Lab 05 - Single Cycle Datapath

Introduction

In this lab, you will be building a single cycle version of the MIPS datapath. The datapath is composed of many components interconnected. They include an ALU, Registers, Memory, and most importantly the Program Counter (PC). The program counter is the only clocked component within this design and specifies the memory address of the current instruction. Every cycle the PC will be moved to the location of the next instruction. **The MIPS architecture is BYTE ADDRESSABLE**. Remember this when handling the PC, and the memory (which is WORD ADDRESSABLE).

Pre-lab

You will need to submit several tests for your prelab. These must be submitted on ilearn prior to coming to the lab (check online for due dates). These tests will each consist of an `init.coe`, a corresponding `.asm` file and a `processor_tb.v` file for each.your test file (`init.coe` a corresponding `.asm` file and a `processor_tb.v`). The tests you should write are:

<table>
<tr>
<td>

```
individualInstructions.asm
lw $v0, X($zero)
lw $v1, Y($zero)
add $a0, $v0, $v1
addi $a0, $v0, Z
...
```

Where X and Y are the address of data you have placed in your .coe file and Z is the immediate you are using. The numbers corresponding to the register numbers are in a table at the end of the document.

</td>
<td>

```
program.asm
# An entire program where each instruction
# has an effect on the final outcome.
# The result will be verified through
# inspection of the write_reg_data value during
# execution of the last instruction (see below)
```

</td>
</tr>
<tr>
<td>

```
individualInstructions.coe
100011 00000 00010 XXXXXXXXXXXXXXXX
100011 00000 00011 YYYYYYYYYYYYYYYY
000000 00010 00011 00100 00000 000000
001000 00010 00100 ZZZZZZZZZZZZZZZZ
```

The spaces are added for clarity and should be removed before running. The X's, Y's, and Z's should also be replaced with there corresponding values.

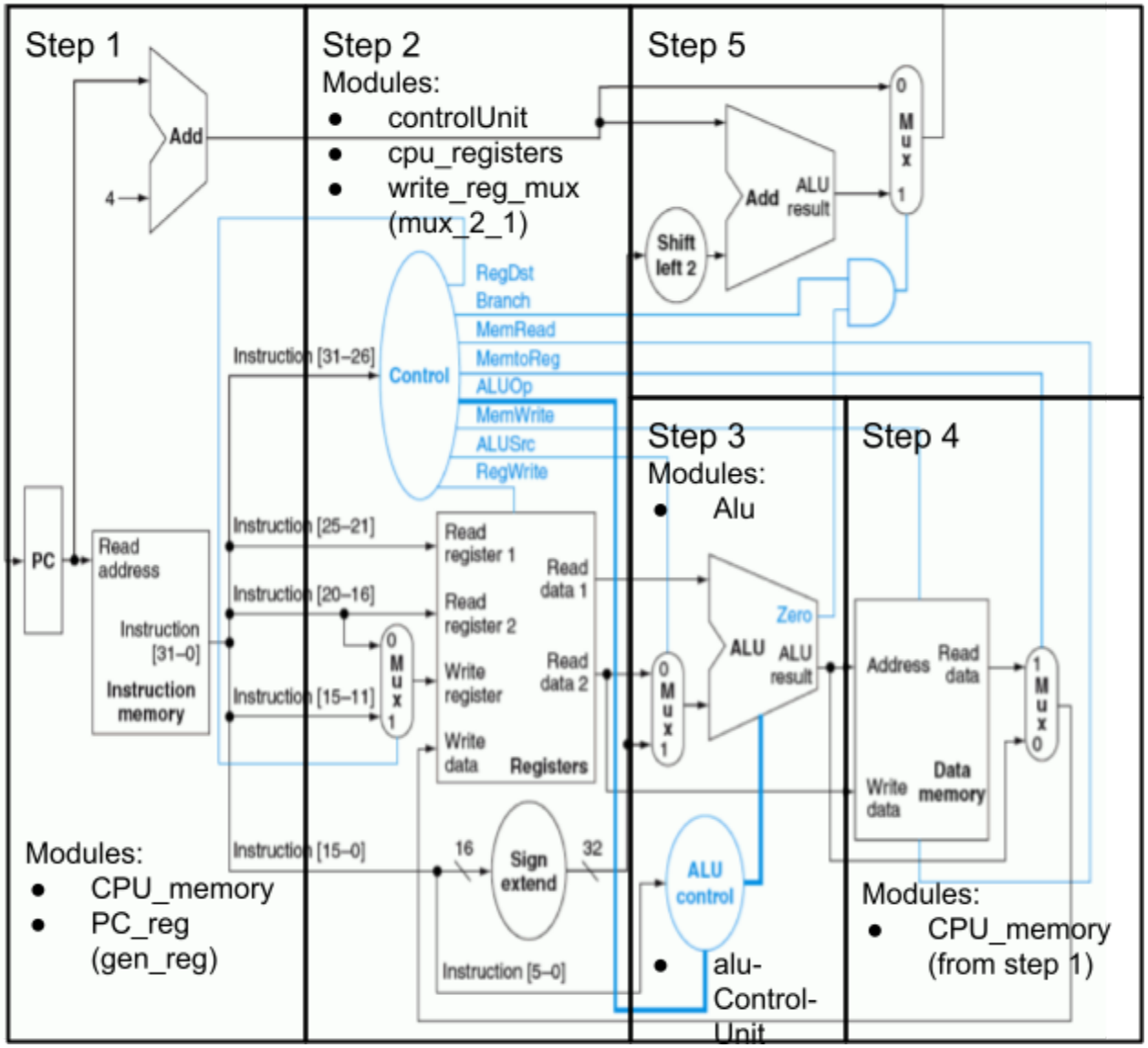
</td>
<td>

`program.coe`
Machine language translation of the program from above.

</td>
</tr>
<tr>
<td colspan="2">See testbench example files below.</td>
</tr>
</table>

Your `init.coe` should demonstrate that you have read through the lab specifications and understand the goal of this lab. You do not need to begin designing yet, but this testbench will be helpful during the lab while you are designing.

The component connections (shown below) are outlined in the CS161 notes.



Recommended Iterative Development

Step 1:

Count up the PC and check that each instruction is correct (`instr_opcode`, `reg1_addr`, `reg2_addr`, `write_reg_addr`)
Modules:

- Instruction memory* (`cpumemory.v`)
- PC Register (`gen_register.v`)
- PC adder (optional)

Step 2:

Verify that the control signals are still correct on the `control_unit` from the previous lab. Verify that you can read values from the registers (at this point they will most likely be all 0's). You should now be able to get defined values for all debug signals (including `reg1_data`, `reg2_data`, `write_reg_data`). The data values will most likely be all 0's since no data manipulation is being done yet.

Step 3:

Add the `alu_control` from the previous lab as well as the ALU. There won't be any change in the debug signals yet, but verify that the ALU output is defined.

Step 4:

Connect the Data Memory* to the ALU and the CPU registers. At this point your data signals (reg1_data, reg2_data, write_reg_data) should be correct for all non-branch instructions.
* Data and Instruction memory are unified for our processor. The CPU_memory module is a dual-port memory unit that allows simultaneous reads of instructions as well as a read/write of data through separate ports. In step 1 you will only have the instruction ports connected, now you'll connect the rest of them.

Step 5:

Add the modules for the branching hardware. This may involve breaking some connections from step 1 to insert the proper hardware for branching. You should not have any undefined signal before this step, so it shouldn't be too difficult to trace down any introduced high Z or undefined signals.

Deliverables

For the turn-in of this lab, you should have a working **single-cycle datapath**. The *true* inputs to the top module (`processor.v`) are only a `clk`, and `rst` signals, although you will need to have the debug signals correctly connected as well. The datapath should be programmed by a "`.coe`" file that holds MIPS assembly instructions. This file is a paramter to the top module, but defaults to "`init.coe`".

For this lab, you are not required to build all the datapath components (in black in the image above) but you are required to connect them together in the datapath template provided (`processor.v`). You will be connecting this datapath and your `alu_control.v` and `control_unit.v` from Lab 03 in the top-level module (`processor.v`). All of the files can be found in the zip file for this lab. If you need more functionality you will have to build the components yourself. To use the given components you only need to copy the given Verilog files into your project. By default, the architecture's memory loads data from an "`init.coe`" file. The programming occurs when the `rst` signal is held high. A sample `init.coe` file is given but **does not fully test the datapath**. You will have to extend it. For convenience, the assembly for the `init.coe` file can be found in the zip file. The last instruction of the program is a `lw` to load the final value from memory into register `$t0` and is used to verify correct functionality in the test bench (see below). If you add a similar line to your own program it will make testing easier.

Submission:

Each student **must** turn in one tar file to Gradescope. Your work can be (and should be) identical to the other members of your group. The contents of which should be:

- A README file with the group members names, and any incomplete or incorrect functionality
- All Verilog file(s) used in this lab (implementation and test benches).

- All `.coe` and `.asm` files used in this lab

If your file does not synthesize or simulate properly, you will receive a 0 on the lab.

Output

The expected waveform for the `init.coe` file is shown below. Notice the PC increments by 4, the code runs a loop so the waveform can continue indefinitely and if you add the memory buffer to the waveform you can scroll to see that the store works properly.

Generated using the following testbench.

`processor_tb.v`

```
`timescale 1ns / 1ps

module processor_tb;
// Inputs
reg clk;
reg rst;
// Outputs
wire [31:0] prog_count;
wire [5:0] instr_opcode;
wire [4:0] reg1_addr;
wire [31:0] reg1_data;
wire [4:0] reg2_addr;
wire [31:0] reg2_data;
wire [4:0] write_reg_addr;
wire [31:0] write_reg_data;
```

```
processor uut (
    .clk(clk),
    .rst(rst),
    .prog_count(prog_count),
    .instr_opcode(instr_opcode),
    .reg1_addr(reg1_addr),
    .reg1_data(reg1_data),
    .reg2_addr(reg2_addr),
    .reg2_data(reg2_data) ,
    .write_reg_addr(write_reg_addr),
    .write_reg_data(write_reg_data)
);
```

```
initial begin
    clk = 0; rst = 1; #50;
    clk = 1; rst = 1; #50;
    clk = 0; rst = 0;
    forever begin
        #5 clk = ~clk;
    end
end
```

The below will print whether your entire program is working correctly.

```
reg[31:0] result;
reg[31:0] expected = 172;
integer last_instruction = 44;
integer passedTests = 0;
integer totalTests = 0;
initial begin
    wait (prog_count == last_instruction)
    result = write_reg_addr;
    $write("Test Case 1: (init.coe)...");
    #1;
    totalTests = totalTests + 1;
    if (result == expected) begin
        passedTests = passedTests + 1;
        $display("passed");
    end else begin
        $display("failed. Expected %0d but got %0d.",expected,result);
    end
    $display("----------------------------------------------------------------------");
    $display("Testing complete\nPassed %0d / %0d tests.",passedTests,totalTests);
    $display("----------------------------------------------------------------------");
end
endmodule
```

The below will let you test each instruction independently. You can combine the two for a more robust test.

```
integer passedTests = 0;
integer totalTests = 0;
initial begin
    @(negedge rst); // Wait for reset
    @(posedge clk); // Skip instructions you aren't testing (lw in this case)
    // The second lw is skipped during the next line, which moves on to the next instruction
    @(posedge clk); #1; // Put this before each test to move on to the next instruction
    totalTests = totalTests + 1;
```

```
    $write("Test Case %0d: (add $a0, $v0, $v1)...",totalTests);
    $write("Test Case %0d: (addi $a0, $v0, Z)...",totalTests);
    if (write_reg_addr === `b00100 && write_reg_data === <expected>) begin
        passedTests = passedTests + 1;
        $display("passed");
    end else begin
        $display("failed.");
    end
    $display("-------------------------------------------------------------------");
    $display("Testing complete\nPassed %0d / %0d tests.",passedTests,totalTests);
    $display("-------------------------------------------------------------------");
end
endmodule
```

Reg #	Alt. Name	Description
0	zero	the value 0
1	$at	(**a**ssembler **t**emporary) reserved by the assembler
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	$t0 - $t7	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	$s0 - $s7	(**s**aved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	$t8 - $t9	(**t**emporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to $t0 - $t7 above. Not preserved across procedure calls.
26-27	$k0 - $k1	reserved for use by the interrupt/trap handler
28	$gp	**g**lobal **p**ointer. Points to the middle of the 64K block of memory in the static data segment.
29	$sp	**s**tack **p**ointer. Points to last location on the stack.
30	$s8/$fp	**s**aved value / **f**rame **p**ointer. Preserved across procedure calls
31	$ra	**r**eturn **a**ddress

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html

Converting `.asm` to `.coe`
https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats