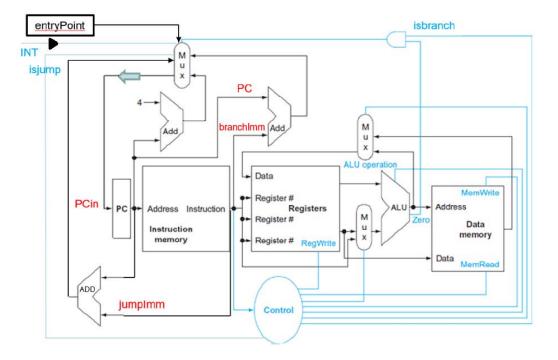# LAB N

Automating the Control

Note: <u>We make a RISK-V CPU in which memory architecture is 32 bits. In lecture time we covered a 64 bits RISK-V CPU. Thus, the length of registers are 32 bits as well as memory addressing.</u>

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 the single cycle RISK-V data path.

3. 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 behaviorally in a testing module, let us implement it structurally in a circuit whose output is **PCin**.

4. Before we proceed, we need to add one more output to our yIF from lab M, i.e., PC, which a register defined in yIF. Here is the completed yIF as follows:

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

wire zero;
wire read, write, enable;
wire [31:0] a, memIn;
wire [2:0] op;

register #(32) pcReg(PC, PCin, clk, enable);
mem insMem(ins, PC, memIn, clk, read, write);
yAlu myAlu(PCp4, zero, a, PC, op);

assign enable = 1'b1;
assign a = 32'h0004;
assign op = 3'b010;
assign read = 1'b1;
assign write = 1'b0;

endmodule
```
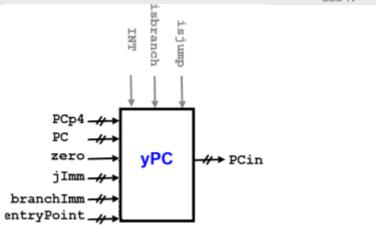
5. But if `PCin` is an *output* of a circuit, how can you ever set it in order to fetch the very first instruction of your program? We clearly need a mechanism to force the CPU to stop the current program and switch to another. To that end, let us introduce two new signals: `INT`, a 1-bit *interrupt* 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 = PCin + branchImm shifted left twice;
   else if (jal)
      PCin = PCin + jImm 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.

6. We want to add the following yPC to our cpu:

```
module yPC(PCin,PC,PCp4,INT,entryPoint,branchImm,jImm,zero,
           isbranch,isjump);
```

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.

INT
isbranch
isjump

```
PCp4 ──#→
PC   ──#→
zero ────→        yPC        ──#→ PCin
jImm ──#→
branchImm ──#→
entryPoint ──#→
```

7. 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.

8. 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**.

```verilog
module yPC(PCin, PC, PCp4,INT,entryPoint,branchImm,jImm,zero,isbranch,isjump);

output [31:0] PCin;
input [31:0] PC, PCp4, entryPoint, branchImm;
input [31:0] jImm;
input INT, zero, isbranch, isjump;
wire [31:0] branchImmX4, jImmX4, jImmX4PPCp4, bTarget, choiceA, choiceB;
wire doBranch, zf;

// Shifting left branchImm twice
assign branchImmX4[31:2] = branchImm[29:0];
assign branchImmX4[1:0] = 2'b00;

// Shifting left jump twice
assign jImmX4[31:2] = jImm[29:0];
assign jImmX4[1:0] = 2'b00;

// adding PC to shifted twice, branchImm
yAlu bALU(bTarget, zf, ?, ?, ?);

// adding PC to shifted twice, jImm
yAlu jALU(jImmX4PPCp4, zf, ?, ?, ?);

// deciding to do branch
and (doBranch, isbranch, zero);

yMux #(32) mux1(choiceA, PCp4, bTarget, doBranch);

yMux #(32) mux2(choiceB, choiceA, jImmX4PPCp4, isjump);

yMux #(32) mux3(?, choiceB, entryPoint, INT);


endmodule
```

9. The second mux chooses between the previous mux output (**choiceA**) and jumping. Its control signal is **isjump**. The jump target is calculated from shift left twice of **jImm**.

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

INT.

11. Replace ? in the yPC module with proper entries.

12. 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.

13. 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 = 32'h28; INT = 1; #1;

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

        RegWrite = 0; ALUSrc = 1; op = 3'b010;
        MemRead = 0; MemWrite = 0; Mem2Reg = 0;
        //-----------Set control signals
        as before but add isbranch and isjump

        //-------------------------------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 **jal** and set the two signals **isbranch** and **isjump** accordingly. Note also that the "*Prepare for the next ins*" section is now empty since its behavioral logic has been promoted to a circuit.

14. 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:

```
02802023: rd1= 0 rd2=36 exeOut= 32 zero=0 wb=32
02a02223: rd1= 0 rd2=15 exeOut= 36 zero=0 wb=36
```

yC1

15. Our "*Set control signals*" section sets *seven* control signals:

   ALUSrc, RegWrite, Mem2Reg, MemRead, MemWrite, isjump, isbranch

(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.

16. As a first step toward this goal, let us build the circuit **yC1** that takes the **opCode** as input (i.e. **ins[6:0]**) and determines if the instruction is load, store, branch-on- equal, jump, or R-type, and outputs **islw**, **isStype (for sw)**, **isbranch**, **isjump**, or **isRtype** accordingly:

```
module yC1(isStype, isRtype, isItype, isLw, isjump, isbranch, opCode);
output isStype, isRtype, isItype, isLw, isjump, isbranch;
input [6:0] opCode;
wire lwor, ISselect, JBselect, sbz, sz;

//    opCode
// lw        0000011
// I-Type    0010011
// R-Type    0110011
// SB-Type   1100011
// UJ-Type   1101111
// S-Type    0100011

// Detect UJ-type
assign isjump=opCode[?];

// Detect lw
or (lwor, opCode[?], opCode[?], opCode[?], opCode[?], opCode[?]);
not (isLw, lwor);

// Select between S-Type and I-Type
xor (ISselect, opCode[?], opCode[?], opCode[?], opCode[?], opCode[?]);
and (isStype, ISselect, opCode[?]);
and (isItype, ISselect, opCode[?]);

// Detect R-Type
and (isRtype, opCode[?], opCode[?]);

// Select between JAL and Branch
and (JBselect, opCode[?], opCode[?]);
not (sbz, opCode[?]);
and (isbranch, JBselect, sbz);

endmodule
```

17. Ignoring the first two bits of **opcode**, which bit is the distinguisher for the **jal** instruction?

18. Given the given code you can ask the same type of question for each type. Add yC1 to your cpu.v file and replace ? with a proper value.

<div style="background-color:#b49be0; text-align:center; color:white; font-weight:bold; padding:4px;">LabN2.v</div>

19. Given you have implemented yC1 properly, save LabN1.v into LabN2.v instantiate an yC1 module, change **isjump** and **isbranch** from register to wire and remove all assignment for these two signals. Compile and run your code. You should get the same result as before. If you don't get the same result, it means you made a mistake in yC1 then you need to revise your module again. Note that yC1 produce **isStype, isRtype, isItype** and **isLw** therefore you need to define corresponding wires for them in LanN2.v.

20. Now we seek to set signals for **ALUSrc, RegWrite, Mem2Reg, MemRead** and **MemWrite** using the output of yC1. If you pay attention to the RISK-V single cycle data path, given the output of yC1, we can set all these of these signals. Add the following module to your cpu.v file:

```
yC2(RegWrite, ALUSrc, MemRead, MemWrite, Mem2Reg,
        isStype, isRtype, isItype, isLw, isjump, isbranch);
  output RegWrite, ALUSrc, MemRead, MemWrite, Mem2Reg;
  input isStype, isRtype, isItype, isLw, isjump, isbranch;

  // You need two or gates and 3 assignments;

Endmodule
```

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

21. To build this component, we need to implement the logic of the "*Set control signals*" section in hardware.

22. To that end, we switch from sequential, if-then-else thinking, to parallel, declarative thinking, and ask: What should the value of **RegWrite** be if you know the instruction type?

23. Complete the module, save LabN2.v into LabN3.v instantiate an yC2 module. In LabN3.v change the definition of **RegWrite, ALUSrc, MemRead, MemWrite** and **Mem2Reg** to wire from reg and remove setting of signals in the body of repeat structure. The output should be the same as before meaning control signals are detected and set by yC2.

```
    yIF myIF(ins, PC, PCp4, PCin, clk);
    yID myID(rd1, rd2, imm, jTarget, branch, ins, wd, RegWrite, clk);
    yEX myEx(exeOut, zero, rd1, rd2, imm, op, ALUSrc);
    yDM myDM(memOut, exeOut, rd2, clk, MemRead, MemWrite);
    yWB myWB(wb, exeOut, memOut, Mem2Reg);
    assign wd = wb;
    yPC myPC(PCin, PC, PCp4, INT, entryPoint, branch, jTarget, zero, isbranch,
    isjump);
    assign opCode = ins[6:0];
    yC1 myC1(isStype, isRtype, isItype, isLw, isjump, isbranch, opCode);
    yC2 myC2(RegWrite, ALUSrc, MemRead, MemWrite, Mem2Reg,
              isStype, isRtype, isItype, isLw, isjump, isbranch);
```

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

```
    initial begin
        //----------------------------------Entry point
        entryPoint = ?; 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
```
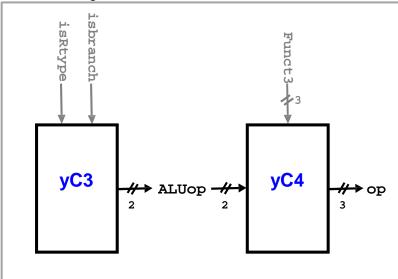
Except for **op**, our CPU has become capable of self-setting the signals it needs.

25. 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:

```
02802023: rd1= 0 rd2=36 exeOut= 32 zero=0 wb=32
02a02223: rd1= 0 rd2=15 exeOut= 36 zero=0 wb=36
```

<div align="center">

yC3

</div>

26. 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[6:0]**) and the funct3 (**ins[14:12]**) before becoming able to determine the correct **op** value.

27. 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:

28. The **yC3** circuit must generate **ALUop** as shown in the table below. Note that since jal 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 funct3.

Type	Instruction	Operation	ALUop
I	lw	addition	00
S	sw	addition	00
I	addi	addition	00
SB	beq	subtraction	01
UJ	jal	don't-care	xx
R	unknown	unknown	10

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

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

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

endmodule
```

<div style="background:purple">yC4</div>

30. We now turn our attention to the fourth and last part of our control unit, **yC4**. This unit sees the funct3 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:

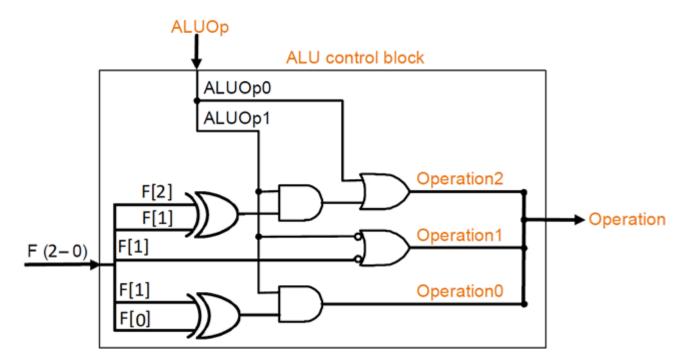| ALUop	Funct3	Instruction	Operation	op
00	*don't-care*	*don't-care*	addition	010
01	*don't-care*	*don't-care*	subtraction	110
10	111	and	conjunction	000
10	110	or	disjunction	001
10	000	add	addition	010

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.

31. Add the following module to your cpu.v file:
```
module yC4(op, ALUop, funct3);
output [2:0] op;
input [2:0] funct3;
input [1:0] ALUop;
// instantiate and connect

endmodule
```

To implement this circuit, see the diagram discussed below.

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

33. 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: `funct3` is ignored if **ALUop** is `00` or `01`.

34. Complete the development of **yC4**. Its body should have exactly eight lines since it is made up of eight primitive gates (2 **_not_**, 2 **_xor_**, 2 **_and_**, 2 **_or_** gates).

<div style="background-color:purple; color:white; text-align:center">**LabN4.v**</div>

35. Save LabN3.v as LabN4.v and modify its instantiation section as follows:

```
yIF myIF(ins, PC, PCp4, PCin, clk);
yID myID(rd1, rd2, imm, jTarget, branch, ins, wd, RegWrite, clk);
yEX myEx(exeOut, zero, rd1, rd2, imm, op, ALUSrc);
yDM myDM(memOut, exeOut, rd2, clk, MemRead, MemWrite);
yWB myWB(wb, exeOut, memOut, Mem2Reg);
yPC myPC(PCin,PC,PCp4,INT,entryPoint,branch,jTarget,zero,isbranch,isjump);
assign opCode = ins[6:0];
yC1 myC1(isStype, isRtype, isItype, isLw, isjump, isbranch, opCode);
yC2 myC2(RegWrite, ALUSrc, MemRead, MemWrite, Mem2Reg,
         isStype, isRtype, isItype, isLw, isjump, isbranch);
yC3 myC3(ALUop, isRtype, isbranch);
assign funct3=ins[14:12];
yC4 myC4(op, ALUop, funct3);
assign wd = wb;
```

36. 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 = ?; 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.

37. 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:

```
02802023: rd1= 0 rd2=36 exeOut= 32 zero=0 wb=32
02a02223: rd1= 0 rd2=15 exeOut= 36 zero=0 wb=36
```

LabN4.v

38. 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)**.

39. 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 LabN4 to the body of this module.

40. Save LabN4 as LabN5.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 = ?; 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
```

41. Compile and run LabN5. The generated output should be similar to the previous. In particular, the last two lines of the output should be:

```
02802023: rd2=36 wb=32
02a02223: 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.