Project 4 - Calculator CPU

CS 0447 — Computer Organization & Assembly Language

Due Sunday April 19, 2015 at 11:59pm

A simple device such as a calculator requires a processor to process users' input and display output on an LCD screen. For this project, you are going to build a 32-bit MIPS processor which will be used in a calculator.

Part I: Modify Your Project 1

In project 1, you have a chance to write an assembly program to control the Simple Calculator (Mars Tool). Your program will be used again in this project to run on your own CPU. Note that your CPU will only support a small set of instructions as follows:

- **Arithmetic Operation**: `add`, `addi`, and `sub`
- **Logical Operations**: `and`, `andi`, `or`, `ori`, and `nor`
- **Shift Operations**: `sll` and `srl`
- **Comparison Operations**: `slt`, `slti`, `beq`, and `bne`
- **Jump Operations**: `j`

The first part of this project is to modify your project 1 (if necessary) in Mars to use only above 15 instructions. Note that for any instruction with immediate value (e.g., `addi`, `ori`, etc), make sure the immediate value is less than 16-bit two's complement.

Behavior of Calculator

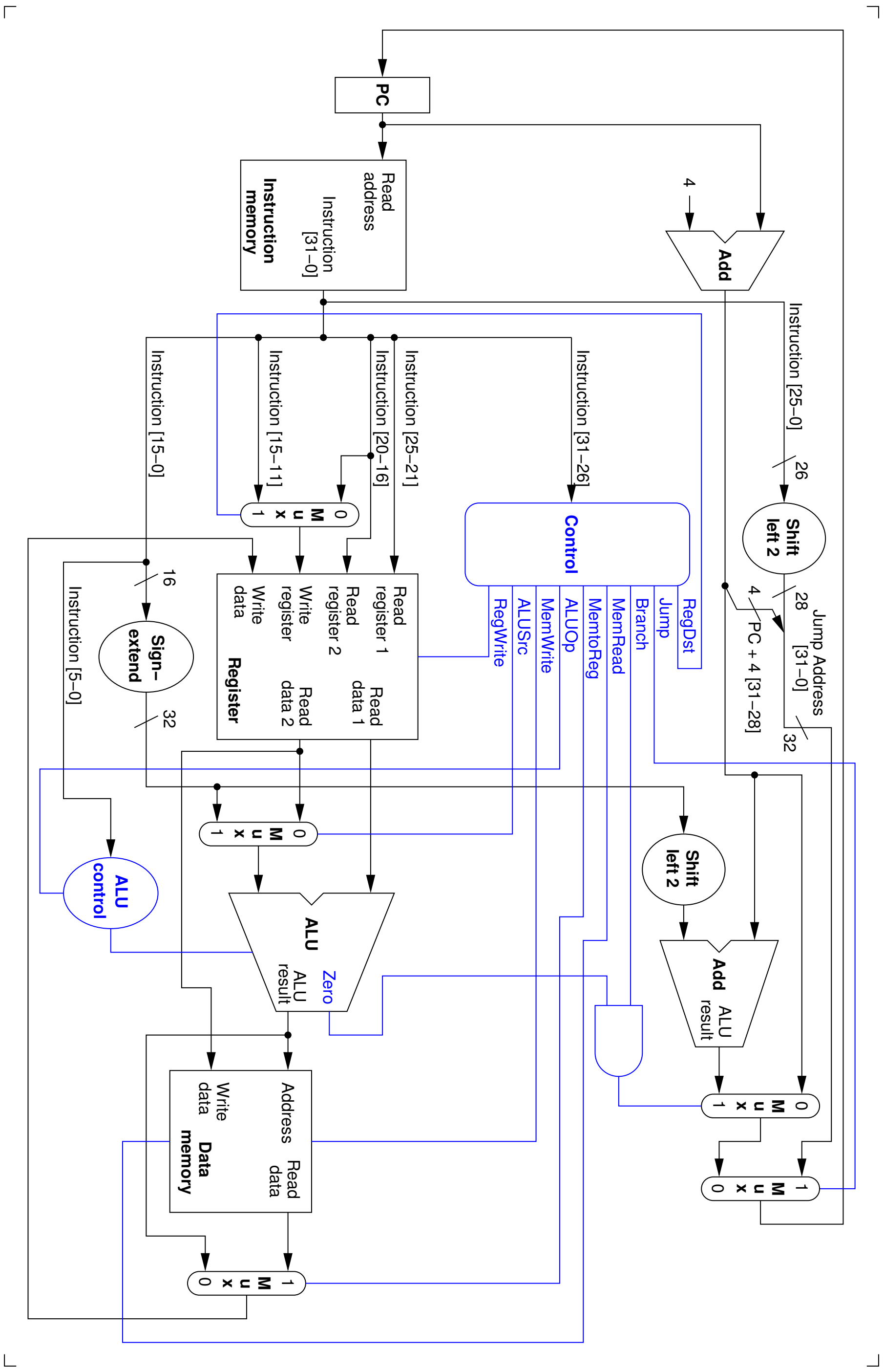
The behavior of the calculator for this project will be simpler than the project 1. If your calculator in project 1 works perfectly, you can skip this part. Otherwise, for this project, we will only test your calculator by simply pressing keypad in the following order:

1. enter the first operand (multiple digit number),
2. enter an operator (+, -, *, or /),
3. enter the second operand (multiple digit number),
4. press equal symbol (=) to check the result,
5. press clear, and go back to step 1.

So, if your calculator cannot support the above behavior, fix your program.

Part II: Hardware

For this project, you should start from the given starter file (`calculatorCPU.circ`). This file consists of the main circuit and various subcircuit. Your job is to build a circuit **almost** exactly the same is the MIPS CPU discussed in class as shown below:



Note that for this project we do not need the data memory. In `calculatorCPU.circ`, you will find a number of circuits. What do you have to do for this project is implement all circuits that marked `TO DO` and the main circuit. **Do not modify any other circuits**.

Main Circuit

The main circuit (given) has an LCD display and keypad is shown in Figure 1. Your job is to put your CPU into this main circuit and connect the following four lines to your CPU:

- `From $t8` (32 bits): This line should be connected to the output of the register `$t8`
- `From $t9` (32 bits): This line should be connected to the output of the register `$t9`
- `Write Data to $t9` (32 bits): This line should be connected to the write data of the register `$t9` is a special way which will be explained later.
- `Write Signal for $t9` (1 bit): This line should be connected to the write signal for the register `$t9` in a special way which will be explained later.

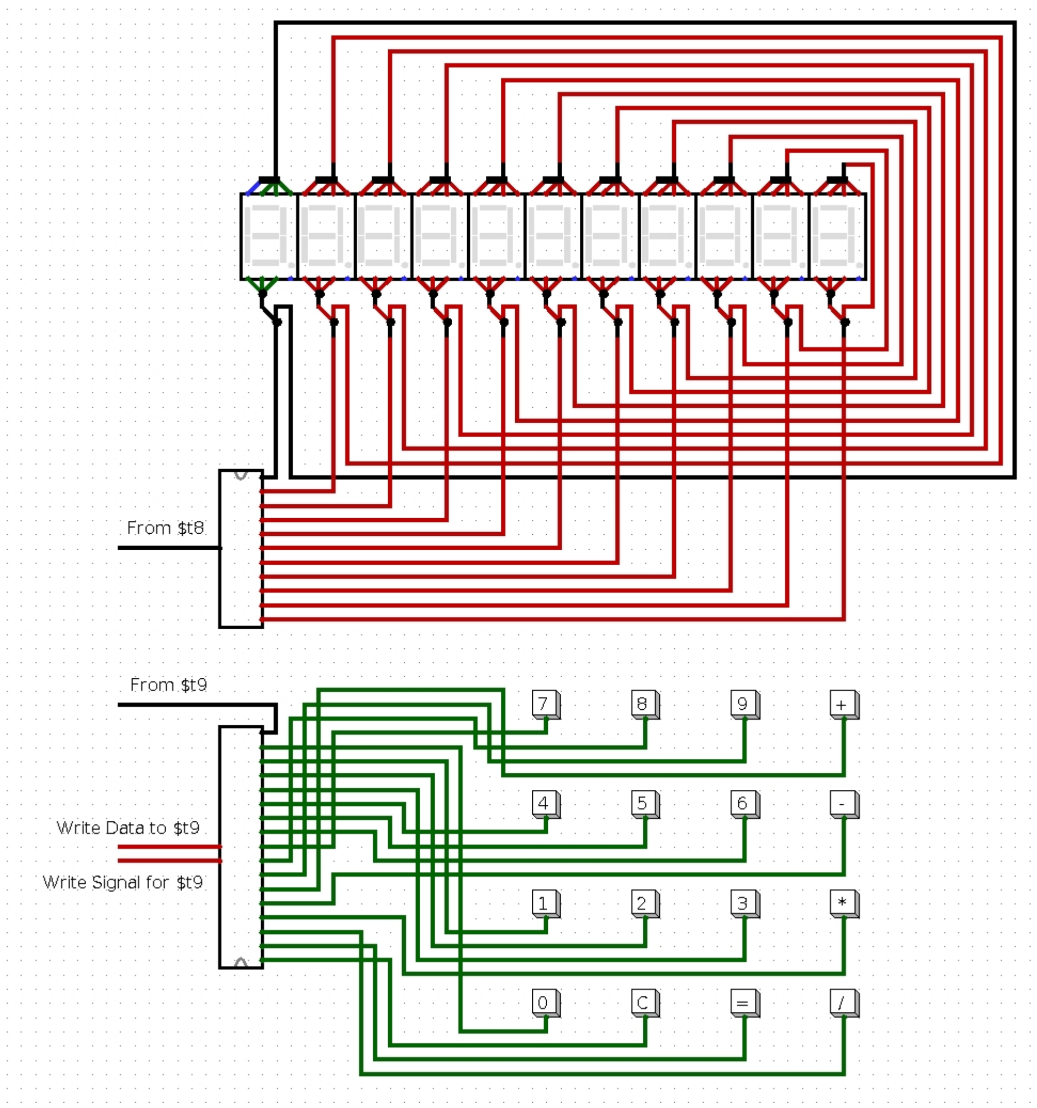
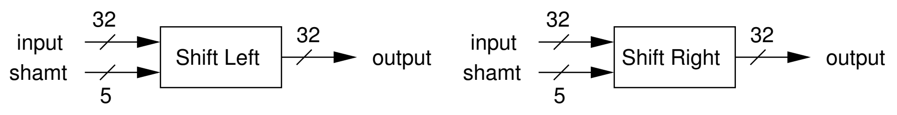


Figure 1: LCD Dispaly and Keypad on the Main Circuit

Shift Left and Shift Right Circuits

Shift left and shift right circuits are circuits that consist of two inputs and one output as shown below:



The inputs to the circuits are a 32-bit value and a 5-bit shamt (shift amount). Shift left and shift right circuits can be achieve using multiplication and division. For example, $a << b$ is equivalent to $a \times 2^b$. Similarly, $a >> b$ is equivalent to $a/2^b$. To generate 2^b in hardware, simply use a 5-bit decoder which consists of 32 1-bit output. There will be exactly one output that is 1 and the rest will be 0s.

Sign Extension

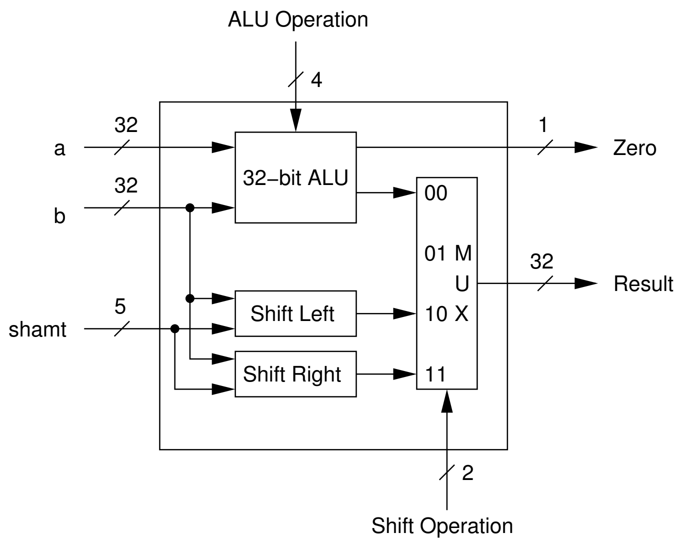
A sign extension circuit is a circuit that extend a 16-bit value into a 32-bit value without changing the value. As discussed in class, this can be done by simply copy the most significant bit value of the 16-bit value and use it for bit 16 to bit 31 of the 32-bit value. Note that this circuit is just a wiring circuit. No gate is needed.

Shift Left Two

A shift left two circuit is a circuit that take a 32-bit bit value and shift it to the left by 2. This can be achieved by simple wiring or multiply by 4.

32-bit ALU

This is a 32-bit ALU that support all instructions except `j`. Note that `sll` and `srl` must be supported in this 32-bit ALU as well but we do not cover this in class. One easy way to support shift instruction is to add another 32-bit 4-input multiplexer as shown below:



For this circuit, you do not have to built the 32-bit ALU from scratch. Simply use 32-bit gates and multiplexers. To control this ALU, we need to give two appropriate values to `ALU Operation`

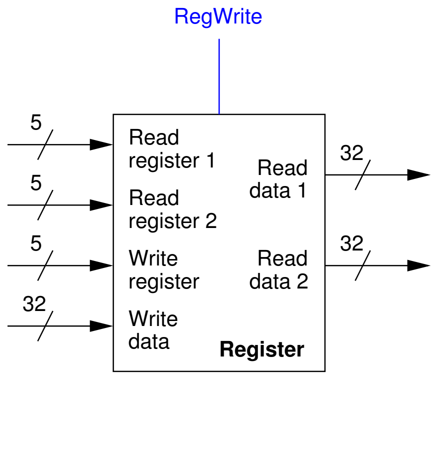
Instruction	op	funct	ALU Control	Shift Control
add	000000	100000	0010	00
addi	001000	N/A	0010	00
and	000000	100100	0000	00
andi	001100	N/A	0000	00
sub	000000	100010	0110	00
or	000000	100101	0001	00
ori	001101	N/A	0001	00
nor	000000	100111	1100	00
slt	000000	101010	0111	00
slti	001010	N/A	0111	00
sll	000000	000000	N/A	10
srl	000000	000010	N/A	11
beq	000100	N/A	0110	N/A
bne	000101	N/A	0110	N/A
j	000010	N/A	N/A	N/A

Table 1: Instructions and Control Values

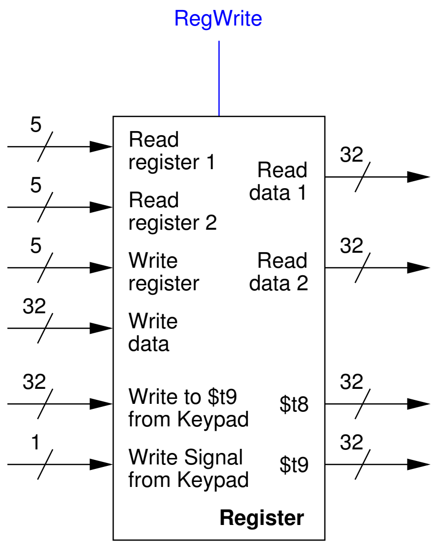
(4-bit) and `Shift Operation` (2 bit). Table 1 shows input values of `ALU Operation` and `Shift Operation` for every instruction.

Register File

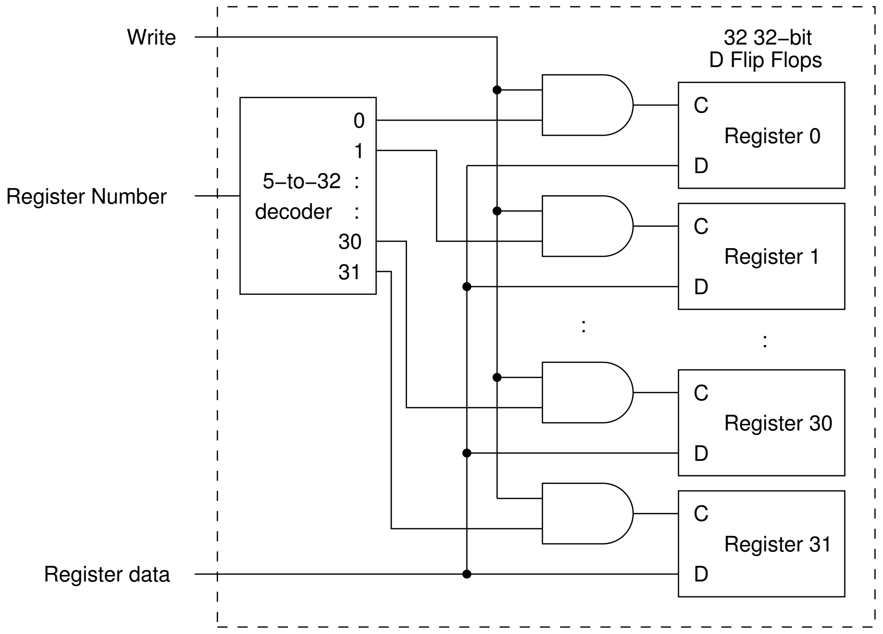
The register file can be build in the similar fashion as discussed in class as shown below:

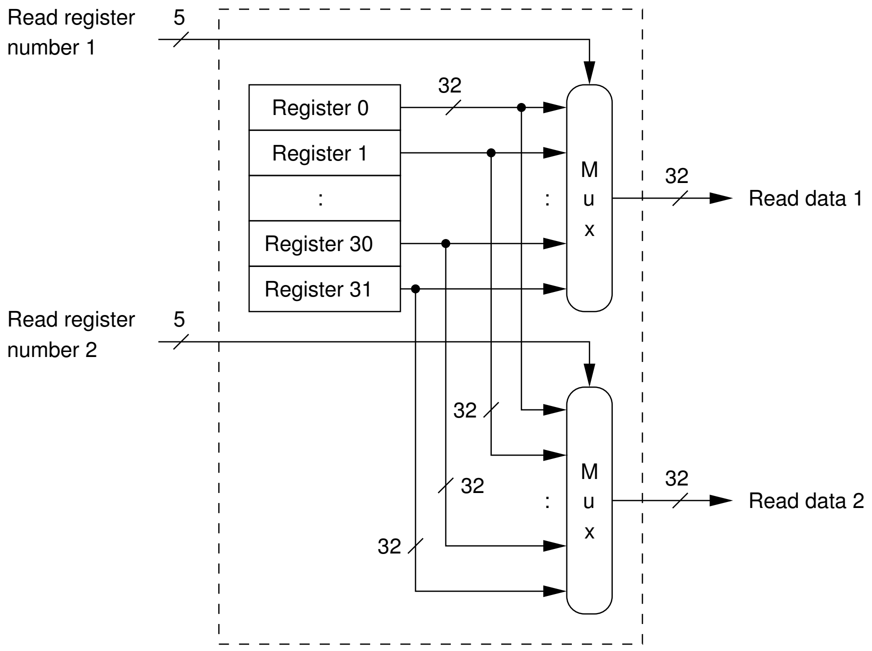


Register File



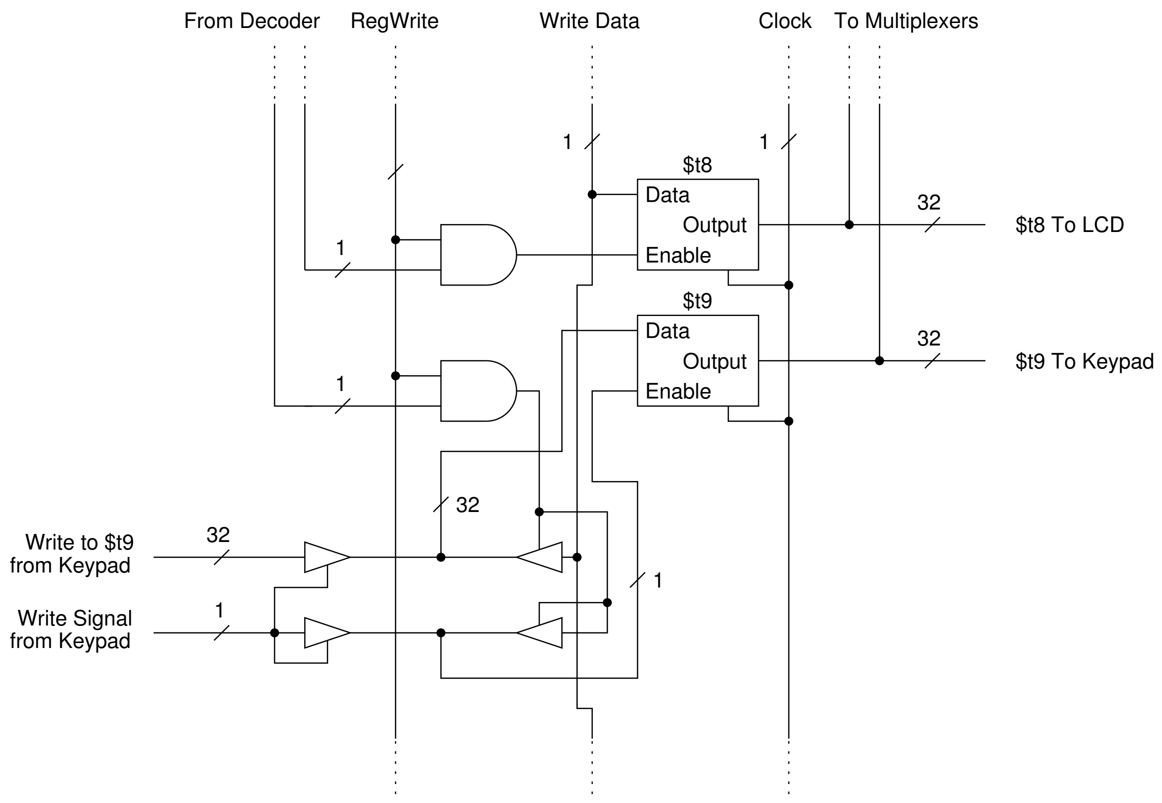
Register File for This Project





But instead of using D-Flip Flops, simply use the register provided by logisim. Note that these registers need a clock and you can set the "Trigger" of registers at "Rising Edge" or "Falling Edge". This setting indicate when the new written value should go out to the output when the clock is ticked. Note that every register should have the same setting.

Note that this register file will be a little bit different than the one discussed in class. Since LCD is control by simply set the value of the register `$t8`, we need a direct output from the register `$t8`. Similarly, the keypad need to check whether the register `$t9` is 0. So, we also need a direct output from the register `$t9`. Keypad itself also needs to write directly to register `$t9`. This will cause a conflict if the CPU and the keypad want to write to `$t9` at the same time. Thus, the faction of your register file focus only on `$t8` and `$t9` should look like the following:



The triangle symbol and look almost like the not gate is called "controlled buffer" which can prevent two signals from conflicting with each other.

Instruction Memory

For instruction memory, simply use a ROM with 16-bit addressing and 32-bit data width. Recall that the MIPS uses byte addressing and every instruction is 32-bit wide. Thus, the least significant two bits always be 0s. To make this ROM compatible with MIPS architecture, simply use bit 2 to bit 17 from your program counter to feed into the 16-bit address input of the ROM. In doing so, every time your program counter is increased by 4, the ROM will send out the next instruction.

In logisim, you are allowed to load data into your ROM. The data will be the machine code that you modified from Part I. But first, you have to dump the machine code from Mars and load it into ROM in logisim. To do so, perform the following steps:

1. In Mars, open your calculator program
2. Assemble it
3. In the file menu, select "Dump Memory..." a new window will pop-up.
4. In the "Memory Segment", select ".text (...)"
5. In the "Dump Format", select "Hexadecimal Text"
6. Then click "Dump to File..."
7. Choose which ever file name you like (e.g., program.txt)

8. Open that file using a text editor. You will see a list of 32-bit instruction (in hex) as shown below. Note that your program may look different.

```
2017000a
0000c020
0000c820
00008020
:
```

9. Insert the string ”v2.0 raw” in the first line and your program should look somewhat like the following:

```
v2.0 raw
2017000a
0000c020
0000c820
00008020
:
```

10. Save your file
11. In logisim, use the arrow tool, right click on your ROM, and select ”Load Image..”
12. Select your file and click ”OK”

You just have to do this once unless you modify your program in Mars.

Control Unit

As we discussed in class, the main task of the control unit is to decode a machine instruction and send various of control signal to components inside CPU to perform a specific task. For this project, you have to design your own control unit from scratch. For this project, it is up to you whether your CPU will have a separate ALU control or not. Ideally, you can also have the control unit that receive both 6-bit opcode as well as 6-bit function field. Note that not all instruction are covered in class such as `addi`, `andi`, `ori`, `sll`, `srl`, and `bne`. These instructions can be supported without major modification to the CPU hardware.

Submission

Your project is due on Sunday April 19, 2015 at 11:59pm. Late submissions will not be accepted. You should submit the file **three** files as follows:

1. Your program from Mars (e.g., `calculator.asm`)
2. Your memory dump from Mars (e.g., `calculatorHex.txt`)
3. `calculatorCPU.circ`.

Zip all three files into a single file and submit it onto CourseWeb.