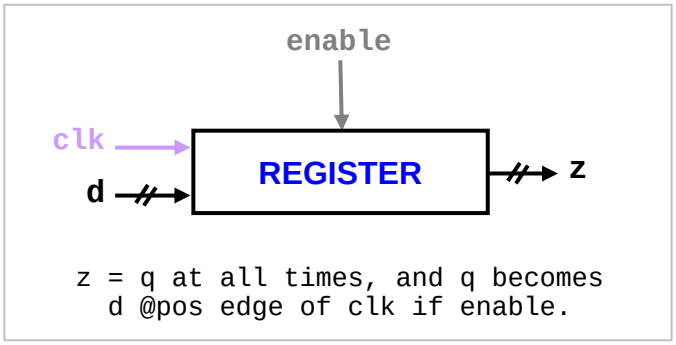
# LAB M
# Building the CPU

Perform the following groups of tasks:

## LabM1.v

1. 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:




   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.

2. 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**.

3. Our compiler is configured to locate our sequential component library. Hence, you can compile and run your programs as if these components are built in.

4. 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
```

Examine the code as you type it in and predict the program's output.

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

   ```
   vvp a.out +enable=1
   ```

   Does the register behave as expected in both cases?

## LabM2.v

6. 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
   ```

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

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

8. 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:

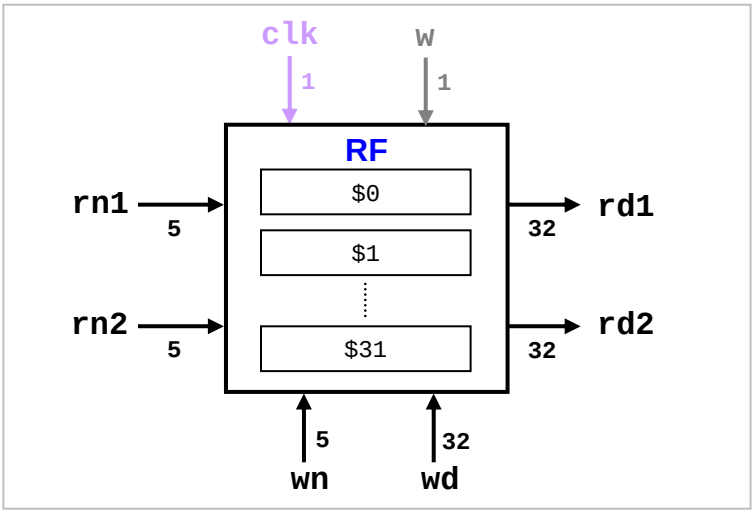
```
always
initial
   $monitor("%5d: clk=%b,d=%d,z=%d,expect=%d", $time,clk,d,z, e);
```

   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).

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

## LabM3.v

11. The next sequential component is the **register file**. It is made up of 32 registers ($0 thru $31) 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:




   In order to read the contents of two registers, we supply their register numbers thru **rn1** and **rn2** (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.

12. You can instantiate a register file using the statement

```
rf myRF(rd1, rd2, rn1, rn2, wn, wd, clk, w);
```

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

13. Write the program LabM3.v such that it starts by setting each register to the square of its register number. Something like this:

```
flag = $value$plusargs("w=%b", w);
for (i = 0; i < 32; i = i + 1)
begin
   clk = 0;
   wd = i * i;
   wn = i;
   clk = 1;
   #1;
end
```

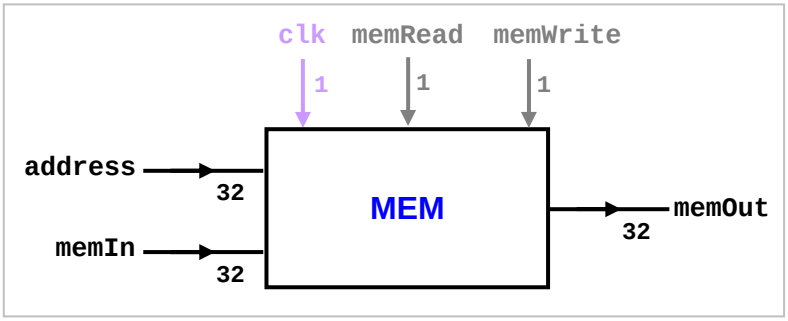
    The program must then generate random values for **rn1** and **rn2** and then output the corresponding contents of **rd1** and **rd2**. Do this in a loop that repeats 10 times.

14. Compile the program and run it with **w** set. Does it behave as expected?

15. Repeat with **w** cleared. Does the register file behave as expected?

## LabM4.v

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




    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**.

17. 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.

18. 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.

19. Write the value **32'h12345678** at address 16 and the value **32'h89abcdef** at address 24. Note that both are word addresses.

20. Read and display the contents of three words beginning at 16; i.e.

```
write = 0; read = 1; address = 16;

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

    Predict the output of the program before proceeding.

21. Compile and run LabM4.v. Ignoring the error message about “ram.dat”, does your program behave as expected?

22. Add a few lines to the program in which you attempt a read or a write using an un-aligned address. Run the modified program and explain its output.

## LabM5.v

23. The **mem** component has a handy feature that allows us to initialise 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 initialise 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.

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

```
          .text
main:     add     $t5, $0, $0       # index
          add     $s0, $0, $0       # sum
          add     $a0, $0, $0       # or reduction

loop:     lw      $t0, 0x50($t5)    # loop: t0 = array[t5]
          beq     $t0, $0, done     # if (t0 == 0) done
          add     $s0, $s0, $t0
          or      $a0, $a0, $t0
          addi    $t5, $t5, 4       # t5++
          j       loop

done:     sw      $s0, 0x20($0)     # done: save s0
          sw      $a0, 0x24($0)     # 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 in this and the following lab. Given a null-terminated array of words at address 0x50, 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 start with:

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

25. We also need to store the array itself beginning at 0x50. Add this to `ram.dat`:

```
@50  00000001  // array[0]
@54  00000003  // array[1]
@58  00000005  // array[2]
@5C  00000007  // array[3]
@60  00000009  // array[4]
@64  0000000B  // array[5]
@68  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 24.

26. In order to store the program itself (in machine language) load the code in `spim` and capture the machine encoding of each statement. We will load the entry point `main` at address 0x80. Here are the first few lines to be appended to your `ram.dat`:

```
@80  00006820  // add  $t5, $0, $0    # t5 = index
@84  00008020  // add  $s0, $0, $0    # s0 = sum
@88  00002020  // add  $a0, $0, $0    # a0 = or reduction
@8C  8da80050  // lw   $t0, 0x50($t5) # loop: t0 = array[t5]
@90  11000004  // beq  $t0, $0, done  # if (t0 == 0) done
...
```

The machine encoding is simply copied-and-pasted from `spim` with the exception of branch labels (for which `spim` over counts by 1) and jump labels (in which we must replace `spim`'s entry point address with our 0x80).

27. 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 = 128; write = 0; read = 1;
   repeat (11)
   ...
endmodule
```

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

## LabM6.v

29. 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:

```
0 0 0 13 32
0 0 0 16 32
0 0 0 4 32
35 13 8 80
4 8 0 4
0 16 8 16 32
0 4 8 4 37
8 13 13 4
2 35
43 0 16 32
43 0 4 36
```

30. 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:

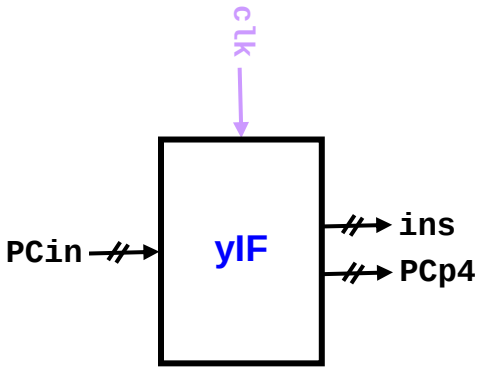
```
if (memOut[31:26] == 0)
```

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

```
if (memOut[31:26] == 2)
```

## LabM7.v

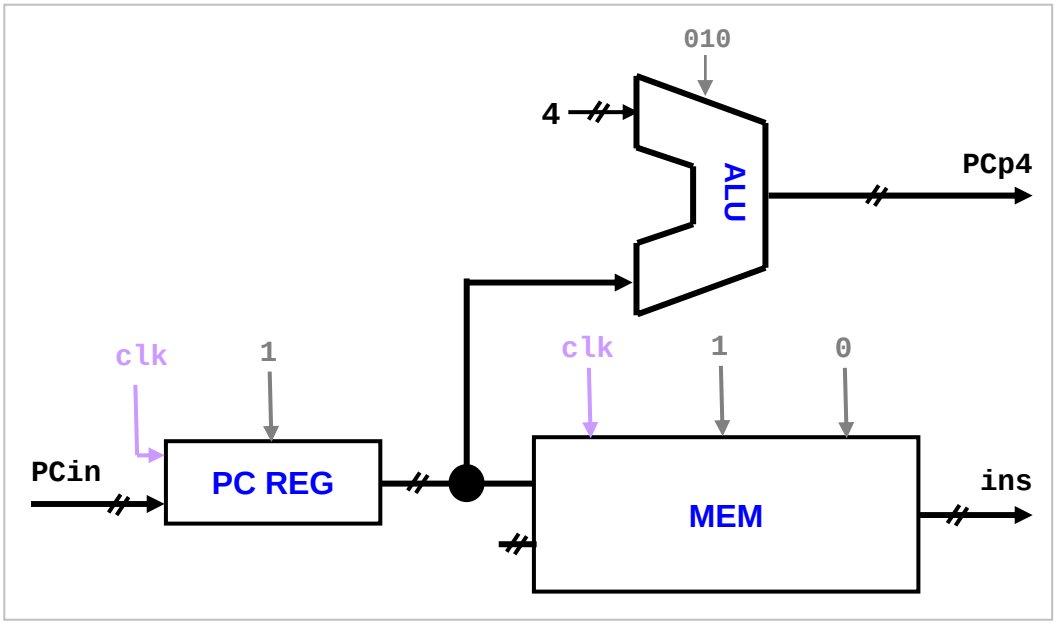
31. As a first step toward building the datapath of the CPU, let us implement a circuit for **i**nstruction **f**etch from memory. Here is its block diagram:




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 **PCp4 = PCin + 4** in anticipation of fetching the physically-following instruction.

32. 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.

33. 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:




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. Argue that the **clk** input of **mem** is irrelevant and can be left dangling.

34. 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. Note that fixed signals, such as `010` for the ALU) can be hard-coded parameters in the instantiation.

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

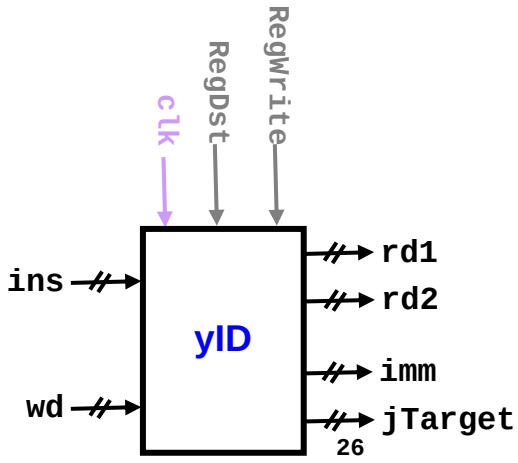
```
module labM;
reg [31:0] PCin;
reg clk;
wire [31:0] ins, PCp4;

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

initial
begin
   //-----------------------------------Entry point
   PCin = 128;
   //-----------------------------------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 the three instructions in your `ram.dat` file.

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

## LabM8.v

36. Next, we build a component for **i**nstruction **d**ecoding. Here is its block diagram:

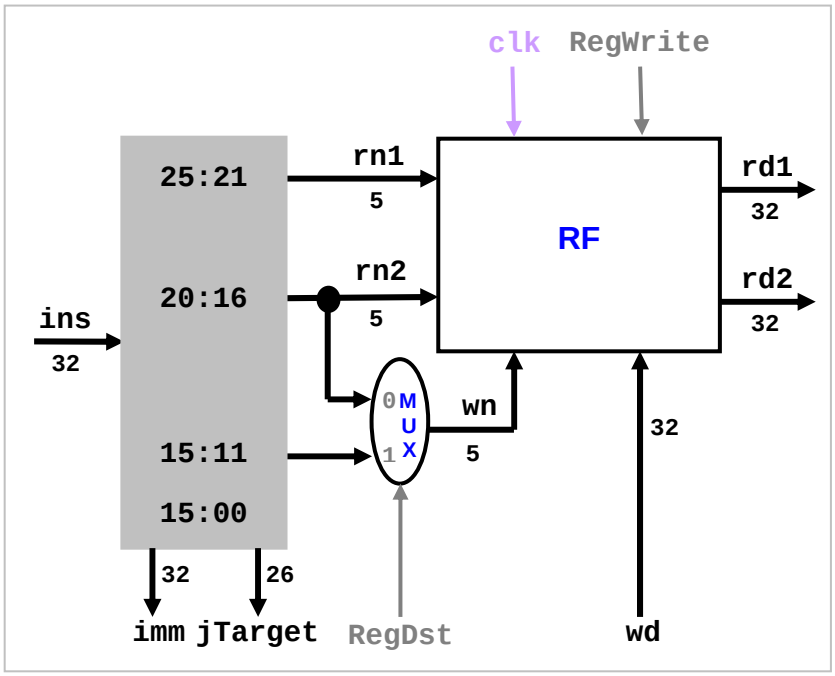



In fact, this component plays two roles: instruction **decoding** and data **write-back**.

37. In the decoding role, this component extracts the various fields of **ins** and looks up the needed registers. It produces in **rd1** and **rd2** the contents of the two registers `rs` and `rt`; in **imm** the sign-extended immediate; and in **jTarget** the (26-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.

38. 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 20:16 or 15:11 of **ins**, depending on the **RegDst** input signal. In addition, writing is only to be done if the **RegWrite** signal is set and then only at the positive edge of **clk**.

39. The circuit diagram of **yID** is shown below.




40. Add the following module to your cpu.v file:

```
module yID(rd1, rd2, imm, jTarget, ins, wd, RegDst, RegWrite, clk);
output [31:0] rd1, rd2, imm;
output [25:0] jTarget;
input [31:0] ins, wd;
input RegDst, RegWrite, clk;
```

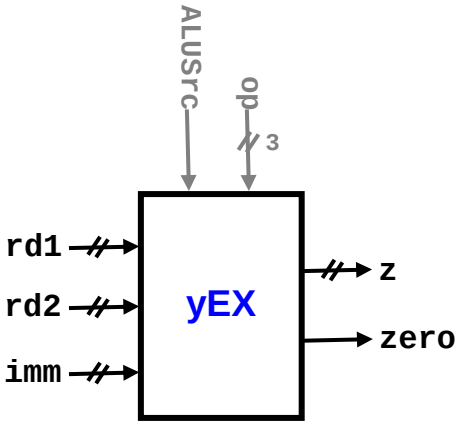
41. Complete the **yID** module. Use part-select to extract the needed parts, e.g.

```
assign rn1 = ins[25:21];
```

42. The immediate field in an I-type instruction is in bits 0 through 15. And since we will focus on *signed* instructions, we need to sign extended this to 32 bits:

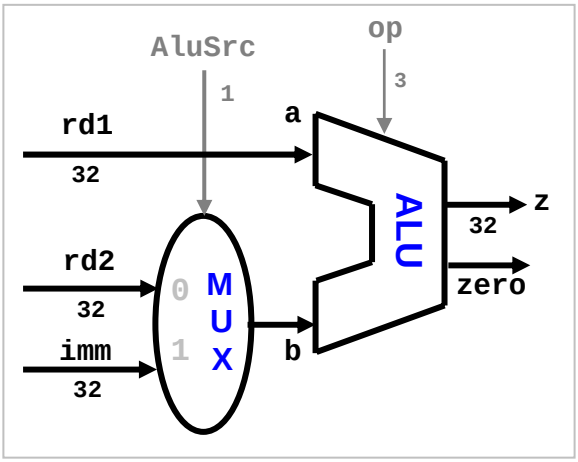
```
assign imm[15:0] = ins[15:0];
yMux #(16) se(imm[31:16], zeros, ones, ins[15]);
```

43. Next, we build a component for **ex**ecuting the instruction. Here is its block diagram:




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.

44. The circuit diagram of **yEx** is shown below.




The circuit follows directly from the definition of **yEX**.

45. 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;
```

46. Complete the development of **yEX**.

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

```
module labM;
reg [31:0] PCin;
reg RegDst, 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, ins, wd, RegDst, RegWrite, clk);
yEX myEx(z, zero, rd1, rd2, imm, op, ALUSrc);
assign wd = z;

initial
begin
   //-----------------------------------Entry point
   PCin = 128;

   //-----------------------------------Run program
   repeat (11)
   begin
      //---------------------------------Fetch an ins
      clk = 1; #1;

      //---------------------------------Set control signals
      RegDst = 0; RegWrite = 0; ALUSrc = 1; op = 3'b010;
      // Add statements to adjust the above defaults

      //---------------------------------Execute the ins
      clk = 0; #1;

      //---------------------------------View results
      ...

      //---------------------------------Prepare for the next ins
      PCin = PCp4;
   end
   $finish;
end
endmodule
```

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

48. Complete LabM8 by adding statements to detect the instruction being executed and accordingly change the default control signals (we are only concerned with the instructions in our `ram.dat`). Here is an example of what needs to be done:

```
if (ins[31:26] == 0)
   begin
   RegDst = 1; RegWrite = 1; ALUSrc = 0;
end
```

49. In the "*View results*" section, display the following signals:

```
ins, rd1, rd2, imm, jTarget, z, zero
```

50. Compile and run LabM8 as follows:

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

Your output should be equivalent to the one shown below. Examine each line and argue that it is consistent with our datapath. Note that our datapath does *not* have data memory and has no support for branches of jumps.

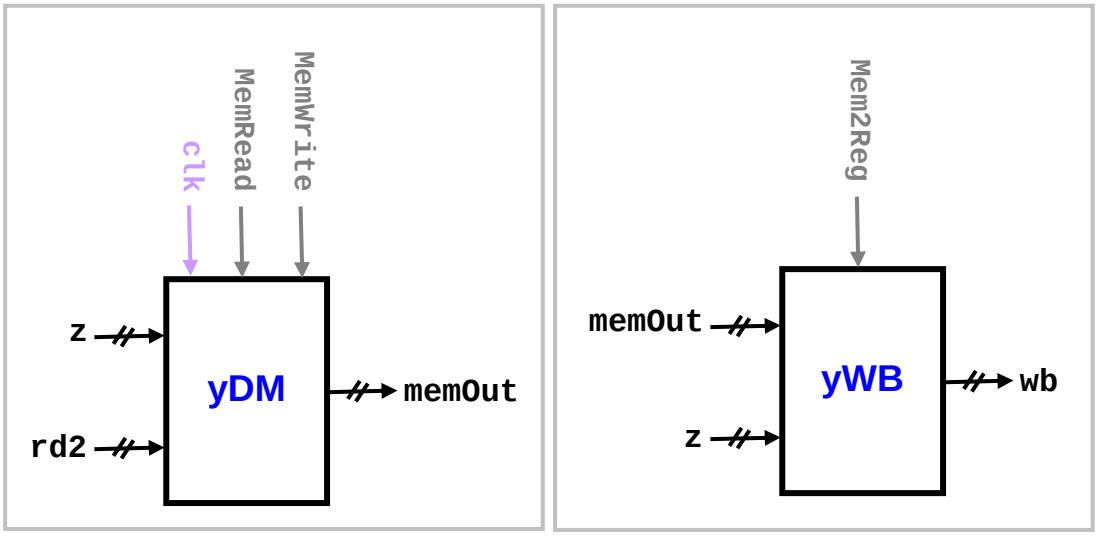
```
00006820: rd1= 0 rd2= 0 imm=00006820 jTarget=0006820 z= 0 zero=1
00008020: rd1= 0 rd2= 0 imm=ffff8020 jTarget=0008020 z= 0 zero=1
00002020: rd1= 0 rd2= 0 imm=00002020 jTarget=0002020 z= 0 zero=1
8da80050: rd1= 0 rd2= x imm=00000050 jTarget=1a80050 z=80 zero=0
11000004: rd1=80 rd2= 0 imm=00000004 jTarget=1000004 z=80 zero=0
02088020: rd1= 0 rd2=80 imm=ffff8020 jTarget=2088020 z=80 zero=0
00882025: rd1= 0 rd2=80 imm=00002025 jTarget=0882025 z=80 zero=0
21ad0004: rd1= 0 rd2= 0 imm=00000004 jTarget=1ad0004 z= 4 zero=0
08000023: rd1= 0 rd2= 0 imm=00000023 jTarget=0000023 z=35 zero=0
ac100020: rd1= 0 rd2=80 imm=00000020 jTarget=0100020 z=32 zero=0
ac040024: rd1= 0 rd2=80 imm=00000024 jTarget=0040024 z=36 zero=0
```

## LabM9.v

51. To complete our datapath we need the two more components:

- **yDM** (**d**ata **m**emory): a data memory unit that reads from address **z** or writes **rd2** to that address based on two control signals and a clock.
- **yWB** (**w**rite **b**ack): a 2-to-1 mux that selects either **memOut** or **z** based on whether the control signal **Mem2Reg** is 1 or 0, respectively.

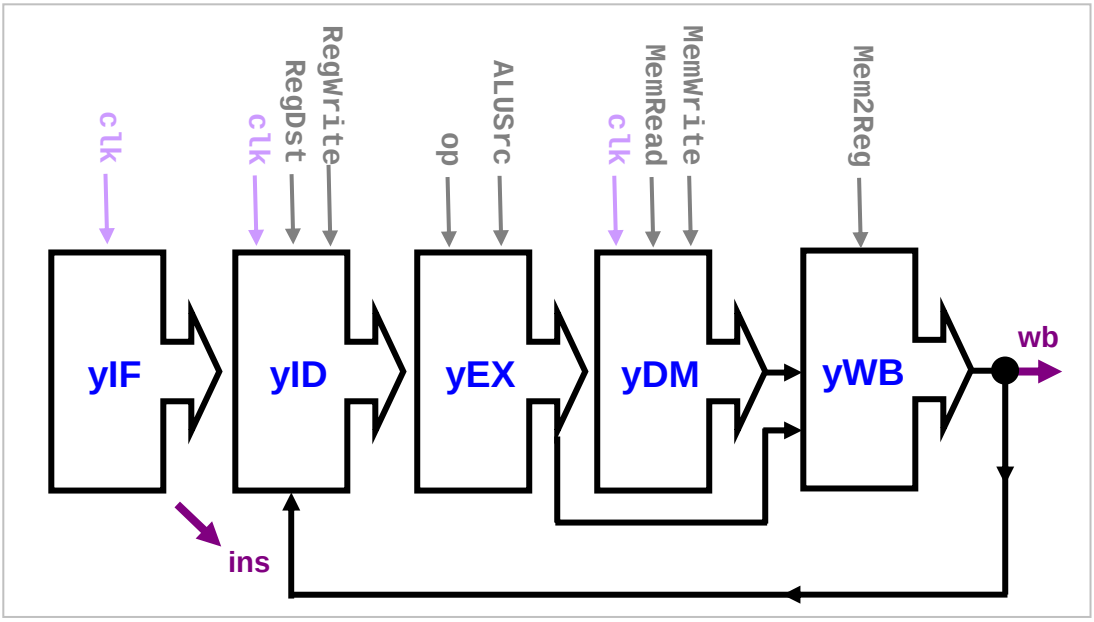
Here are the block diagrams of these components:



52. Add the following two modules to your cpu.v file and supply the missing lines:

```
module yDM(memOut, exeOut, rd2, clk, MemRead, MemWrite);
output [31:0] memOut;
input [31:0] exeOut, rd2;
input clk, MemRead, MemWrite;

// instantiate the circuit (only one line)

endmodule

//-----------------------------------------------------------------

module yWB(wb, exeOut, memOut, Mem2Reg);
output [31:0] wb;
input [31:0] exeOut, memOut;
input Mem2Reg;

// instantiate the circuit (only one line)

endmodule
```

53. We now have all the pieces needed to build our datapath. These components fit together like a jigsaw puzzle as shown in the diagram below.




54. Save LabM8.v as LabM9.v and modify it so it instantiates as follows:

```
yIF myIF(ins, PCp4, PCin, clk);
yID myID(rd1, rd2, imm, jTarget, ins, wd, RegDst, RegWrite, clk);
yEX myEx(z, zero, rd1, rd2, imm, op, ALUSrc);
yDM myDM(memOut, z, rd2, clk, MemRead, MemWrite);
yWB myWB(wb, z, memOut, Mem2Reg);
assign wd = wb;
```

55. You will need to upgrade the "*Set control signals*" section in order to issue three new signals **`MemRead`**, **`MemWrite`**, and **`Mem2Reg`**. Again, focus only on the instructions in our program in `ram.dat`.

56. Have the program output the results as follows:

```
$display("%h: rd1=%2d rd2=%2d z=%3d zero=%b wb=%2d",
         ins, rd1, rd2, z, zero, wb);
```

57. Compile and run your program. You should obtain the following output:

```
00006820: rd1= 0 rd2= 0 z=  0 zero=1 wb= 0
00008020: rd1= 0 rd2= 0 z=  0 zero=1 wb= 0
00002020: rd1= 0 rd2= 0 z=  0 zero=1 wb= 0
8da80050: rd1= 0 rd2= x z= 80 zero=0 wb= 1
11000004: rd1= 1 rd2= 0 z=  1 zero=0 wb= 1
02088020: rd1= 0 rd2= 1 z=  1 zero=0 wb= 1
00882025: rd1= 0 rd2= 1 z=  1 zero=0 wb= 1
21ad0004: rd1= 0 rd2= 0 z=  4 zero=0 wb= 4
08000023: rd1= 0 rd2= 0 z= 35 zero=0 wb=35
ac100020: rd1= 0 rd2= 1 z= 32 zero=0 wb=32
ac040024: rd1= 0 rd2= 1 z= 36 zero=0 wb=36
```

## LabM10.v

58. Save LabM9.v as LabM10.v and modify its "*Prepare for the next ins*" section to incorporate branches and jumps: rather than always setting **`PCin`** to **`PCp4`**, it should set it based on the instruction and its results:

```
//--------------------------------Prepare for the next ins
if (beq && zero == 1)
   PCin = PCp4 + imm shifted left twice;
else if (j)
   PCin = jTarget shifted left twice;
else
   PCin = PCp4;
```

59. Argue that the loop must now repeat *43 times* rather than 11.

60. Compile and run your program. The last two lines of the output should be:

```
ac100020: rd1= 0 rd2=36 z= 32 zero=0 wb=32
ac040024: rd1= 0 rd2=15 z= 36 zero=0 wb=36
```

    If not, trace the error back to the source by examining earlier output lines and then determining if the problem is control-signal or datapath related.

# LAB M
# Notes

- The **`rf`** module is in fact parameterized by a debugging parameter named `DEBUG`. If you set this parameter to 1 (0) then it turns debugging on (off). In debug mode, every register that is read from or written to is displayed on the screen, which can be very helpful in diagnosing problems. To activate this mode, instantiate **`rf`** as follows:

```
rf #(1) myRF(rd1, rd2, rn1, rn2, wn, wd, clk, RegWrite);
```

- The **`mem`** module is in fact parameterized by the memory size. If a parameter is not supplied upon instantiation, the size defaults to `16'hffff`, i.e. to 65,535 words. If you need to instantiate **`mem`** with a different size, say 100 bytes, do this:

```
mem #(100) data(memOut, address, memIn, clk, read, write);
```

- The datapath built in this lab can handle the following instructions: `and`, `or`, `add`, `sub`, `slt`, `addi`, `lw`, `sw`, `beq`, and `j`.

- The CPU built in this lab cannot self-adapt depending on the instruction. Its datapath has all the computational units needed for executing the above instructions but must rely on an external agent to supply the correct control signals for each instruction.

- The next lab will add a control unit to the CPU so that control signal generation will be automated. The resulting CPU will be able to execute programs without needing any signal other than a clock.