ECS 154B Lab 2, Spring 2017
Due by 11:59 PM on April 21, 2017
Via Canvas

Goals

- Build and test a single cycle MIPS CPU that implements a subset of the MIPS instruction set.
- Design a combinational logic control unit.

Description

In this lab, you will use Logisim to build a single cycle CPU to understand the MIPS control and datapath signals. To test your CPU, you will run an assembly language program given to you, and simulate the operations in Logisim. You will be given several functional blocks to help you out.

Provided Blocks

In creating your CPU, you are allowed to use all of the features and modules available in Logisim. You will be given an empty project to start your lab, which includes an implementation of an ALU and a Register File. Those blocks are described below.

ALU

- **A**, **B**: The 32 bit data inputs to the ALU.
- **ALUResult**: The 32 bit result of the ALU operation.
- **Overflow**: Not used in this lab.
- **Shamt**: 5 bit shift amount. Not used in this lab. Connect a constant zero to this input.
- **Zero**: A flag bit that is set when the ALU result equals zero (all bits are low).
- **ALUCtl**: The 4 bit control input, as described in the following table:

Instruction	ALUCtl3	ALUCtl2	ALUCtl1	ALUCtl0
`XOR`	0	0	0	0
`SLTU`	0	0	0	1
`SLT`	0	0	1	0
`AND`	0	0	1	1
`NOR`	0	1	0	0
`SUB`	0	1	0	1
`OR`	0	1	1	0
`ADD`	0	1	1	1

More instruction types to be supported will be added in Lab 3, thus the extra ALUCtl signal.

Register File

- **Clock**: The main clock signal.
- **RdAddr1**, **RdAddr2**: The 5 bit register addresses to read. MIPS has 32 registers in its register file.
- **RdData1**, **RdData2**: The 32 bit data values from the read registers.
- **WrAddr**: The 5 bit register address to write.
- **WrReg**: A control signal that causes the register file to be written to when high. If **WrReg** is low, no register values will change.
- **WrData**: The 32 bit value to store in the register specified by **WrAddr**, if **WrReg** is set.

Blocks to Implement

You will need to add additional logic blocks to your diagram: a PC, an instruction memory, a data memory, and combinational logic circuits to decode the current instruction word and enable the respective datapaths.

PC

- Use a 32 bit Register as your PC.
- **Please note that MIPS is a byte addressable architecture, and therefore PC is incremented by 4, not 1.**

Instruction Memory - ROM

- Use an 8 bit address, 32 bit data ROM from the memory library as your instruction memory, to which you can load your hex code.
- A sample test program will be included. To load the instructions into your ROM, right-click the ROM and select *Edit Contents...* In the Hex Editor that pops up, click *Open* and select **instructions.lst**, which is provided with the assignment files.
- The corresponding MIPS instructions are in **instructions.mps**.
- You can assemble your own test programs using Sean Davis' ASIDE program, available at `http://csiflabs.cs.ucdavis.edu/~ssdavis/50/`. After compiling, you will need to edit the `.lst` file that corresponds to your code to match the input that Logisim expects.
- The MIPS architecture is byte addressable, so the PC is incremented by 4 to reach the next instruction. For example the first instruction will be at `0x00000000`, the second at `0x00000004`, the third at `0x00000008`, and so on.

Data Memory - RAM

- Use an 8 bit address, 32 bit data RAM from the memory library as your data memory. The separate load and store ports option is easier to use but you can use the other two options if you'd like.
- **A**: the 8 bit address of the word you want to access.
- **D**, left side: the 32 bit word to be written or stored at the address specified by A.
- **str**: When high, the value on the left D is stored at address A.
- **sel**: When 0, the chip is disabled. Set to high or leave floating.
- ^: the main clock signal.

- **ld**: When high, the value on the memory at address A is placed on the right D.
- **clr**: When high, sets all of the values in the RAM to 0. Either set low or leave floating.
- **D**, right side: the 32 bit word to be read or loaded from the address specified by A.

Control Unit

The control unit outputs the various signals for the datapath that are specific to each instruction. You can find more information about how to implement this in the lecture notes or in the book. **You must implement your control unit with combinational logic for this lab - you may not use a ROM.** You may also not use a MUX with constant inputs to generate your control signals.

MIPS Architecture

Instructions to Implement

A PDF, **MIPS Architecture.pdf**, was included in the archive to help you understand the encoding of the instructions that you will need to implement. They are as follows:

- Data instructions: `ADD, ADDI, AND, ANDI, NOR, OR, ORI, SLT, SLTI, SLTU, SUB, XOR`
- Memory access instructions: `LW, SW`
- Control flow instructions: `BEQ, J, JAL, JR`

`JAL` and `JR`

The book does not explain too much about `JAL` or `JAR`, nor give any hints on how to implement them, so it will be up to you to figure out what you need to do.

`JAL`

`JAL` (jump and link) is just like `J`, except it stores the contents of `PC + 4` in register `$31`.

- *Use*: `jal offset`
- *Effect*: `$31 = PC + 4.  PC[31...0] = PC + 4[31...28], Inst[25...0], 00`
- *Encoding*: `0000 11ii iiii iiii iiii iiii iiii iiii`, where the `i`'s are the bits encoding the immediate value.

`JR`

`JR` (jump, register) sets the PC to the value contained in register `s`.

- *Use*: `jr $s`
- *Effect*: `PC = $s`
- *Encoding*: `0000 00ss sss0 0000 0000 0000 0000 1000`, where `s` is the address of the register.

Probes

The following probes are included to help you debug your circuit:

Label Name	Radix	Description
PC	Unsigned Decimal	The current value of the PC.
WrReg	Binary	1 if writing to the register file, 0 otherwise.
WrAddr	Unsigned Decimal	The address of the register that is going to be written to.
WrData	Signed Decimal	The value that is going to be written to the register.
MemWr	Binary	1 if writing to memory, 0 otherwise.
MemData	Signed Decimal	The data to be written to memory.
Branch	Binary	1 if taking a branch, 0 otherwise.
Jump	Binary	1 if executing the jump instruction, 0 otherwise.
Jal	Binary	1 if executing the jump and link instruction, 0 otherwise.
Jr	Binary	1 if executing the jump register instruction, 0 otherwise.

Grading

To grade your assignment, the TAs will run your CPU with the instructions in the file **instructions.lst** and look at the contents of your registers after the program is finished. If you look at the comments in **instructions.mps**, it will tell you what the final states of the registers and memory locations should be.

- 50% Implementation
  - 50% for correct implementation of non-control instructions: registers 1 to 13 are all correct.
  - 50% for correct implementation of control instructions `BEQ`, `J`, `JAL`, and `JR`. In addition to the above, register 14 is correct, registers 27 to 30 are zero, register 31 is correct, and the program ends in the infinite loop.
  - Partial credit at the grader's discretion.
- 50% Interactive Grading

Submission

Warning: read the submission instructions carefully. Failure to adhere to the instructions will result in a loss of points.

- Upload to Canvas the zip/tar of your .circ file along with a README file that contains:
  - The names of you and your partner.
  - Any difficulties you had.
  - Anything that doesn't work correctly and why.
  - Anything you feel that the graders should know.
- **Copy and paste the README into the text submission box when you are submitting your assignment**, as well.
- Only one partner should submit the assignment.
- You may submit your assignment as many times as you want.
- You have 2 slip days to use for this assignment.

Hints

- You can use Logisim's Analyze Circuit tool, under the Project menu, to automatically construct combinational circuits for you. This can be an extremely time-saving tool, so make an effort to learn how to use it.
- Test and debug in steps. Start with a subset of the lab requirements, implement it, test it, and then add other requirements. Performing the testing and debugging in steps will ease your efforts. For example, you could implement the R-type and `ADDI` instructions, and verify that those work as intended. Then, add the branch instruction, and finally add the memory access instructions.
- Think about the hardware you are creating before trying it out. The text is necessarily vague and leaves out details, so do not simply copy the figures and expect your CPU to work.
- Remember that, though the PC and data addresses are 32 bits, the instruction memory and data memory addresses are only 8 bits. Be careful which bits you use to address the two memories.
- It is helpful to construct an Excel spreadsheet of the instructions and the various control signals needed. This way, errors in the control logic can be easily identified.