Introduction to Digital Logic Design Lab
EECS 31L

LAB 6

CECS Department
California State University, Long Beach

Spring 2020

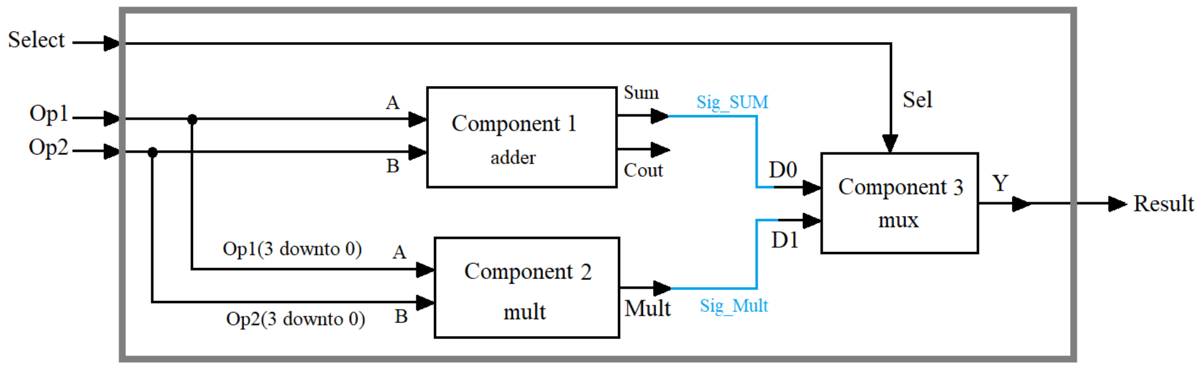
Due on Friday 04/24/2020 by 8:00 pm

Through this course, we want to design a RISC-V Single Cycle Processor. Here in this Lab, we will work on the Datapath part of the processor. Part 1 reviews some information from Verilog regarding the design hierarchies. In Part 2, we review the RISC-V datapath. In part 3 we talk about how to design the data memory and in part 4 we talk about designing the datapath (as a top module) and in part 5 we test the datapath.

1 Manage Design Hierarchies with Component Declaration

This section shows you how to use partitioning and design management to manage larger design. Hierarchy is a way of managing a design by creating references to external, lower-level design modules from within a higher-level design module. The basic unit of hierarchy in Verilog is the component. A component is a Verilog module that is referenced as a lower-level module from another, higher-level module. A Verilog design (a particular module/architecture pair) can be referenced from another architecture as a Verilog component. Instantiating components within another design provides a mechanism for partitioned design, or for using existing designs as reusable components of larger designs.

Figure 1 : Hierarchy and Components.



2 Datapath

Figure 2 : RISC-V Datapath.

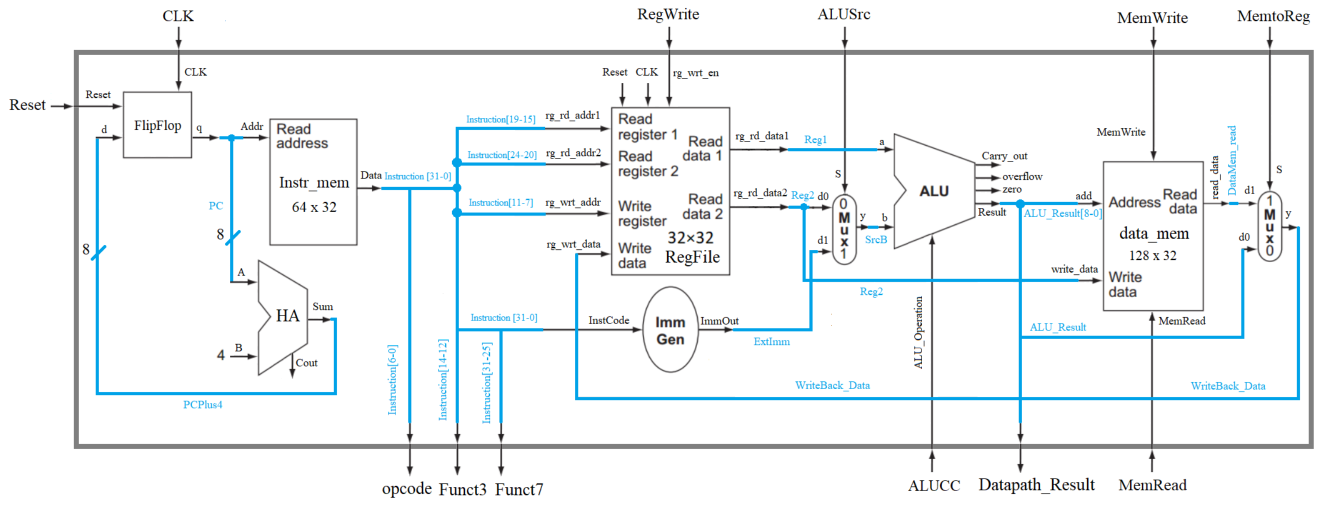


Figure 2 shows the datapath of a RISC-V single cycle processor. The instruction execution starts by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file. The blue lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

Some of the inputs (RegWrite, ALUSrc, ALUCC, MemRead, MemWrite, MemtoReg) are control signals which are derived by a module named "Control". The control unit is supposed to be designed later and here in this lab you assume you have all the control signals as inputs.

Table 1 shows the list of Instructions that our Datapath supports.

Table 1 : Instruction Set.

imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	100	rd	0010011	NORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	100	rd	0110011	NOR
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Note: along with the provided instructions in Table 1, your datapath needs to support `"lw"` and `"sw"` instructions too. Table 2 and 3 shows format of these two data-transfer instructions.

Table 2 : Instruction Set (lw).

imm[11:0]	rs1	010	rd	0000011	LW

Table 3 : Instruction Set (sw).

imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

For this part, you need to save these two instructions into the "Instruction memory" that you designed in previous lab. The following two commands will write these two instructions into the instruction memory:

memory[18] = 32'h02b02823; // sw r11, 48(r0) alu_result = 32'h00000030
memory[19] = 32'h03002603; // lw r12, 48(r0) alu_result = 32'h00000030 r12 = 32'h00000005

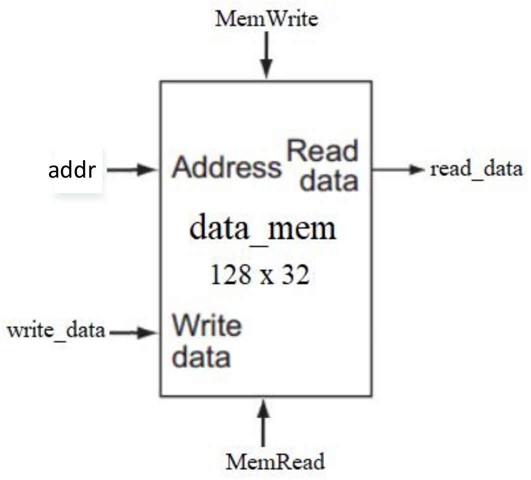
3 Lower Level Modules

As it is shown in Figure 2, there is a top module (Datapath) and nine lower-level modules (FlipFlop, HA, Instr_mem, RegFile, Imm_Gen, Mux (two instantiations), ALU, data_mem). You have already designed four lower-modules such as FlipFlop, Inst_mem,RegFile, and ALU. As it was mentioned in Lab 5, HA module can be easily implemented in Verliog using "+" operation (simply adding PC with value 4: PC_Next = PC + 4). Imm_Gen and Mux designs also provided for you in Lab 5.
In this lab, we start by designing the last sub-module which is Data_mem.

Note: 32-bit ALU design is also provided for you in section 3.3. You are welcome to use your own ALU design , if your design got the full credit.

3.1 Data Memory

Same as the Instruction memory (refer to the previous lab), the data memory in our processor is byte addressable. We can store 128 data each with 32 bits (128 x 32). To address 128 x 4 = 512 bytes 9 bits are required for address line. These 9-bits come from the 9 LSBs of the output of the ALU (ALU_Result). To read a data we need an address and the read enable signal (MemRead). To write a data we need an address, the write enable signal (MemWrite), and a data to write (Write_data).



Note: Use the provided module definition to design your Data Memory. Otherwise, your submission will not be considered for grading.

Code 1: Data Memory

```
 `timescale 1ns / 1ps
// Module definition
module DataMem(MemRead, MemWrite, addr, write_data, read_data);
//Define I/O ports


// Describe data_mem behaviour


endmodule   // data_mem
```

3.1.1 Result Mux

The MUX on the output of the Data Memory will decide whether the writing data (to the register file) should come from the ALU or come from the Data Memory (refer to Figure 2). You can use 2-to-1 Mux design from Lab 5.

3.2 Immediate Generator

Use the code provided in Lab 5.

3.3 32-bit ALU

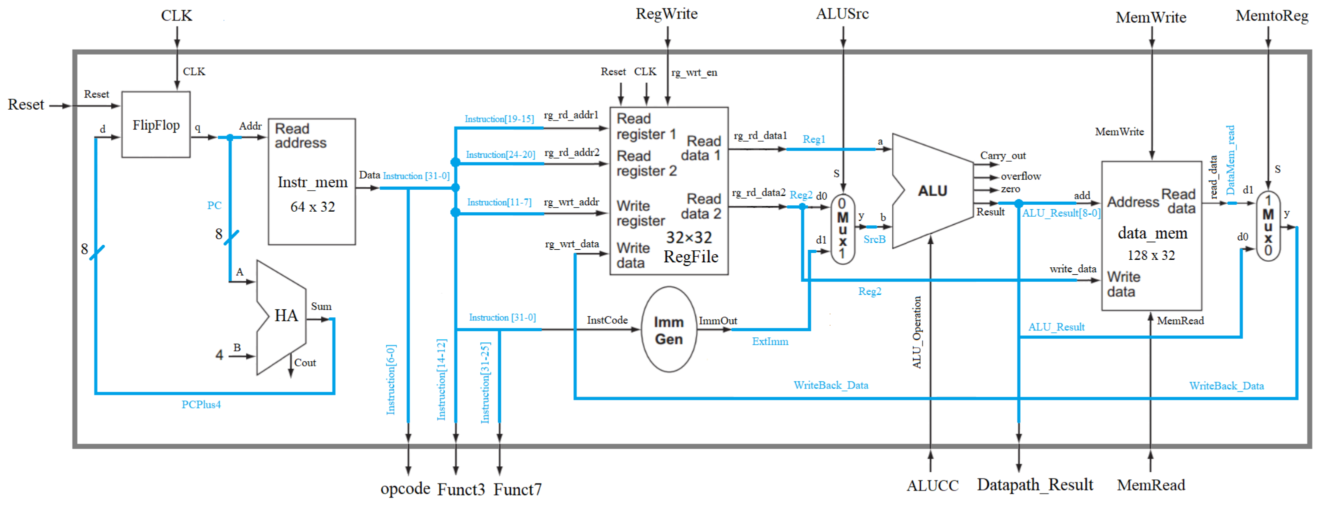
Code 2: 32-bit ALU

```
module alu_32(
   input  [31:0] A_in,B_in,        // ALU 32 bit inputs
   input  [3:0]  ALU_Sel,          // ALU 4 bits selection
   output [31:0] ALU_Out,          // ALU 32 bits output
   output reg    Carry_Out,
   output        Zero,             // 1 bit Zero Flag
   output reg    Overflow = 1'b0   // 1 bit Overflow flag
    );
  reg  [31:0] ALU_Result;
  reg [32:0] temp;
  reg [32:0] twos_com; // to hold 2'sc of second source of ALU

  assign ALU_Out   = ALU_Result;              // ALU Out
  assign Zero      = (ALU_Result == 0);      // Zero Flag

  always @(*)
    begin
     Overflow = 1'b0;
     Carry_Out = 1'b0;
     case(ALU_Sel)
        4'b0000: //  and
          ALU_Result = A_in & B_in;

        4'b0001: //  or
          ALU_Result = A_in | B_in;

        4'b0010: // Signed Addition with Overflow and Carry_out checking
           begin
             ALU_Result = $signed(A_in) + $signed(B_in);
             temp = {1'b0, A_in} + {1'b0, B_in};
             Carry_Out = temp[32];
```

```
              if ((A_in[31] & B_in[31] & ~ALU_Out[31]) |
              (~A_in[31] & ~B_in[31] & ALU_Out[31]))
                  Overflow = 1'b1;
              else
                  Overflow = 1'b0;
            end

         4'b0110: // Signed Subtraction with Overflow checking
            begin
              ALU_Result = $signed(A_in) - $signed(B_in) ;
              twos_com = ~(B_in) + 1'b1;
              if ((A_in[31] & twos_com[31] & ~ALU_Out[31]) |
              (~A_in[31] & ~twos_com[31] & ALU_Out[31]))
                  Overflow = 1'b1;
              else
                  Overflow = 1'b0;
            end

          4'b0111: // Signed less than comparison
            ALU_Result = ($signed(A_in) < $signed(B_in))?32'd1:32'd0;

          4'b1100: //   nor
            ALU_Result = ~(A_in | B_in);

          4'b1111: // Equal comparison
            ALU_Result = (A_in==B_in)?32'd1:32'd0 ;

          default: ALU_Result = A_in + B_in ;
        endcase
      end

endmodule
```

4 Higher Level Module

Now that we have designed all of the submodules, we can use them as a component and design the Datapath. Here again, you see the Datapath. Blue lines are some wires which we used to connect the submodules. Define these blue lines as ”wire” and connect the components to complete the Datapath.
Note: In data memory design (which is a top module), you need to instantiate from each sub module and make wire connections as it is shown in the figure bellow.



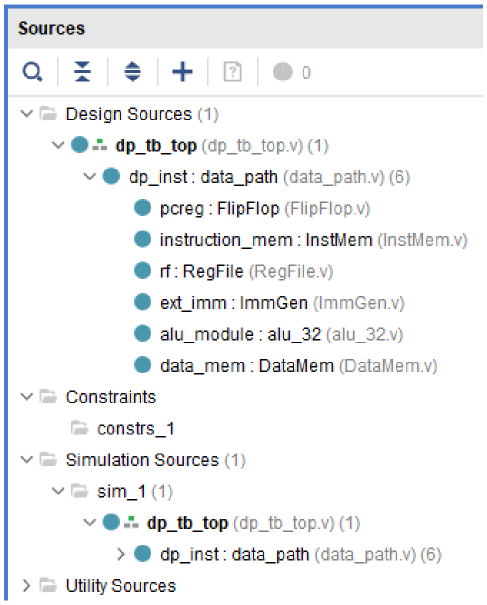
Note: For Datapath code, we used lowercase letters for input/output naming. You need to use the exact code samples provided for you to design the Datapath and tb_Datapath. Otherwise, your submission will not be considered for grading.
Use the following code for the module definition of your Datapath.

Code 3: Datapath

```
module data_path #(
    parameter PC_W = 8,        // Program Counter
    parameter INS_W = 32,      // Instruction Width
    parameter RF_ADDRESS = 5, // Register File Address
    parameter DATA_W = 32,     // Data WriteData
    parameter DM_ADDRESS = 9, // Data Memory Address
    parameter ALU_CC_W = 4     // ALU Control Code Width
 )(
    input                   clk ,        // CLK in Datapath figure
    input                   reset,       // Reset in Datapath figure
    input                   reg_write,   // RegWrite in Datapath figure
    input                   mem2reg,    // MemtoReg in Datapath figure
    input                   alu_src,    // ALUSrc in Datapath figure
    input                   mem_write,   // MemWrite in Datapath Figure
    input                   mem_read,    // MemRead in Datapath Figure
    input   [ALU_CC_W-1:0]  alu_cc,      // ALUCC in Datapath Figure
    output           [6:0]  opcode,     // opcode in Datapath Figure
    output           [6:0]  funct7,     // Funct7 in Datapath Figure
    output           [2:0]  funct3,    // Func3 in Datapath Figure
    output    [DATA_W-1:0]  alu_result // Datapath_Result in Datapath Figure
 );

// Write your code here


endmodule   // Datapath
```

Important: we want you to have separate source files for each of the datapath submodules. As you can see in the picture bellow, in the "Design Sources" section, all the submodules are included along with data_path design (data_path is the top module and other modules listed bellow it are the submodules in the design).
Note: In this picture, you don't see any submodule design for MUX, since MUX is designed with another method using Verilog statements. You can use the provided sample code for MUX (in Lab 5),

and in this case, you also need to include MUX source code in ”Design Sources” section.



```
Design Sources (1)
  dp_tb_top (dp_tb_top.v) (1)
    dp_inst : data_path (data_path.v) (6)
      pcreg : FlipFlop (FlipFlop.v)
      instruction_mem : InstMem (InstMem.v)
      rf : RegFile (RegFile.v)
      ext_imm : ImmGen (ImmGen.v)
      alu_module : alu_32 (alu_32.v)
      data_mem : DataMem (DataMem.v)
Constraints
    constrs_1
Simulation Sources (1)
  sim_1 (1)
    dp_tb_top (dp_tb_top.v) (1)
      dp_inst : data_path (data_path.v) (6)
Utility Sources
```

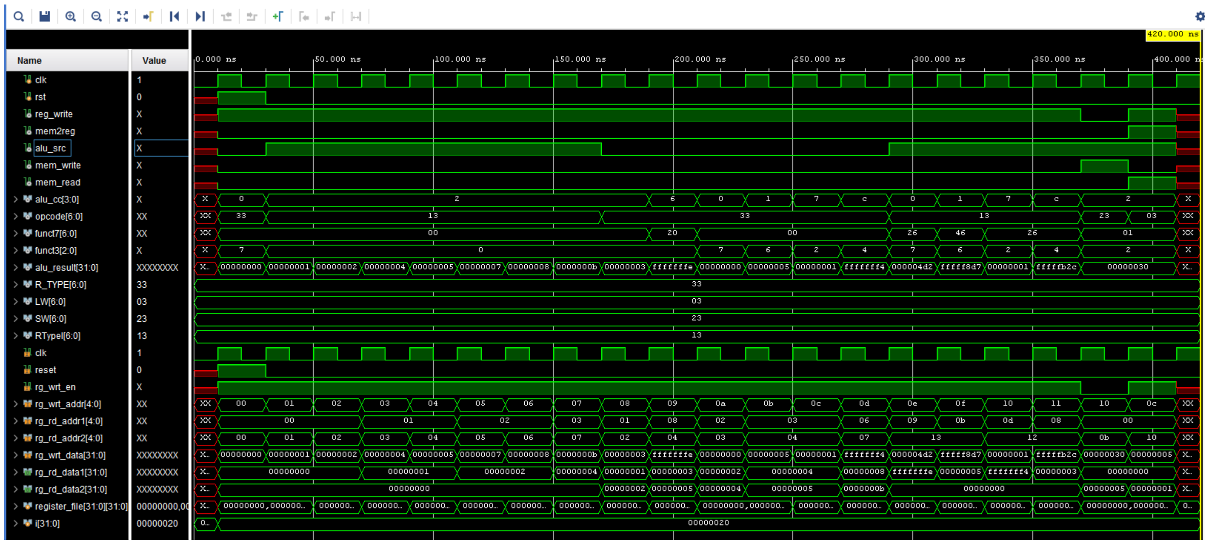
5 Test the Datapath

Use the provided test-bench to test your Datapath design.

Code 4: tb_Datapath

```
module dp_tb_top();

  /** Clock & reset **/
  reg clk, rst;
  always begin
    #10;
    clk = ~clk;
  end

  initial begin
    clk = 0;
    @(posedge clk);
    rst = 1;
    @(posedge clk);
    rst = 0;
  end

  /** DUT Instantiation **/
  wire          reg_write  ;
  wire          mem2reg    ;
  wire          alu_src    ;
  wire          mem_write  ;
  wire          mem_read   ;
  wire   [3:0]  alu_cc     ;
  wire   [6:0]  opcode     ;
  wire   [6:0]  funct7     ;
  wire   [2:0]  funct3     ;
  wire  [31:0]  alu_result ;

  data_path dp_inst(
    .clk        ( clk        ),
    .reset      ( rst        ),
    .reg_write  ( reg_write  ),
    .mem2reg    ( mem2reg    ),
    .alu_src    ( alu_src    ),
    .mem_write  ( mem_write  ),
    .mem_read   ( mem_read   ),
    .alu_cc     ( alu_cc     ),
    .opcode     ( opcode     ),
    .funct7     ( funct7     ),
    .funct3     ( funct3     ),
    .alu_result ( alu_result )
  );

  /** Stimulus **/
  wire [6:0] R_TYPE, LW, SW, RTypeI;

  assign  R_TYPE = 7'b0110011;
  assign  LW     = 7'b0000011;
  assign  SW     = 7'b0100011;
  assign  RTypeI = 7'b0010011;


  assign alu_src   = (opcode==LW || opcode==SW || opcode == RTypeI);
  assign mem2reg   = (opcode==LW);
  assign reg_write = (opcode==R_TYPE || opcode==LW || opcode == RTypeI);
  assign mem_read  = (opcode==LW);
  assign mem_write = (opcode==SW);

  assign alu_cc = ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0000000) && (funct3 == 3'b000)) ? 4'b0010 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0100000)) ? 4'b0110 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0000000) && (funct3 == 3'b100)) ? 4'b1100 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0000000) && (funct3 == 3'b110)) ? 4'b0001 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0000000) && (funct3 == 3'b111)) ? 4'b0000 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct7 == 7'b0000000) && (funct3 == 3'b010)) ? 4'b0111 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct3 == 3'b100)) ? 4'b1100 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct3 == 3'b110)) ? 4'b0001 :
                  ((opcode==R_TYPE || opcode == RTypeI)
                  && (funct3 == 3'b010)) ? 4'b0111 :
                  ((opcode==LW || opcode == SW)
                  && (funct3 == 3'b010))? 4'b0010 : 0;

  initial begin
    #420;
    $finish;
  end

endmodule
```

Check the outputs (opcode, funct3, funct7, alu_result) to see if they are correct. Put a screenshot of the wave in your report.
Here you can see screenshot of the waveform for the datapath design:



6 Assignment Deliverable

Your submission should include the following:

- Block designs and testbenches. (FlipFlop.v, Instr_mem.v, RegFile.v, Imm_Gen.v, Mux.v, ALU.v, Data_mem.v, Datapath.v. tb_Datapath.v)
- If you designed a HA module to perform PC + 4 operation, you need to include it in your submission files. Otherwise (in case of using simple "PC_Next = PC + 4"), no need to submit any design code for HA module.
- A report in **pdf** format. Your report should have all details for your design + screenshot of the wave. Each report should include group members if your are working as a group.

Note1: Compress all files (.v files + report) into zip and upload to the **Beachboard Dropbox** before deadline.
Note2: Use the code samples that are given in the lab description. **The module part of your code should exactly look like the code sample otherwise your submission will not be considered for grading.**.