to support a set of specified MIPS instructions. By reading from input files, the skeleton feeds the memory and register state file with instructions and data, and simulates the

standard 5-stage instruction execution cycle. The output functions are already implemented and called in the skeleton.

**Skeleton Organization:** The skeleton is broken down into four source code files. Each file accomplishes particular tasks. You are allowed to modify and submit only certain files. The functionality and permission to modify for each file are described in Table 1.

| Filename | Functionality | Modifiable |
|---|---|---|
| `mipssim.c` | Multi-cycle MIPS processor (datapath + control) | **Yes** |
| `mipssim.h` | Data structure definitions for datapath | No |
| `memory_hierarchy.c` | Memory hierarchy implementation | No |
| `parser.h` | Reading and parsing input files | No |

Table 1: Source Files

*mipssim.c:* This file describes the multicycle MIPS processor as studied in class. The processor consists of the following core components: PC, Pipeline registers (IR, A, B, MDR, and ALUOut), Programmer-visible registers (in the register file), ALU, ALU Control, and Control.

The processor's functionality can be broken down into the following logical stages: *instruction_fetch*, *decode_and_read_RF*, *execute*, *memory_access* and *write_back*. Note that each stage updates some architectural or microarchitectural state. For instance, the instruction_fetch stage updates the IR. The write_back stage updates the RF.

Furthermore, this file handles the Control component by implementing a finite state machine. The state machine can be found in a function named *FSM*.

*mipssim.h:* This file defines the following required data structures for the MIPS processor:

- *ctrl_signals*, which control the datapath and are updated on a cycle-by-cycle basis by the control FSM

- *instr_meta*, which stores information about the instruction currently stored in the IR

- *memory_stats_t*, which consists of memory stats for loads, stores and instruction fetches

- *pipe_regs*, which includes the PC and microarchitectural registers of the processor (IR, A, B, ALUOut and MDR). For convenience, we refer to these registers as *pipeline registers* to indicate that they are spread out over the datapath (unlike the programmer-visible registers, which are all located inside the register file).

On any given cycle, the complete state of the processor is stored in a structure called *architectural_state*. This structure includes the current clock cycle since the start of execution (*clock_cycle*), state of the current instruction's execution (e.g., INSTR_FETCH or DECODE), current values of control signals (*control*) and memory stats (*mem_stats*). This structure also includes an array of *registers*, which models a register file, as well

as the *memory*. Finally, *architectural_state* also includes the pipeline registers, which are maintained in two *pipe_regs* structs: *curr_pipe_regs* and *next_pipe_regs*. The former (*curr_pipe_regs*) is used within a cycle to *read* the value of a given register. Meanwhile, a value that needs to be written into a pipeline register at the end of the cycle should be stored in *next_pipe_regs*. At the end of each clock cycle, *curr_pipe_regs* is updated with values from *next_pipe_regs*; this functionality is already provided for you.

**IMPORTANT:** your program must ensure that the state of the processor as represented by *architectural_state* is correct on any given cycle. To accomplish that, you must maintain the *control* signals, *curr_pipe_regs*, *registers* and *memory*. These updates must happen inside the designated functions in **mipssim.c**.

**memory_hierarchy.c:** This file provides the memory interface via two functions: *memory_read*, which is used for reading from memory and *memory_write*, which is used for writing to memory. By default, reads and writes access the memory directly, i.e., there is no cache.

**parser.h:** This file contains the implementations of reading instructions and data from input files. The format of input files are described in Sec 3.
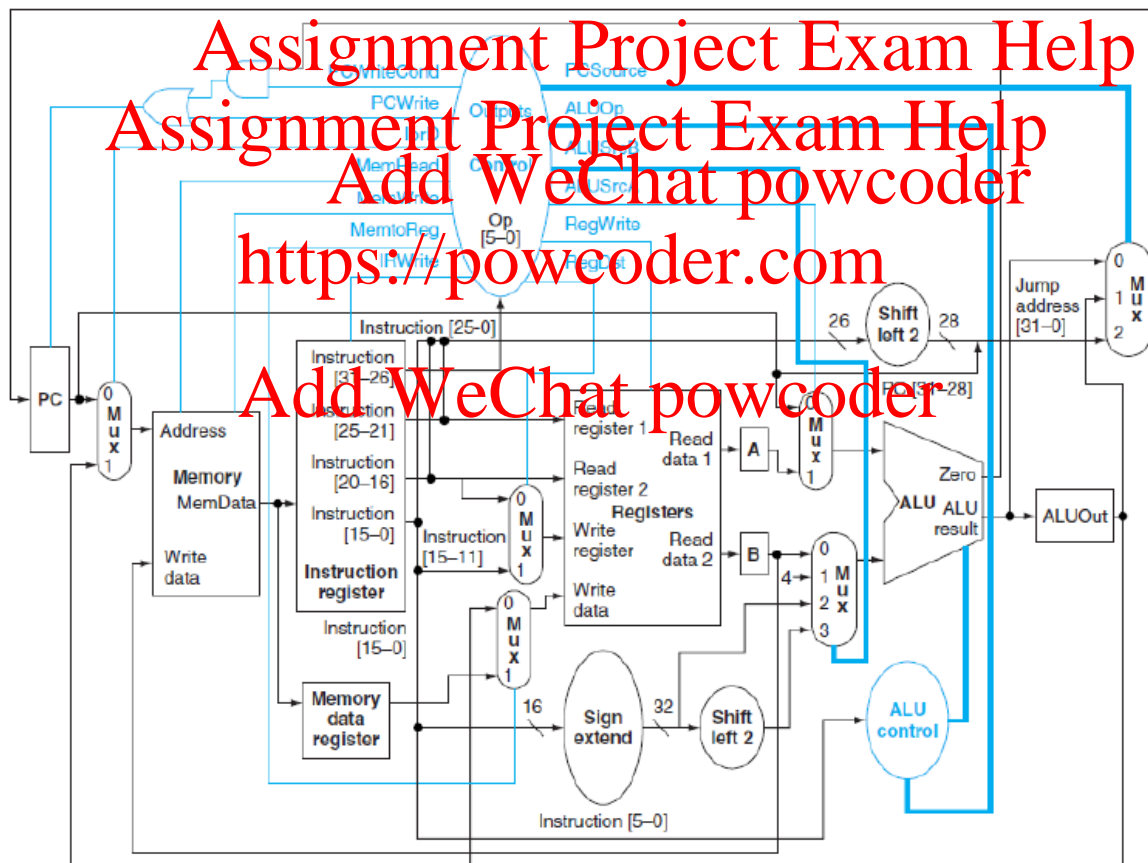


Figure 1: Multi-cycle MIPS processor

3

## 2 MIPS Processor

In this task, your job is to complete the datapath of a MIPS processor provided in the skeleton *mipssim.c*. Figure 1 illustrates the processor datapath and the control signals. In the skeleton, there are five functions: *instruction_fetch*, *decode_and_read_RF*, *execute*, *memory_access* and *write_back*, corresponding to the five stages of the processor. Some of these functions are incomplete, and you need to implement them according to the MIPS data and control paths as specified in the P&H chapter on the multi-cycle processor (available on Learn).

Additionally, you are required to complete the Control implementation in *mipssim.c*. In every cycle, the Control component controls the state of each stage of the processor using a finite state machine (FSM). Figure 2 illustrates the FSM used in the Control component of the multi-cycle MIPS processor as discussed in class. Note that the names of the states in Figure 2 may be different from the source files provided to you. Each circle in Figure 2 corresponds to one state of the FSM and lists (1) the values for mux select signals that are set in that state, and (2) all enable signals that are set in that state. For instance, State 0 (i.e. *instruction fetch*) shows that enable signals *MemRead*, *IRWrite* and *PCWrite* must be set.

In the skeleton, a global struct named *arch_state_control* contains the control signals generated by the Control component. You **must use these signals** and complete the *FSM* function to control the 5 stages. You may add new FSM states if necessary.

The skeleton already implements both the datapath and the control for the *ADD* instruction. Your task is to extend the code to support the following instructions: *LW*, *SW*, *ADDI*, *ADDU*, *J*, *BQE* and *SLT*. Note that Figure 2 does not show the required states and transitions for an *ADDI* instruction. You must design the state machine for ADDI by yourself (**Hint:** *ADDI* is similar to *ADD* except it has one different operand).

## 3 Input Files

The skeleton will read two files:

1. *The memory state file:* a text file, which is a mix of instructions and data represented by a sequence of 0 and 1 characters, one word (32 bits) per line. Note that the *first* non-comment line of the memory file is always an instruction, which starts at address 0x0. Subsequent words are placed in consecutive memory locations. For this assignment, a special instruction with opcode *111111* (in binary) is considered as the End-Of-Program (EOP) instruction, which terminates the program. In the provided skeleton, there are two memory state file examples: `memfile-simple.txt` and `memfile-complex.txt`.

2. *The register state file:* a text file, which contains the initial state of programmer-visible registers. This file has up to 31 uncommented lines for registers (i.e., all of the programmer-visible registers except $0) starting with register $1. Each line specifies a decimal value to which the corresponding register will be initialized. If fewer than 31
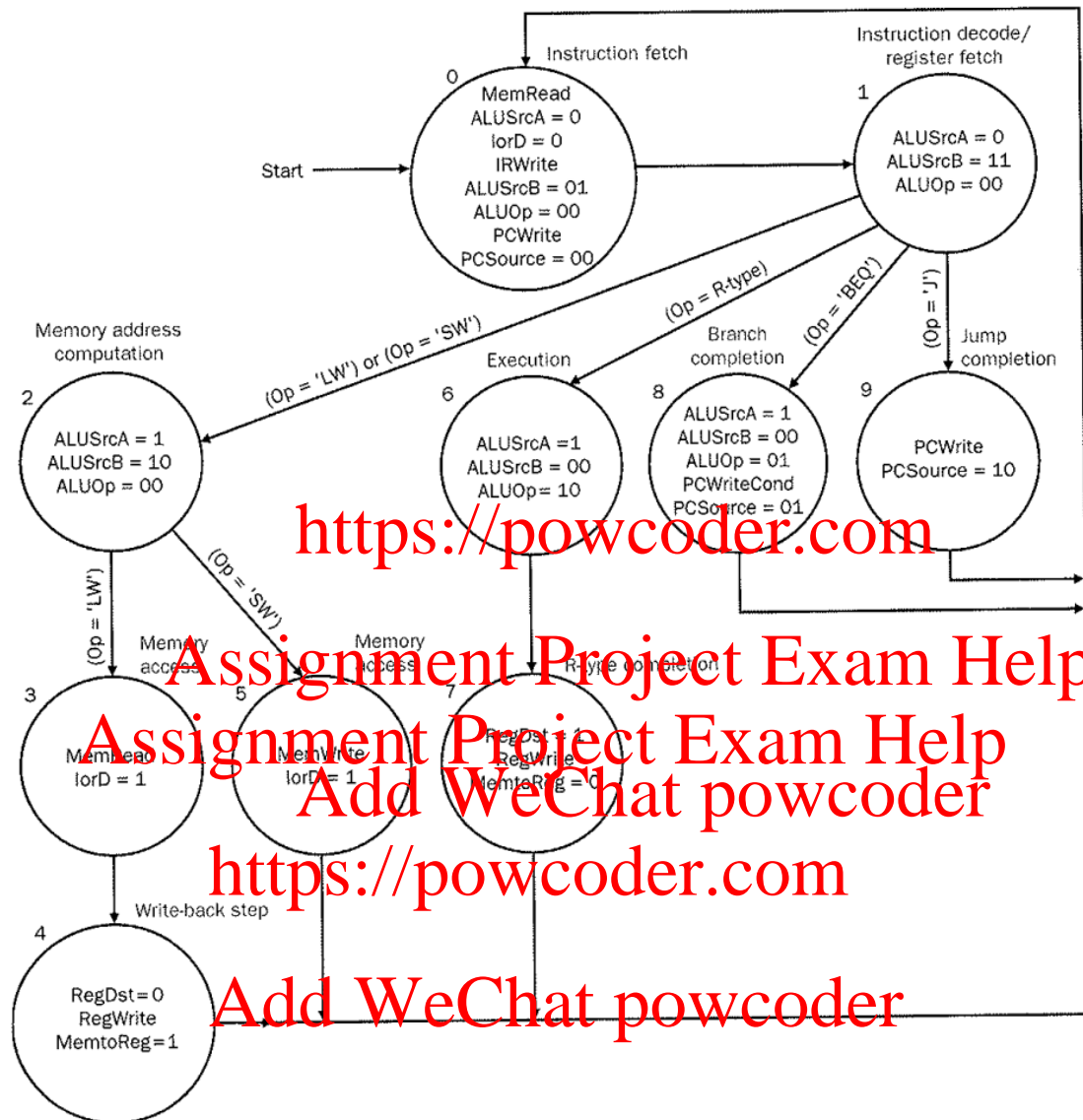
Figure 2: Finite state machine used for a multi-cycle MIPS processor

values are specified, the remaining registers will be initialized with zeros. In the provided skeleton, there is one register state file example: `regfile.txt`.

# 4    Output Format

You do not need to generate any output for marking purposes. Instead, you must ensure that in each cycle all relevant variables/structs of the *arch_state* struct are properly updated. The automated marking will check the *arch_state* struct in each cycle, checking the correctness of the control signals (i.e., the *arch_state.control* field), and datapath

state (i.e., fields: *arch_state.curr_pipe_regs* and *arch_state.registers*). The automated marking will use the functions *marking_after_clock_cycle()* and *marking_at_the_end()*. Make sure that you **do not use or modify** these functions!

# 5    Debugging

1. To verify correctness of your implementation, you should write your own memory and register state files. For verifying the datapath, start testing each instruction individually.

2. Your simulator will be tested with memory and register state files different from the provided ones.

3. You can use *printf* function for debugging your code. It will not affect your marking.

# 6    Compiling and Running the Simulator

You **must** compile the simulator on the DICE machines with the following command:

```
gcc -o mipssim mipssim.c memory_hierarchy.c -std=gnu99 -lm
```

Note that this is the exact command we will use for compiling your code for marking purposes. Compiling the source files creates an executable *mipssim*. Make sure that your simulator both compiles with the exact command and runs on a **DICE** machine without errors and warnings. Otherwise, you will receive a **0** mark.

The following are examples of invoking the simulator with valid command-line parameters.

```
./mipssim 0 memfile-simple.txt regfile.txt
```

Where `mipssim` is the name of the executable file and `0` indicates cache is disabled. Additionally, `memfile-simple.txt` is the name of the memory file and `regfile.txt` is the name of the register state file. Both memory and register state files are located in the same directory as `mipssim`.
**Note:** in this assignment cache is always disabled, and you **must** pass `0` as the first argument after the executable file.

# 7    Submission

The skeleton code and example inputs are contained in the course work Git repository on GitLab: https://git.ecdf.ed.ac.uk/asmith47/inf2c-cs-20/cw2

You will modify the skeleton files provided for the tasks and push your changes to your fork of the coursework Git repo on GitLab. You can push changes to GitLab any time up until the submission deadline. We will not mark any changes pushed after the submission deadline has passed.

The submission deadline is 4pm on November 20, 2020.

For additional information about late penalties and extension requests, see the School web page below. Do **not** email any course staff directly about extension requests – we are unable to grant them. Instead, follow the instructions on the web page: `http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests`

# 8 Assessment

The assignment will be auto-marked. Your solutions will be evaluated with a number of test inputs. For each task, your mark will be proportional to the pass rate of the tests.

# 9 Similarity Checking and Academic Misconduct

You must submit your own work. Any code that is not written by you must be clearly identified and explained through comments at the top of your files. Failure to do so is plagiarism. Detailed guidelines on what constitutes plagiarism can be found at: `http://web.inf.ed.ac.uk/infweb/admin/policies/guidelines-plagiarism`. All submitted code is checked for similarity with other submissions using the MOSS¹ system. MOSS has been effective in the past at finding similarities. It is not fooled by name changes and reordering of code blocks.

# 10 Questions

If you have any questions about the assignment, please start by checking existing discussions on Piazza – chances are, others have already encountered (and, possibly, solved) the same problem. If you can't find the answer to your question, start a new discussion. You should also take advantage of the drop-in labs and the lab demonstrators who are there to answer your questions.

November 6, 2020

---

[1] `http://theory.stanford.edu/~aiken/moss/`