LAB N

Automating the Control

Perform the following groups of tasks:

LabN1.v

1. Create a directory for this lab and copy to it the files `cpu.v` and `ram.dat` that were created in the previous lab. Copy also the file `LabM10.v` and name it `LabN1.v`.

2. Recall that LabN1 determines the address of the instruction to be executed next (i.e. **`PCin`**) based on whether we are jumping, branching, or continuing sequentially. But rather than representing this logic behaviourally in a testing module, let us implement it structurally in a circuit whose output is **`PCin`**.

3. But if **`PCin`** is an *output* of a circuit, how can you ever set it in order to fetch the very first instruction of your program? We clearly need a mechanism to force the CPU to stop the current program and switch to another. To that end, let us introduce two new signals: **`INT`**, a 1-bit *interrupt* signal, and **`entryPoint`**, a 32-bit signal containing the address to switch to. Our **`PCin`** logic now becomes:

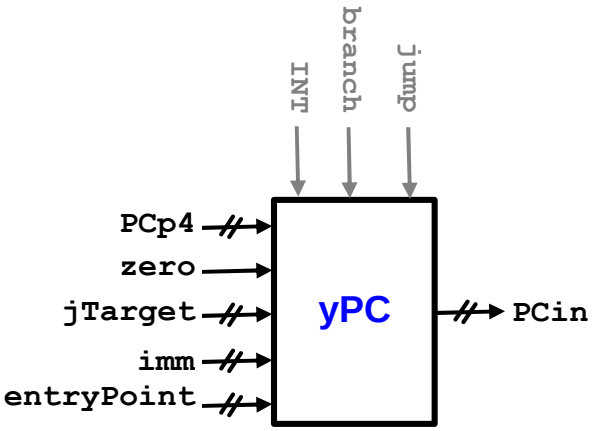
```
//---------------------------------Prepare for the next ins
if (INT == 1)
   PCin = entryPoint;
else
   if (beq && zero == 1)
      PCin = PCp4 + imm shifted left twice;
   else if (j)
      PCin = jTarget shifted left twice;
   else
      PCin = PCp4;
```

   Note that if the interrupt signal **`INT`** is set, we fetch the next instruction from address **`entryPoint`** thereby affecting a context switch. With this new scheme, **`PCin`** is no longer set externally and, hence, can indeed be an output of a circuit.

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

```
module yPC(PCin, PCp4,INT,entryPoint,imm,jTarget,zero,branch,jump);
output [31:0] PCin;
input [31:0] PCp4, entryPoint, imm;
input [25:0] jTarget;
input INT, zero, branch, jump;
```

The **yPC** component takes 8 inputs and determines **PCin** accordingly. The jump and branch inputs are simply flags that are set to 1 if the current instruction is a jump or a branch on equal. Here is the block diagram of the component.



5. To implement this module, we clearly need several multiplexers to choose between alternates. Since we have three nested if statements, we will need three mux's.

6. The first mux chooses between sequential processing (i.e. **PCp4**) and branching. Its control signal is the `and` of **branch** ad **zero**; i.e. we branch if this is a `beq` instruction and its registers are equal. The branch target address, **bTarget**, is computed by multiplying **imm** by 4 and adding the result to **PCp4**.

```
wire [31:0] immX4, bTarget, choiceA;
wire doBranch, zf;

assign immX4[31:2] = imm[29:0];
assign immX4[1:0] = 2'b00;
yAlu myALU(bTarget, zf, PCp4, immX4, 3'b010);
and (doBranch, branch, zero);
yMux #(32) mux1(choiceA, PCp4, bTarget, doBranch);
```

7. The second mux chooses between the previous mux output (**choiceA**) and jumping. Its control signal is **jump**. The jump target is a 32-bit address whose high-order 4 bits come from **PCp4** and whose lower 26 bits are the input **jTarget**.

8. The third mux chooses between the previous mux output and **entryPoint**. Its control signal is **INT**.

9. Complete the development of the **yPC** module.

10. Modify LabN1.v so that it instantiates **yPC** in addition to the five components it already instantiates. This requires changing **PCin** from **reg** to **wire** and adding declarations for the new signals.

11. In addition, modify the body of LabN1.v so it starts with a context switch to launch our program. Here is the new template:

```
initial
begin
   //-----------------------------------Entry point
   entryPoint = 128; INT = 1; #1;

   //-----------------------------------Run program
   repeat (43)
   begin
      //---------------------------------Fetch an ins
      clk = 1; #1; INT = 0;

      //---------------------------------Set control signals
      as before but add branch and jump

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

      //---------------------------------View results
      as before

      //---------------------------------Prepare for the next ins
      do nothing!

   end
   $finish;
end
```

Notice that the "*Set control signals*" section must now detect **beq** and **j** and set the two signals **branch** and **jump** accordingly. Note also that the "*Prepare for the next ins*" section is now empty since its behavioural logic has been promoted to a circuit.

12. Compile and run LabN1. The generated output should be exactly as in the previous lab. Specifically, 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
```

yC1

13. Our "*Set control signals*" section sets *eight* control signals:

```
RegDst, ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite, jump, branch
```

(It also sets the 3-bit **op** signal but let us ignore that for now.) We seek to automate the generation of these eight signals by building structural circuits that output them.

14. As a first step toward this goal, let us build the circuit **yC1** that takes the **opCode** as input (i.e. **ins[31:26]**) and determines if the instruction is load, store, branch-on-equal, jump, or R-type, and outputs **lw**, **sw**, **branch**, **jump**, or **rtype** accordingly:

```
module yC1(rtype, lw, sw, jump, branch, opCode);
output rtype, lw, sw, jump, branch;
input [5:0] opCode;
```

15. We generate **lw** by noting that the opCode of the lw instruction is 100011. Hence, if we and the 6 bits bits of **opCode** after negating the ones that are 0 for lw, the result will only be 1 if **opCode** were indeed 100011. This leads to the following circuit:

```
wire not4, not3, not2;
not (not4, opCode[4]);
not (not3, opCode[3]);
not (not2, opCode[2]);
and (lw, opCode[5], not4, not3, not2, opCode[1], opCode[0]);
```

16. The signals **sw**, **branch**, and **jump** can be generated similarly by noting the opCode of the corresponding instructions.

17. The **rtype** signal is generated by detecting the case of all 6 bits of opCode being 0. This can be done in a single instantiation.

18. Add the **yC1** module to your cpu.v file and complete its development. Its body should have about 10 lines. You cannot test this module yet because it generates only two of the eight control signals needed for the datapath.

yC2

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

```
module yC2(RegDst, ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite,
           rtype, lw, sw, branch);
output RegDst, ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite;
input rtype, lw, sw, branch;
```

    This module represents the second part of the control unit (hence the **C** in its name). It takes four of the signals generated by **yC1** as input, and generates the six control signals we need.

20. To build this component, we need to implement the logic of the "*Set control signals*" section in hardware. To that end, we switch from sequential, if-then-else thinking, to parallel, declarative thinking, and ask: What should the value of **RegDst** be?

21. Looking at all the cases, and manipulating don't-cares to our advantage, we see that **RegDst** has to be 1 for R-type instructions and 0 otherwise. Hence, this signal can be generated in one line:

```
assign RegDst = rtype;
```

22. Consider **ALUSrc** next. This signal must be 0 for R-types and branches, 1 for loads and stores and addi, and don't-care for jumps. Again, we manipulate the don't-care and treat it as 1 so as to end up with a simple rule: 0 for R-types and branches and 1 otherwise. This leads to the one-liner:

```
nor (ALUSrc, rtype, branch);
```

23. Generating the remaining signals can be done similarly.

24. Complete the development of **yC2**. Its body should have about 6 lines.

LabN2.v

25. Save LabN1.v as LabN2.v and modify it so it instantiates the two parts of the control unit in addition to the datapath components:

```
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;
yPC myPC(PCin, PCp4,INT,entryPoint,imm,jTarget,zero,branch,jump);

assign opCode = ins[31:26];
yC1 myC1(rtype, lw, sw, jump, branch, opCode);
yC2 myC2(RegDst, ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite,
         rtype, lw, sw, branch);
```

26. In addition, change the declaration of the eight control signals from **reg** to **wire** and remove their references from the Our "*Set control signals*" section:

```
initial
begin
   //-----------------------------------Entry point
   entryPoint = 128; INT = 1; #1;

   //-----------------------------------Run program
   repeat (43)
   begin
      //---------------------------------Fetch an ins
      clk = 1; #1; INT = 0;

      //---------------------------------Set control signals
      set only the op signal

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

      //---------------------------------View results
      as before

      //---------------------------------Prepare for the next ins
      do nothing!

   end
   $finish;
end
```

Except for **op**, our CPU has become capable of self-setting the signals it needs.

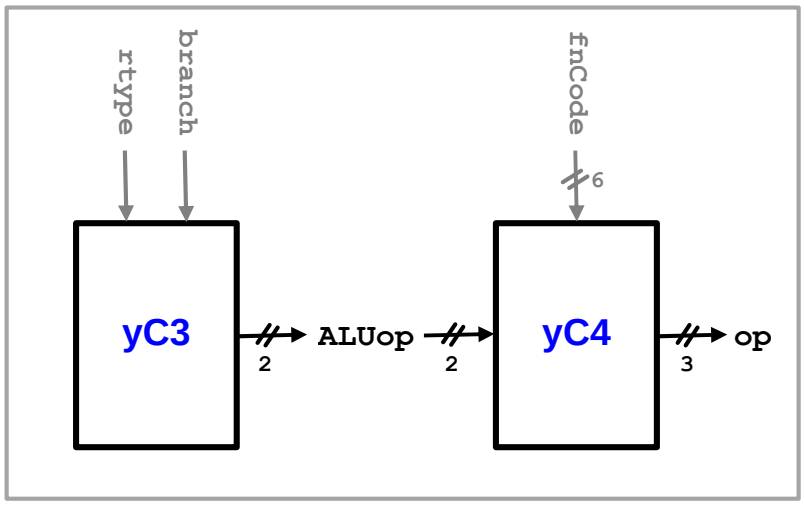
27. Compile and run LabN2. The generated output should be exactly as in the previous lab. Specifically, The last two lines of the output should be:

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

yC3

28. We now turn our attention to the **op** signal. This is the hardest control signal to generate because it depends sensitively on the instruction. Indeed, we may need to look at both the `opCode` (**ins[31:26]**) and the `fnCode` (**ins[5:0]**) before becoming able to determine the correct **op** value.

29. We overcome the above difficulty by dividing the problem into two and building two back-to-back circuits: The first, **yC3**, is responsible for non-R-type instructions and the second, **yC4**, takes care of R-types. These two circuits interact with each other through a new 2-bit signal **ALUop** as shown in this block diagram:



30. The **yC3** circuit must generate **ALUop** as shown in the table below. Note that since `j` doesn't involve the ALU, it doesn't matter what operation is performed. Note also that **yC3** cannot determine the operation for R-types since it doesn't see the `fnCode`.

Type	Instruction	Operation	ALUop
I	**lw**	addition	00
	sw	addition	00
	addi	addition	00
	beq	subtraction	01
J	**j**	don't-care	**xx**
R	**unknown**	unknown	10

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

```
module yC3(ALUop, rtype, branch);
output [1:0] ALUop;
input rtype, branch;

// build the circuit
// Hint: you can do it in only 2 lines

endmodule
```

yC4

32. We now turn our attention to the fourth and last part of our control unit, **yC4**. This unit sees the `fnCode` and the **ALUop** signal generated by **yC3** and outputs the 3-bit ALU signal **op**. Because of this, it is sometimes referred to as the *ALU Control Unit*. Here is the specification of this unit:

ALUop	funCode	Instruction	Operation	op
00	*don't-care*	*don't-care*	addition	010
01	*don't-care*	*don't-care*	subtraction	110
10	100100	and	conjunction	000
10	100101	or	disjunction	001
10	100000	add	addition	010
10	100010	sub	subtraction	110
10	101010	slt	set-on-less-than	111

As you can see, this unit operates primarily based on **ALUop**. If this signal is 00 or 01 then **yC4** *trusts* and findings of **yC3** and generates **op** accordingly. But if **ALUop** is 10 then **yC4** *knows* that this is an R-type instruction and hence generates **op** based on the function code.

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

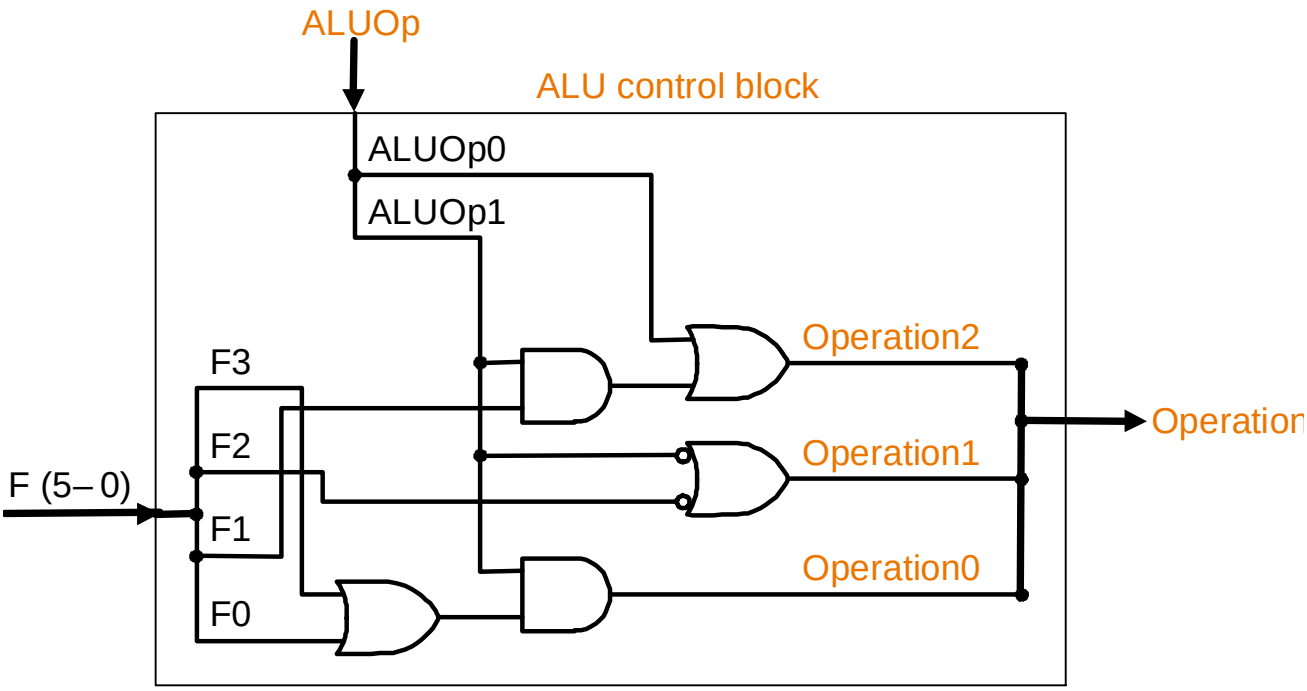
```
module yC4(op, ALUop, fnCode);
output [2:0] op;
input [5:0] fnCode;
input [1:0] ALUop;

// instantiate and connect

endmodule
```

To implement this circuit, see the diagram discussed below.

34. Consider the following circuit which is made up of five simple gates:



35. Argue that this circuit does indeed behave as required by the table. At first glance, it may seem impossible that any circuit can sometimes ignore one of its inputs but this one does: `fnCode` is ignored if **`ALUop`** is `00` or `01`.

36. Complete the development of **`yC4`**. Its body should have exactly five lines since it is made up of five primitive gates.

LabN3.v

37. Save LabN2.v as LabN3.v and modify its instantiation section 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;
yPC myPC(PCin, PCp4,INT,entryPoint,imm,jTarget,zero,branch,jump);
assign opCode = ins[31:26];
yC1 myC1(rtype, lw, sw, jump, branch, opCode);
yC2 myC2(RegDst, ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite,
         rtype, lw, sw, branch);
assign fnCode = ins[5:0];
yC3 myC3(ALUop, rtype, branch);
yC4 myC4(op, ALUop, fnCode);
```

38. In addition, change the declaration of the **op** signal from **reg** to **wire** and remove it from the Our "*Set control signals*" section:

```
initial
begin
   //-----------------------------------Entry point
   entryPoint = 128; INT = 1; #1;

   //-----------------------------------Run program
   repeat (43)
   begin
      //---------------------------------Fetch an ins
      clk = 1; #1; INT = 0;

      //---------------------------------Set control signals
      do nothing!

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

      //---------------------------------View results
      as before

      //---------------------------------Prepare for the next ins
      do nothing!

   end
   $finish;
end
```

Notice that the control signal section has become empty. The CPU is now capable of executing the program without any assistance from external modules.

39. Compile and run LabN3. The generated output should be exactly as in the previous lab. Specifically, 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
```

LabN4.v

40. In this task we repackage our components so as to fully separate the concerns. Let us put all the needed instantiation in one module that represents the CPU chip:

```
module yChip(ins, rd2, wb, entryPoint, INT, clk);
output [31:0] ins, rd2, wb;
input [31:0] entryPoint;
input INT, clk;
```

In fact, this module needs not have any output (the program makes changes to the registers and to memory) but we have declared **ins**, **wb**, and **rd2** simply to be able to test it (**rd2** helps us test **sw)**.

41. Add the **yChip** module to your cpu.v file and complete its development. You simply need to copy all the instantiation lines, along with their corresponding declarations, from LabN3 to the body of this module.

42. Save LabN3 as LabN4.v and modify it by replacing all the instantiated circuits with an instantiation of **yChip**:

```
module labN;
reg [31:0] entryPoint;
reg clk, INT;
wire [31:0] ins, rd2, wb;

yChip myChip(ins, rd2, wb, entryPoint, INT, clk);

initial
begin
   //-----------------------------------Entry point
   entryPoint = 128; INT = 1; #1;

   //-----------------------------------Run program
   repeat (43)
   begin
      //---------------------------------Fetch an ins
      clk = 1; #1; INT = 0;

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

      //---------------------------------View results
      $display("%h: rd2=%2d wb=%2d", ins, rd2, wb);

   end
   $finish;
end
endmodule
```

43. Compile and run LabN4. The generated output should be similar to the previous. In particular, the last two lines of the output should be:

```
ac100020: rd2=36 wb=32
ac040024: rd2=15 wb=36
```

LAB N
Notes

- The CPU built in this Lab communicates with the outside world through four channels:
  - The clock signal (input)
  - The interrupt signal (input)
  - The entry point signal (input)
  - The BIU (Bus Interface Unit in **`yIF`** and **`yDM`**) (input and output)

- The clock rate is determined based on the longest path that an instruction takes. For the subset we considered, this would be the **`lw`** instruction.