Memory component

Some notes before we start:

- **Note 1:** *We make a RISK-V CPU in which memory architecture is 32 bit. In lecture time we covered a 64 bits RISK-V CPU. Thus, the length of registers are 32 bits as well as memory addressing.*
- **Note 2:** *You will need your **cpu.v** file from Lab L*
- You will need to use the sequential component library for this lab. Click **here** to download it.

Perform the following groups of tasks:

Register

In a previous lab we built and used memory-less, combinational components. In this lab we will use sequential components from a ready-made library. The first is called a **register** and has this diagram:

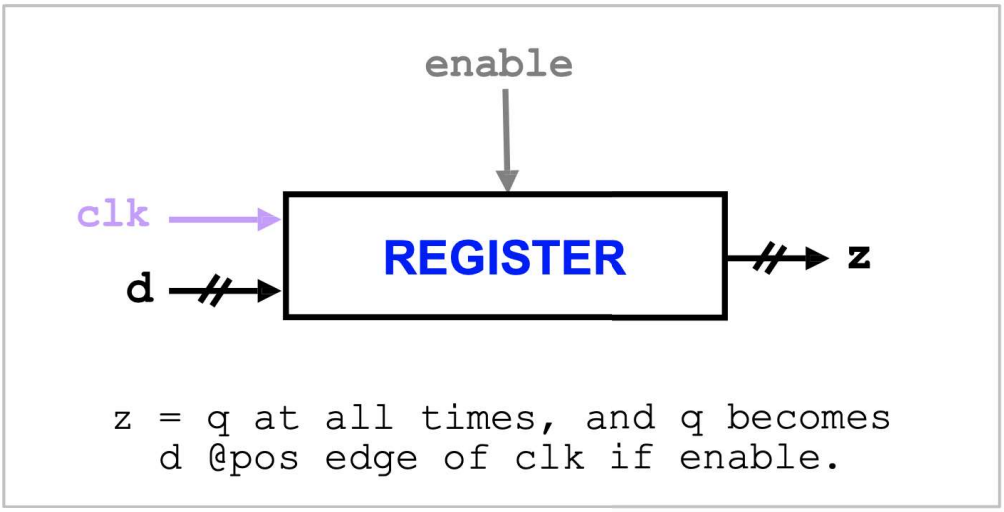

Figure 1: Register

As all sequential components, a **register** has an internal state, denoted by **q**, and it is made up of n bits, where n is a parameter set upon instantiation. The output **z** is equal to **q** at all times; i.e. you can always read the content of a register. In order to write (i.e. store) a value in the register, supply the value through d, set **enable** to 1, and then wait until the **clk** input rises from 0 to 1.

Note that **enable** is so named because if it is **0** then nothing can be written to the register; i.e. its state (and thus its output) remains unchanged, so it is effectively disabled. Note also that even if enable is set to 1 the state won't change until the next leading edge of **clk**. Think of **clk** as a periodic signal that oscillates between 0 and 1 at fixed intervals, a **clock**. The signal's period (measured in sec) is referred to as the clock cycle while its frequency (measured in Hz) is known as the clock **rate**.

If you need to point the compiler to the sequential component library use `-y<path_to_library>`.

Create the program **LabM1.v** that instantiates and tests a `32-bit` **register**. In order to test the **enable** control input, we will supply it as a command-line argument. As to the clock and data inputs (**clk** and **d**), let us start by hard coding several test values for them so we can get a feel for the circuit. Here is our first attempt:

```
module labM; reg [31:0] d;
    reg clk, enable, flag;
    wire [31:0] z;
    register #(32) mine(z, d, clk, enable);

    initial
    begin
        flag = $value$plusargs("enable=%b", enable);

        d = 15; clk = 0; #1;
        $display("clk=%b d=%d, z=%d", clk, d, z);

        d = 20; clk = 1; #1;
        $display("clk=%b d=%d, z=%d", clk, d, z);

        d = 25; clk = 0; #1;
        $display("clk=%b d=%d, z=%d", clk, d, z);

        d = 30; clk = 1; #1;
        $display("clk=%b d=%d, z=%d", clk, d, z);

        $finish;
    end
endmodule
```

Compile and run **LabM1** supplying 0 and then 1 for enable:

```
iverilog -y./hrLib/ LabM1.v
vvp a.out +enable=1
```

Does the register behave as expected in both cases?

We seek to generalize the above tester. We will generate random values for d every two units of time:

```
repeat (20) begin
    #2 d = $random;
end
```

As to the clock, let us make its cycle 5 units:

```
always begin
    #5 clk = ~clk;
end
```

Save **LabM1** as **LabM2** and modify it so it generates **d** and **clk** as above and keep enable as a command-line argument. Remember to initialize the clock to 0.

9. In order to capture the values of the signals, it is preferable to use monitor rather than display in order to capture changes as they occur. Add this block:

```
always initial
    $monitor("%5d:clk=%b,d=%d,z=%d,expect=%d", $time,clk,d,z, e);
```

The **e** signal is the expected value of the register's output as produced by the oracle. You will need to add an oracle that computes **e** (perhaps in the always block).

Run **LabM2** with enable set to `1`. **Does the register behave as expected?**

Register file

The next sequential component is the **register file**. It is made up of 32 registers (x0 thru x31) each of which is 32-bit. It allows us to read any two registers in it in parallel and to write to any one register. The rf component has this block diagram:

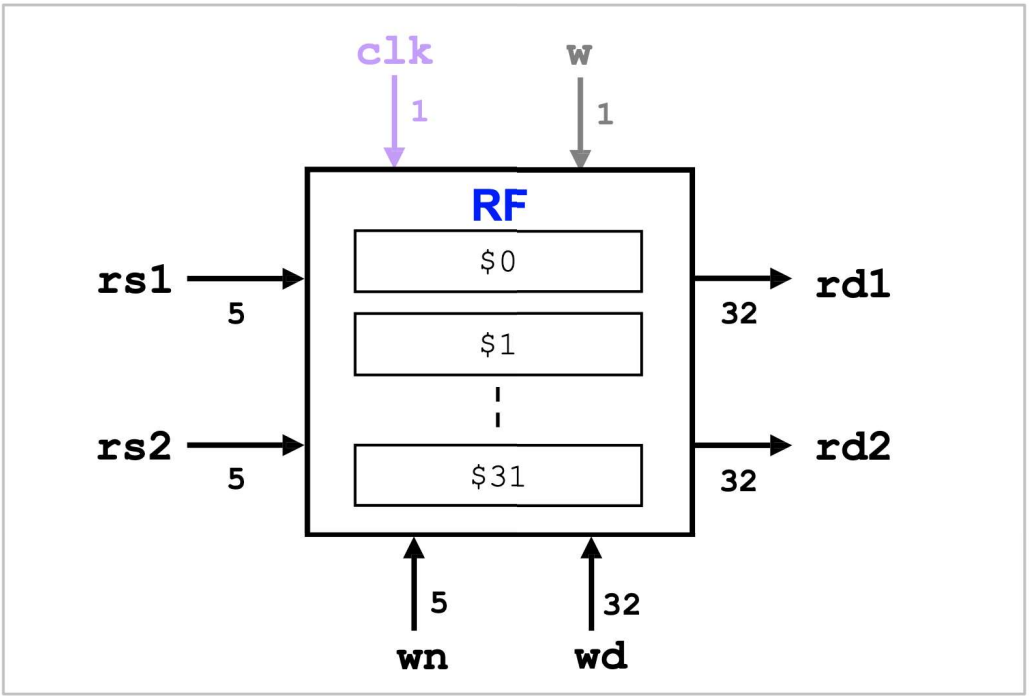

Figure 2: Register file

In order to read the contents of two registers, we supply their register numbers thru **rs1** and **rs2** (5 bits each). After some delay, the corresponding contents of these two registers will become stable on **rd1** and **rd2**. In order to store a `32-bit` value in some register, we supply it on **wd** and supply the `5-bit` register number on **wn**. The value will be written on the positive edge of **clk** if and only if **w=1**. Hence, **w** is in fact an enabler. Note that register `$0` is read-only and its value is `0`.

You can instantiate a register file using the statement:

```
rf myRF(rd1, rd2, rs1, rs2, wn, wd, clk, w);
```

As in the register, writing involves setting **w** and waiting for the rising edge of **clk**.

Mem

The last sequential component is **mem** and it simulates memory:

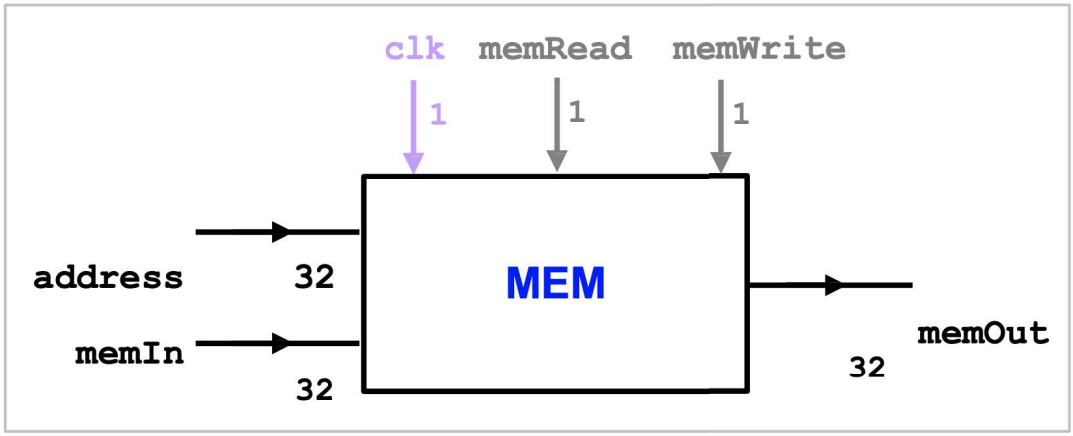

Figure 3: Register file

This sequential component is made up of many `32-bit` words. In order to read a word, set its address on address and set **memRead**. After some delay, the word's content will become stable on **memOut**. To write, set the data to be written on **memIn**, set the destination address on address, and set **memWrite**. The data will be written on the destination at the next positive edge of **clk**.

Note that mem stores `32-bit` words, not bytes, and hence, it assumes **word** addresses (i.e. divisible by 4). If the supplied address is not a word address, **mem** will display the message *"unaligned address"* and ignore the read or write request

Create the program **LabM4.v** that instantiates **mem** as follows:

```
mem data(memOut, address, memIn, clk, read, write);
```

To get a feel for this component, let's store some numbers in it. Write the value `32'h12345678` at address 16 and the value `32'h89abcdef` at address `24`. Note that both are word addresses. Read and display the contents of three words beginning at 16.

```
write = 0; read = 1; address = 16;
repeat (3)
begin
    #1 $display("Address %d contains %h", address, memOut); address = address + 4;
end
```

Compile and run **LabM4.v**. Ignoring the error message about *"ram.dat"*. **Does your program behave as expected?**

The **mem** component has a handy feature that allows us to initialize memory from a text file named "**ram.dat**".

When the component is first instantiated, it looks (in the current working directory) for that file, and if present, it reads its content to initialize the memory words. Each record in the file holds an address, content pair:

```
@a c // optional comment
```

The address **a** must be prefixed with @ and is followed by the content **c**. Both values must be in hex and are separated by whitespace. The record may end with an in-line comment using the // separator.

Let us create such a file so it represents the memory map of the following program:

```
ARRAY:
    DW 1, 3, 5, 7, 9, 11, 0
SUM-ORRED:
    DW 0, 0
START:
    add t5, x0, x0        # index
    add s0, x0, x0        # sum
    add a0, x0, x0        # or reduction
LOOP:
    lw t0, 0(t5)          # loop: t0 = array[t5]
    beq t0, x0, DONE      # if (t0 == 0) done
    add s0, s0, t0
    or a0, a0, t0
    addi t5, t5, 4        # t5++
    jal x0, LOOP
DONE:
    sw s0, 0x20(x0)       # done: save s0
    sw a0, 0x24(x0)       # save a0
```

It is important that we understand all aspects of this program because we will use it as a test bed for all our circuits. Given a null-terminated array of words at address `0x18`, the program computes the sum and the OR of all its elements and stores them at addresses `0x20` and 0x24, respectively. In order to allocate room for these two words, your —ram.dat— file must contains:

```
@20 00000000 // the sum
@24 00000000 // the or reduction
```

We also need to store the array itself beginning at `0x30`. Add this to *ram.dat*:

```
@00 00000001 // array[0]
@04 00000003 // array[1]
@08 00000005 // array[2]
@0C 00000007 // array[3]
@10 00000009 // array[4]
@14 0000000B // array[5]
@18 00000000 // null terminator
```

Note that the array elements add up to `36` (decimal) and their OR reduction is `15` (decimal). The correct execution of the code should store these at `0x20` and `0x24`.

In order to store the program itself (in machine language) load the code in the RISK- V simulator and capture the machine encoding of each statement. We will load the entry point main at address `0x00`. Here are the first few lines to be appended to your `ram.dat`:

```
@00 00000001 // array[0]
@04 00000003 // array[1]
@08 00000005 // array[2]
@0C 00000007 // array[3]
@10 00000009 // array[4]
@14 0000000B // array[5]
@18 00000000 // null terminator @20 00000000 // the sum
@24 00000000 // the or reduction
@28 00000F33 // add x30, x0, x0         # index
@2C 00000433 // add x8, x0, x0          # sum
@30 00000533 // add x10, x0, x0         # or reduction
@34 000F2283 // lw  x5, 0(x30)          # loop: x5 = array[x30]
@38 00028563 // beq  x5, x0, DONE        # if (x5 == 0) done
...
```

Listing 1: The machine encoding is simply transferred from the RISK-V simulator.

Create **LabM5.v** to display the program in ram.dat along the following lines:

```
module labM;
    reg clk, read, write;
    reg [31:0] address, memIn;
    wire [31:0] memOut;

    mem data(memOut, address, memIn, clk, read, write);
    initial
    begin
        address = 16'h28; write = 0; read = 1; repeat (11)
        ...
endmodule
```

Complete **LabM5.v** and run it. It should output the program in machine language

Save **LabM5.v** as **LabM6.v** and modify it so that it displays memory content in a format that is instruction-aware. For example, if the instruction is an I-type, then output the contents of its opCode, two registers, and immediate, as four separate outputs. Here is the correct output of the sought program:

```
00  00  00   0   1e   33 // R-Type
00  00  00   0   08   33 // R-Type
00  00  00   0   0a   33 // R-Type
000 1e  2    05  03       // I-Type
00  00  05   0   0a   63 // SB-Type
00  05  08   0   08   33 // R-Type
00  05  0a   6   0a   33 // R-Type
004 1e  0    1e  13      // I-Type
ff7ff   00   6f           // UJ-Type
01  08  00   2   00   23 // S-Type
01  0a  00   2   04   23 // S-Type
```

You can use the part-select operator to easily extract sub-fields from the instruction word. For example, here is how R-type can be detected:

```
if (memOut[6:0] == 7'h33)
```

Similarly, you can detect the UJ-type as follows:

```
if (memOut[6:0] == 7'h6F)
```

For I-Type:

```
if (memOut[6:0] == 7'h3 || memOut[6:0] == 7'h13)
```

For S-Type:

```
if (memOut[6:0] == 7'h23))
```

And for SB-Type:

```
if (memOut[6:0] == 7'h63)
```

CPU

Fetch Instruction

As a first step toward building the datapath of the CPU, let us implement a circuit for instruction fetch from memory. Here is its block diagram:

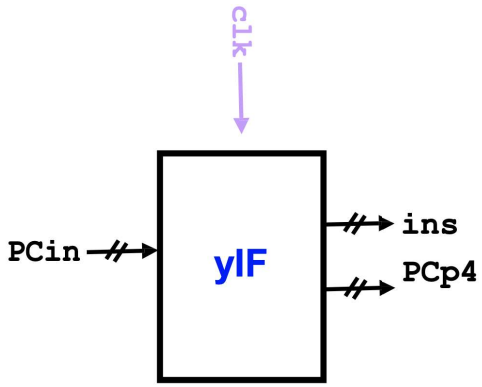

Figure 4: Given a memory address $PCin$, this circuit fetches from memory the instruction ins stored at that address and makes it available. The circuit also computes and outputs $PC_{p4} = PC_{in} + 4$ in anticipation of fetching the physically-following instruction.

Note that the circuit has a clock input **clk** that allows us to exercise precise control over its timing. Specifically, the circuit should initiate its fetch at the positive edge of **clk**. At any other time, the circuit should do nothing, and its outputs should remain unchanged even if **PCin** changed. In order to implement this circuit, we will use a register named **PC** (program counter) to store the memory address from which the instruction is to be fetched. The register is enabled at all times and has clk as its clock, as shown below:

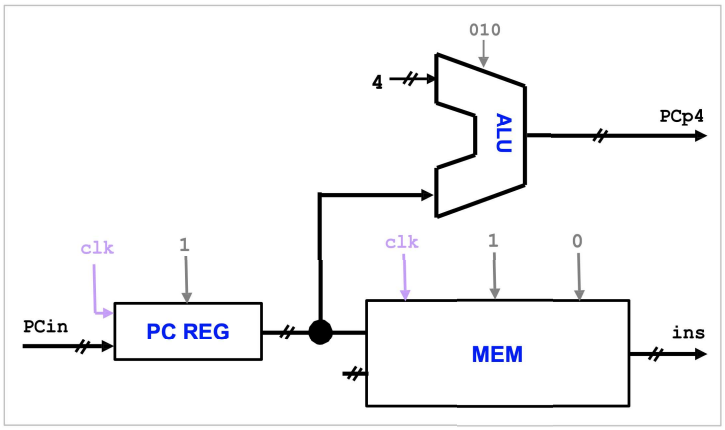

Figure 5: Fetch Instruction stage

We connect the output of the **PC** register to the address input port of our **mem** unit. Since we always read instructions from memory and never write, the **memIn** port of **mem** is left unconnected and the signals **memRead** and **memWrite** are set to 1 and 0 respectively. Note **clk** input of **mem** is irrelevant and can be left dangling (why?).

Copy **cpu.v** that was created in**Lab-L** to the directory of this lab and add the following component to it:

```
module yIF(ins, PCp4, PCin, clk);
output [31:0] ins, PCp4;
input [31:0] PCin;
input clk;

 // build and connect the circuit

endmodule
```

Complete the development of this module by instantiating the needed components and connecting them as shown in the diagram in Figure 5. Note that fixed signals, such as `010` for the ALU) can be hard-coded parameters in the instantiation.

Create **LabM7.v** to test your yIF component as follows:

```
module labM;
reg [31:0] PCin;
reg clk;
wire [31:0] ins, PCp4;

yIF myIF(ins, PCp4, PCin, clk);

initial
begin
//-------------------------------------Entry point
PCin = 16'h28;
//-------------------------------------Run program
repeat (11)
begin
//---------------------------------Fetch an ins
clk = 1; #1;
//---------------------------------Execute the ins
clk = 0; #1;
//---------------------------------View results
$display("instruction = %h", ins);
// Add a statement to prepare for the next instruction
end
$finish
end
endmodule
```

Compile and run **LabM7** as shown below. The output should be instructions in your **ram.dat** file.

```
iverilog LabM7.v cpu.v
vvp a.out
```

Instruction Decoder

Next, we build a component for instruction decoding. Here is its block diagram:

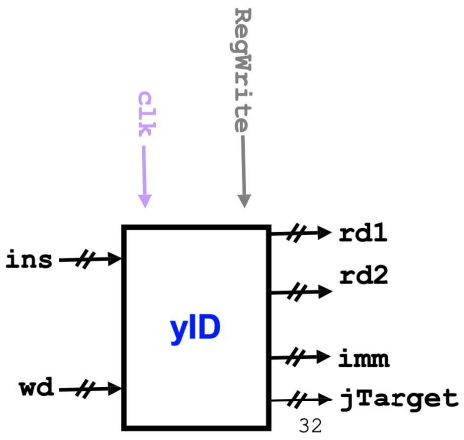

Figure 6: Instruction Decoder

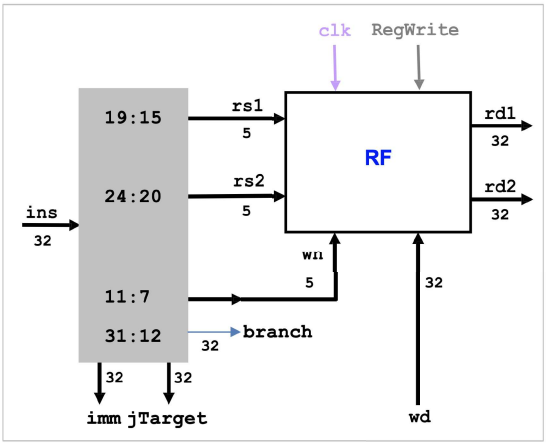

Figure 7: Instruction Decoder circuit diagram

In fact, this component plays two roles: *instruction decoding* and *data write-back*. In the decoding role, this component extracts the various fields of ins and looks up the needed registers. It produces in `d1` and `rd2` the contents of the two registers `rs` and `rt`; in **imm** the sign-extended immediate; and in **jTarget** the (20-bit) jump target. Not all these outputs will be meaningful for a given instruction; e.g. some instructions do not address two registers and some do not have immediates. The jump target is meaningful only if this is a jump-type instruction.

The write-back role of this component is used in a later stage of the execution. In it, the value of `wd` must be written to the register rd of the instruction. Recall that rd is determined by bits `11:7` of ins. In addition, writing is only to be done if the **RegWrite** signal is set and then only at the positive edge of `clk`.

The circuit diagram of **yID** is shown in Figure 7.

Add the following module to your **cpu.v** file.

```
    module yID(rd1, rd2, immOut, jTarget, branch, ins, wd, RegWrite, clk);
    output [31:0] rd1, rd2, immOut;
    output [31:0] jTarget;

    input [31:0] ins, wd;
    input RegWrite, clk;

    wire [19:0] zeros, ones; // For I-Type and SB-Type
    wire [11:0] zerosj, onesj; // For UJ-Type
    wire [31:0] imm, saveImm; // For S-Type

    rf myRF(rd1, rd2, ins[19:15], ins[24:20], ins[11:7], wd, clk, RegWrite);

    assign imm[11:0] = ins[31:20];
    assign zeros = 20'h00000;
    assign ones = 20'hFFFFF;
    yMux #(20) se(imm[31:12], zeros, ones, ins[31]);

    assign saveImm[11:5] = ins[31:25];
    assign saveImm[4:0] = ins[11:7];

    yMux #(20) saveImmSe(saveImm[31:12], zeros, ones, ins[31]);
    yMux #(32) immSelection(immOut, imm, saveImm, ins[5]);

    assign branch[11] = ins[31];
    assign branch[10] = ins[7];
    assign branch[9:4] = ins[30:25];
    assign branch[3:0] = ins[11:8];
    yMux #(20) bra(branch[31:12], zeros, ones, ins[31]);

    assign zerosj = 12'h000;
    assign onesj = 12'hFFF;
    assign jTarget[19] = ins[31];
    assign jTarget[18:11] = ins[19:12];
    assign jTarget[10] = ins[20];
    assign jTarget[9:0] = ins[30:21];
    yMux #(12) jum(jTarget[31:20], zerosj, onesj, jTarget[19]);

endmodule
```

Next, we build a component for executing the instruction. Here is its block diagram:

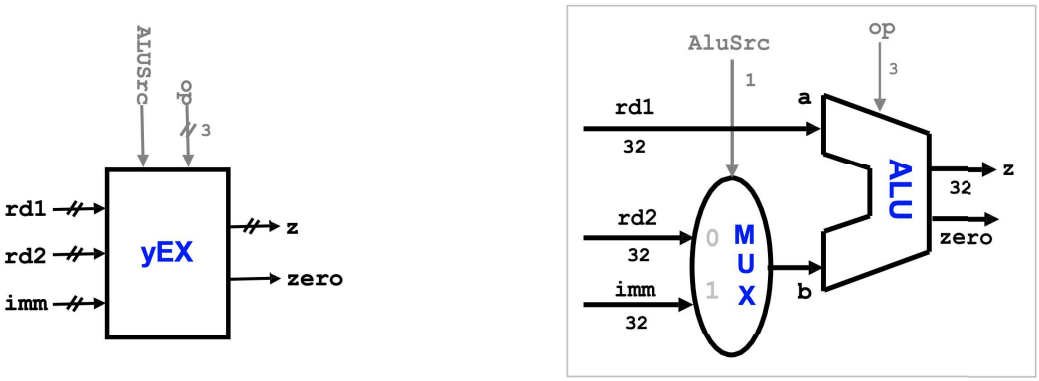

Figure 8: Instruction Execution block: This unit performs the operation specified via the op signal. We assume the same 3- bit operation codes used by the ALU. The operands are `rd1| and either rd2|or imm| depending on whether ALUSrc| is 0 or 1.`

The circuit follows directly from the definition of **yEX**. Add the following module to your **cpu.v** file:

```
module yEX(z, zero, rd1, rd2, imm, op, ALUSrc);
output [31:0] z;
output zero;
input [31:0] rd1, rd2, imm;
input [2:0] op;
input ALUSrc;

//Complete the development of yEX.
```

Save **LabM7.v** as **LabM8.v** and modify it to test **yID** and **yEX** as follows:

```
module labM;
reg [31:0] PCin;
reg RegWrite, clk, ALUSrc;
reg [2:0] op;

wire [31:0] wd, rd1, rd2, imm, ins, PCp4, z; wire [25:0] jTarget;
wire zero;

yIF myIF(ins, PCp4, PCin, clk);
yID myID(rd1, rd2, imm, jTarget, branch, ins, wd, RegWrite, clk);
yEX myEx(z, zero, rd1, rd2, imm, op, ALUSrc);

assign wd = z;

initial
    begin
        //-----------------------------------Entry point
        PCin = 16'h28;
        //-----------------------------------Run program
        repeat (11)
        begin
            //---------------------------------Fetch an ins
            clk = 1; #1;
            //---------------------------------Set control signals
            RegWrite = 0;
            ALUSrc = 1;
            op = 3'b010;
            // Add statements to adjust the above defaults
            //---------------------------------Execute the
            ins clk = 0; #1;
            //---------------------------------View results
            // display the following singals ins, rd1, rd2, imm, jTarget, z, zero
            ...
            //---------------------------------Prepare for the next ins
            PCin = PCp4;
        end
    $finish;
end
endmodule
```

Example: Notice that we connected the **yEX** output z back to the **wd** input of **yID**. This allows us to test the write-back functionality.

Finally, you can complete **LabM8** by adding statements to detect the instruction being executed and accordingly change the default control signals (we are only concerned with the ins- tructions in our **ram.dat**). Here is an example of what needs to be done:

Compile and run **LabM8** as follows:

```
iverilog LabM8.v cpu.v
vvp a.out
```

Note that our datapath does not have data memory and has no support for branches of jumps. **We will continue our CPU development in the next lab...**