LAB 6
MIPS Control Unit

A CPU is often said to be the brains of a computer. In that case the Control Unit is the brain of the CPU.
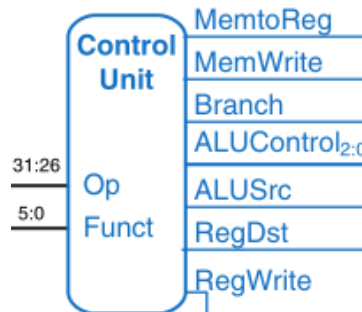

Figure 3.1 Control Unit

The Control unit's task is to decode an instruction and then tell the rest of the processor what to do. Refer to Figure 3.1: in the MIPS processor, the 6-bit Opcode field determines the instruction, therefore the Control Unit takes the Op field (bits 31:26 of the instruction) as its input. MIPS R-format of instruction is shown in Figure 3.2. If the instruction is the R-Format then the Funct field must also be deciphered. Hence, the Control Unit needs "Funct" field (bits 5:0 of the instruction) as an input too.

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Figure 3.2 MIPS R-format instruction

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 4.16   The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See 🌐 Appendix B for further discussion of this problem.)

The Control Unit (Figure 3.1) generates total of 7 output (control) signals. The values of the signals depend on the current instruction, i.e. the inputs of the Control Unit. Details of the effects of each signal, when deasserted and asserted, are given in Figure 4.6.

The Control Unit consists of two parts (Figure 3.3): Main Decoder that takes as the input Op (Opcode) field and generates 6+1 control signals, and ALU Decoder that takes in Funct field and ALUOp control signal, and generates 3-bit ALUContro control signal.
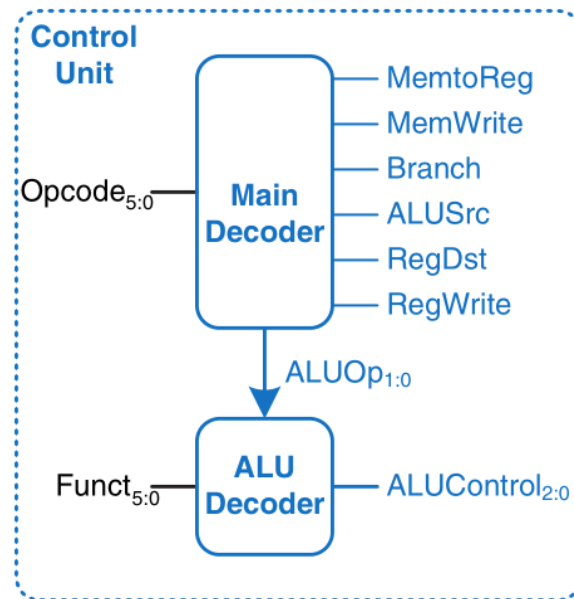

Figure 3.3 Control Unit and its two submodules

We will start by generating those two sub modules. First, create a new file in EDA Playground using the + button, name the file aludec.v:
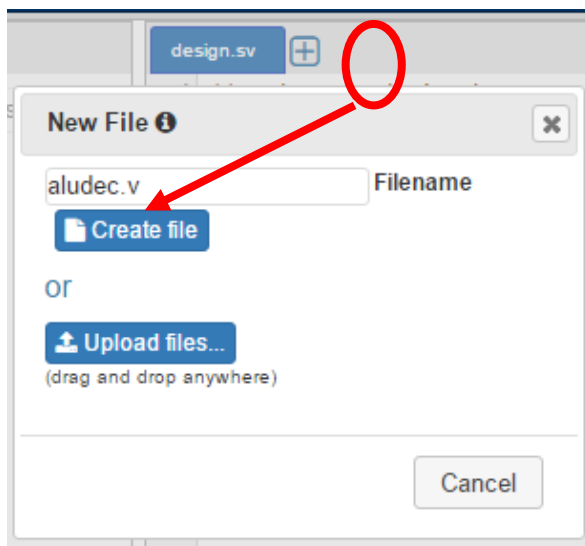

Figure 3.4 Adding aludec.v to your design 00

Use the Verilog module definition below for the ALU Decoder and name the file **aludec.v**:

```
1        `timescale 1ns / 1ps
2
3        module aludec(funct,aluop,alucontrol);
4            input       [5:0] funct;
5            input       [1:0] aluop;
6            output  [2:0] alucontrol;
7            reg     [2:0] alucontrol;
8
9            always @(funct, aluop)
10           case(aluop)
11               2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
12               2'b01: alucontrol <= 3'b110; // sub (for beq)
13               default: case(funct) // R-type instructions
14                   6'b100000: alucontrol <= 3'b010; // add
15                   6'b100010: alucontrol <= 3'b110; // sub
16                   6'b100100: alucontrol <= 3'b000; // and
17                   6'b100101: alucontrol <= 3'b001; // or
18                   6'b101010: alucontrol <= 3'b111; // slt
19                   default: alucontrol <= 3'bxxx; // ???
20               endcase
21           endcase
22       endmodule
```

Figure 3.5 Verilog code for aludec module

Next, create a new file in EDA Playground named maindec.v
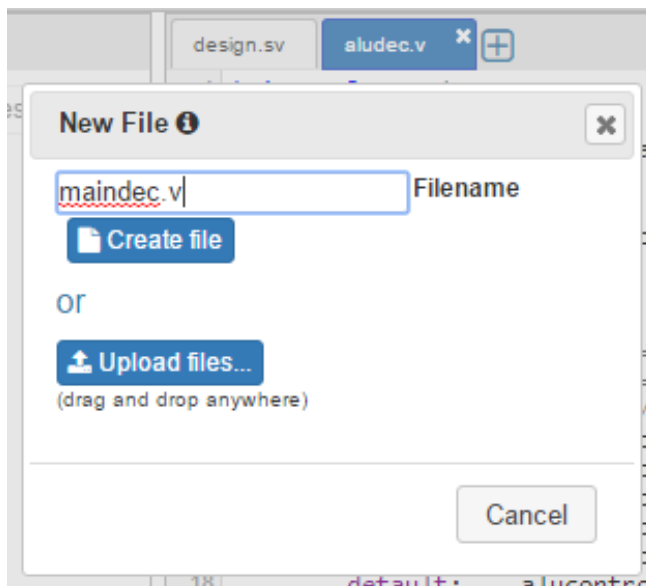

Figure 3.6 Adding maindec file to the design

Use the Verilog definition below for the main instruction decoder named **maindec.v**:

```
1       `timescale 1ns/1ps
2
3       module maindec(op,memtoreg,memwrite,branch,alusrc,regdst,regwrite,jump,aluop);
4           input       [5:0]   op;
5           output          memtoreg, memwrite;
6           output          branch, alusrc;
7           output          regdst, regwrite;
8           output          jump;
9           output  [1:0]   aluop;
10          reg     [8:0] controls;
11
12          assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop}=
            controls;
13          always @(op)
14          case(op)
15              6'b000000: controls <= 9'b110000010; // RTYPE
16              6'b100011: controls <= 9'b101001000; // LW
17              6'b101011: controls <= 9'b001010000; // SW
18              6'b000100: controls <= 9'b000100001; // BEQ
19              6'b001000: controls <= 9'b101000000; // ADDI
20              6'b000010: controls <= 9'b000000100; // J
21              default: controls <= 9'bxxxxxxxxx; // illegal op
22          endcase
23      endmodule
24
```

Figure 3.7 Verilog code for maindec module

The control unit being designed in this lab for the MIPS processor integrates the ALU Control, the main control unit (or main instruction decoder) and an additional logic gate to better consolidate control related aspects of the processor. This is depicted in the Figure 3.8.
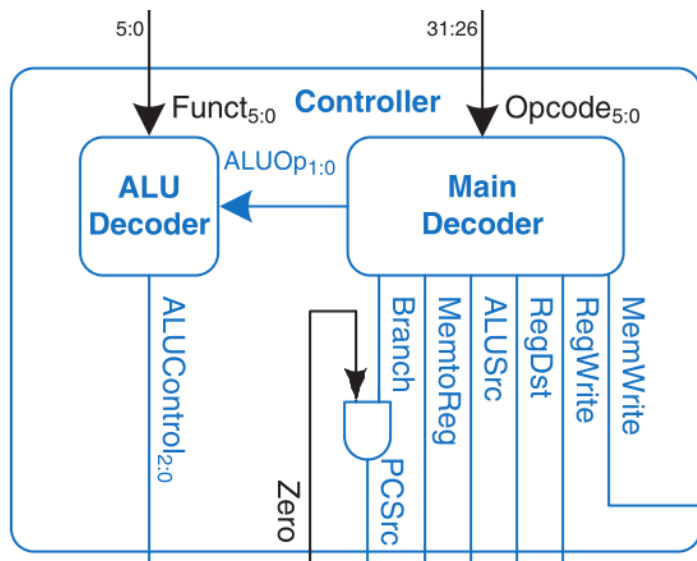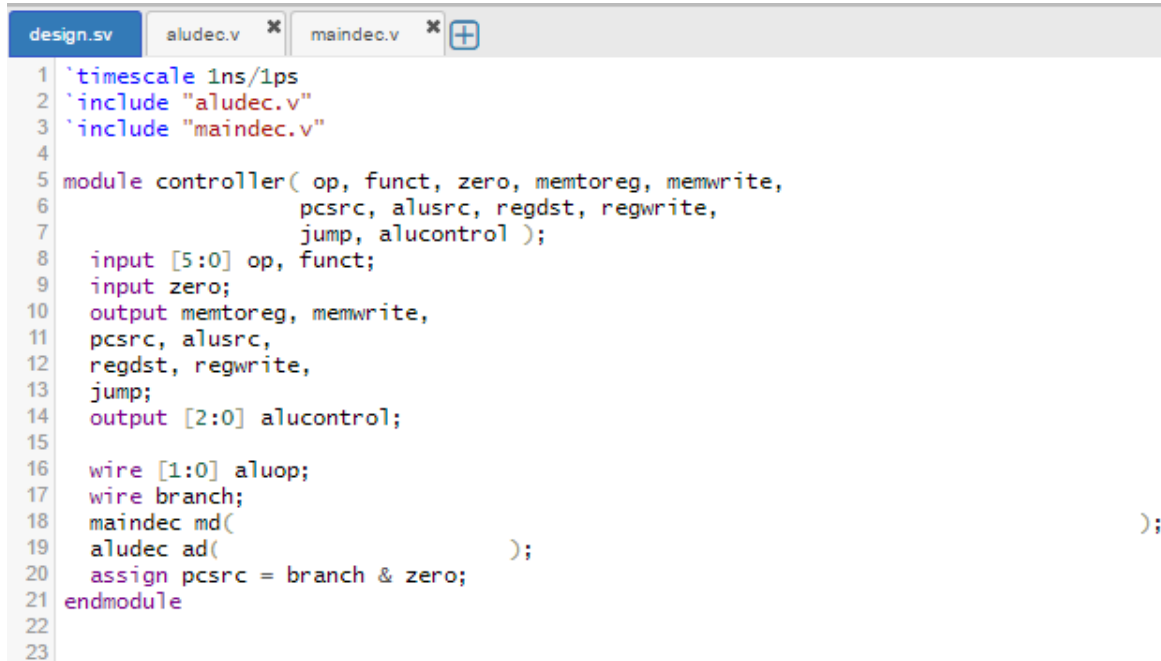

Figure 3.8 Controller is the top module integrating Main Decoder, ALU Decoder and taking in additional input "Zero" to generate control signal for selection of the next value of PC

The "top level module" for this lab will be the controller (Figure 7). By default, EDA playground looks to design.sv for the top level module. Skeleton code for the top level module is given in a file named **controller.v**. You need to complete the code in order to instantiate and connect two sub modules inside the **controller.v** file.

```
 1  `timescale 1ns/1ps
 2  `include "aludec.v"
 3  `include "maindec.v"
 4
 5  module controller( op, funct, zero, memtoreg, memwrite,
 6                     pcsrc, alusrc, regdst, regwrite,
 7                     jump, alucontrol );
 8     input [5:0] op, funct;
 9     input zero;
10     output memtoreg, memwrite,
11     pcsrc, alusrc,
12     regdst, regwrite,
13     jump;
14     output [2:0] alucontrol;
15
16     wire [1:0] aluop;
17     wire branch;
18     maindec md(                                             );
19     aludec ad(                    );
20     assign pcsrc = branch & zero;
21  endmodule
22
23
```

Figure 3.8 Verilog code (skeleton, to be populated) for the entire controller

Finally test your design to ensure it works correctly using the Verilog test fixture code shown in Figure 3.9, and verify the results using the Figure 3.10.

WHAT TO SUBMIT

Once you have verified proper functionality of your project, copy the contents of **controller module (design.sv)** to a text file named **controller.txt** and upload the BeachBoard Dropbox for Lab 6. Additionally, upload your Lab report (see the LabReportTemplete in the Documents folder on BeachBoard).

NOTE:

- Comment your code
- Keep these lab files as they will be needed for future labs!

Created by Josh Hayter, updated by Jelena Trajkovic

```systemverilog
testbench.sv  ⊞

 1  `timescale 1ns/1ps
 2  module t_controller();
 3    reg [5:0] op,funct; reg zero;
 4    wire memtoreg,memwrite,pcsrc,alusrc,regdst,regwrite,jump;
 5    wire [2:0] alucontrol;
 6
 7    controller dut(op,funct,zero,memtoreg,memwrite,
 8                   pcsrc,alusrc,regdst,regwrite,jump,alucontrol);
 9    integer i;  reg [5:0] funct_codes [4:0];
10
11    initial begin
12      funct_codes[0] = 6'b100000; //add
13      funct_codes[1] = 6'b100010; //sub
14      funct_codes[2] = 6'b100100; //and
15      funct_codes[3] = 6'b100101; //or
16      funct_codes[4] = 6'b101010; //slt
17    end
18
19    initial begin
20      $display("Testing R-Type instructions");
21      op = 0;
22      for(i = 0; i < 5; i = i + 1)
23        begin
24          funct = funct_codes[i]; #1 showsignals();
25        end
26      funct = 6'bx;    //funct field doesn't matter for remaining instructions
27      op = 6'b100011; #1 $display("lw"); showsignals();
28      op = 6'b101011; #1 $display("sw"); showsignals();
29      op = 6'b000100; zero = 1; #1 $display("beq (branch taken)"); showsignals();
30                      zero = 0; #1 $display("beq (branch NOT taken)"); showsignals();
31      op = 6'b001000; #1 $display("addi"); showsignals();
32      op = 6'b000010; #1 $display("j"); showsignals();
33      $display("end simulation!"); $finish;
34    end
35    task showsignals; begin
36      $display("%b", {memtoreg,memwrite,pcsrc,alusrc,regdst,regwrite,jump,alucontrol});
37      end endtask
38  endmodule
```

Figure 3.9 Verilog code for the testbench

```
Testing R-Type instructions
0000110010
0000110110
0000110000
0000110001
0000110111
lw
1001010010
sw
0101000010
beq (branch taken)
0010000110
beq (branch not taken)
0000000110
addi
0001010010
j
0000001010
end simulation!
Done
◀
```

Figure 3.10 Reference values for the output/results of the simulation

Created by Josh Hayter, updated by Gayathri Venna and Jelena Trajkovic