Lab 8: a MIPS Simulator (Project)

Please read the entire assignment.

Objectives

The goal of this lab is to implement a MIPS simulator. The simulator reads encoded MIPS instructions from a file and simulates the execution of the instructions on a single cycle MIPS processor. This lab helps you understand the design of MIPS processors.

The simulator supports the following instructions. Note that addi is included.

Table 1: List of instructions

Type	Instruction	Opcode	Function code
R-Type	add	00 0000	10 0000
R-Type	sub	00 0000	10 0010
R-Type	and	00 0000	10 0100
R-Type	or	00 0000	10 0101
R-Type	slt	00 0000	10 1010
I_Type	addi	00 1000	
I-Type	lw	10 0011	
I-Type	sw	10 1011	
I-Type	beq	00 0100	
J-Type	j	00 0010	

Deliverables and Deadlines

The project has two phases (see descriptions in a later section). After the completion of each phase, submit the core_sc file (a single file) in HuskyCT.

In Phase 1, generate the control signals. It is due by the end of Thursday, 11/07/2019.

In Phase 2, construct a complete MIPS core. It is due by the end of Friday, 12/06/2019.

If you cannot complete Phase 1 before its deadline, you will have to continue to work on it. Late submissions of Phase 1 are accepted, with penalty, until the deadline of Phase 2.

Project Site

The project web site is http://zshi.uconn.edu/mips_sim/.

You can download template files and test vectors. You can also find a link to a page where you can check the signals your simulator generates in Phase 1.

Template

A template is provided (in both Java and Python). Study the code and learn how to do things in the simulator. For example, access memory, read register files, use combinational modules like MUX.

The program reads a list of instructions from a text file and store the instructions in instruction memory. The input files to your simulator are generated by MARS. You can find information on how to generate your own test files on the project site.

Before the simulation starts, the program sets PC to the address of the first instruction in the instruction memory. The program then enters a function that simulates a single cycle MIPS processor. The function is located in **core_sc.*** file. **It is the only file you need to modify and submit.**

The function has a loop and each iteration of the loop simulates what is happening in a cycle. At the beginning of each cycle, all the registers are updated by calling the clock() method of registers and register files. After the registers are updated, the program fetches instruction from the address in PC (which is just updated) and increase PC by 4. After the instruction is fetched, the program displays instruction address and instruction.

The simulation completes when it has run for a specified number of cycles, or when the address of the instruction is out of range (i.e., no instructions are loaded to that address).

The simulator is a console program. Ask questions if you do not know how to run Python scripts in a console (or a terminal). The simulator takes a few arguments. The following is one way to run the Python code, which may be different on your system.

```
python3 mips_sim input_file [options]
```

The first argument is the input file that contains instructions. After the input file, you can specify some options.

- A nonnegative number. The number specifies the number of cycles the simulator runs. If it is 0, the simulation continues until PC is out of range.
- `-v`. This flag turns on the verbose mode. You can see the activities in register file and memory, for example, write to memory.
- `-p1`. This flag disables the phase 2 code. You can let simulator do Phase 1 only even if you have completed Phase 2.

Project Phases

Phase 1: Control and Other Signals

Study the pseudocode on page 6 to have a global picture about the simulator and the work in Phase 1.

Your main goal in this phase is to generate control signals and some other signals that do not depends on data.

In the template, you can find a Signals class, which keeps all the signals you need to generate. You can call function `print_signal_1()` to print all the signals you need to generate in Phase 1. Table 2 lists important signals.

You may find the following resources are helpful.

- The diagram of a MIPS processor we have studied is in Figure 4.24 in the textbook (also in f04-24.pdf). Some signals are not named in the textbook.
- Figures 4.12 and 4.13 specifies how the 4-bit ALU operation signal (the ALU control input signal in Figure 4.12 and the Operation signal in Figure 4.13) is generated from ALUOp and Funct field.
- Figure 4.16 (description of control signals) and Figure 4.18 (values of control signals) describe seven of the control signals generated by the main control unit.

In this phase, **you do not need to change run()**. You only need to implement the following functions that simulates combinational modules.

- A function that generates signals by extracting bits from the encoded instruction (like opcode, rs, rt, rd, immediate, and so on).
- Main control unit. It takes the 6-bit opcode as input and generates the nine (9) signals. **Note that you need support the addi instruction.**
- Sign-extend unit. The unit extend a 16-bit 2's complement number to a 32-bit 2'c complement number. Make sure you can handle both positive and negative values. In Java, the output is an 'int' type.
- ALU control unit. The input of this unit is ALUOp form the main control unit and the funct field from the instruction. The output is the 4-bit ALU operation signal.
- A function that computes the branch address.
- A function that computes the jump address from (PC + 4) and the instruction.

Some signals have 'don't care' values so designers can simplify the circuit. In your implementation, set all don't care values to 0.

Print out the value of all signals (by calling print_signal_1()). Check the output of your code at the project site. Please do not attack or abuse the site.

You will deal with bits in Python/Java. **Do not convert integers to strings for bit operations**. See the bit operation examples on page 7.

Phase 2: Putting everything together

In this phase, you complete the rest of the single-cycle MIPS processor that supports all the instructions in Table 1. Study the pseudocode on the last page.

Connect modules with signals (or send signals to correct modules). Almost all control signals are already been generated in Phase 1. You may have to add a few more like PCSrc (1 for taken branches and 0 otherwise).

The tasks you need to do in this phase include:

- Read register file.
- Use ALU.
- Access data memory.
- Prepare to write to the register file.
- Computer PC for the next cycle.

In each step, you will generate new values for some signals, which you can find in Signals class.

At the end of each cycle, function print_signal_2() prints out the values of important signals in Phase 2.

Note that **you cannot use conditional statements in this phase**. The components you need are in the combinational module (or hardware module), including MUXes and AND gates.

Table 2: Control signals in the single-cycle-execution processor

Signal	Width	Description
PC	32	The address of an instruction. Must be a multiple of 4. (Given)
instruction	32	Instruction encoded. (Given)
opcode	6	The opcode of the instruction (bits 31:26).
rs	5	The rs field (bits 25:21). Register 1 to be read.
rt	5	The rt field (bits 20:16). Register 2 to be read.
rd	5	The rd field (bits 15:11). Register 2 to be read.
funct	6	The funct field from the instruction (bits 5:0).
immediate	16	The immediate from the instruction (bits 15:0).
RegDst	1	Specifies where to get the destination register number. 1 for rd.
ALUSrc	1	Specifies where to get the second operand for ALU. 0 for value from RF.
RegWrite	1	If asserted, register file will perform a write operation.
MemRead	1	If asserted, data memory performs read operation.
MemWrite	1	If asserted, data memory performs write operation.
MemtoReg	1	Select the value to be written into the register file. 1 for memory.
Jump	1	When asserted, use Jump_address as the next PC value.
Branch	1	When asserted, use Branch_target as the next PC value.
ALUop	2	Indicating ALU operation types. Used in ALU control unit.
Write_register	5	The register number indicating the register to be written.
Sign_extended_immediate	32	The output of the sign-extend module. Sign extension of bits 15:0.
ALU_operation	4	The output of ALU control unit specifying ALU function.
Branch_address	32	The output of the adder that computes branch target address.
Jump_address	32	The target address generated for jump instructions.

Pseudocode of the main simulation function

```
# The staring PC has been presented at the input port of PC register.
While (Need to run more cycles) {
    Send clock signals to PC and the register file.
```

```
        Get the updated PC value and send it to the instruction memory.
        PC_4 = PC + 4.
        Read the instruction from I_Mem.
        Exit from the loop if no more instructions to fetch.
        # Everything above has been implemented.
        # Phase 1
        Generate the signal values that can be extracted from instr.
        Let the main control unit to generate control signals.
        Perform sign extension.
        Find out Write Register (use rd or rt?).
        Generate ALU_operation with ALU control.
        Compute branch address.
        Compute jump address.
        # Phase 2
        Read register file (RF). (Set input, and retrieve output).
        Use ALU. (Call function with proper input).
        Access D_mem. (Set input, call run(), and retrieve output).
        Set correct values at the input of RF (Write_register, …)
        Find out PC to be used in the next cycle (PC_new).
        Set correct values at the input of PC (PC_new and write enable).
}
```

Examples of bit operations in Python

```
# Assume v is an integer and bit 0 is the least significant bit.
# To extract lowest 6 bits of v into r
r = v & 0x3F
# To extract bits 9 to 5 of v into r
r = (v >> 5) & 0x1F
# To test bit 9 of v:
if (v & 0x200) != 0:   # or if (v & 0x200):
```

```
if (v & 0x200) == 0:

# To test if bits 10 and 9 of v are 0b10

if (v >> 9) & 0x3 == 2:          # or == 0b10

# To generate a mask that has lowest 5 bits set to 1

mask = (1 << 5) - 1
```