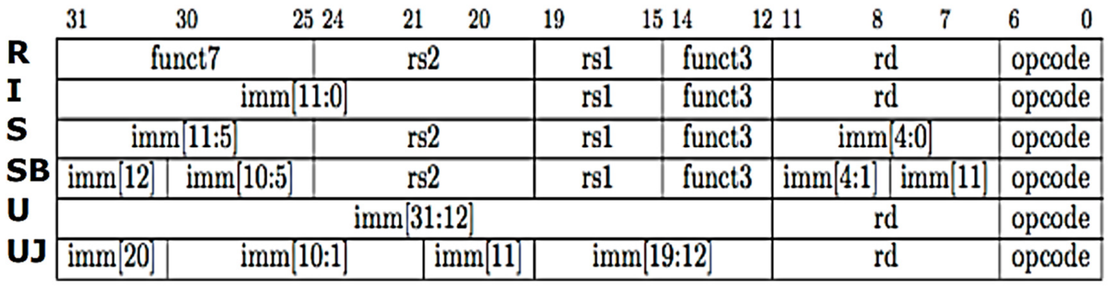
RISC CPU Design Description

Author: Yang Zhao

In this project, we are going to build a 32-bit RISC-V CPU. Due to the limit of time, we just implement 11 instructions that are LW, SLLI, SW, BEQ, ADD, SUB, SLL, XOR, OR, JAL, HALT. Among them, LW, SLLI and HALT belong to family I, SW, BEQ, JAL and others are belong to family S, SB, UJ, and R. There instruction format and are given in Fig. 1.



	31	30 25	24 21	20	19 15	14 12	11 8	7	6 0
R	funct7		rs2		rs1	funct3	rd		opcode
I	imm[11:0]				rs1	funct3	rd		opcode
S	imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode
SB	imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode
U	imm[31:12]						rd		opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode

Fig. 1. 32-bit Instruction family format

The binary codes for those instructions are provided in Table I.

Table I. instruction code and the usage

Instruction	Binary Codes	Meaning/usage
LW	32'b?????????????????010?????0000011	lw rd, rs1, imm
SLLI	32'b000000???????????001?????0010011	slli rd, rs1, shamt
SW	32'b?????????????????010?????0100011	sw rs1, rs2, imm
BEQ	32'b?????????????????000?????1100011	beq rs1, rs2, imm
ADD	32'b0000000??????????000?????0110011	add rd, rs1, rs2
SUB	32'b0100000??????????000?????0110011	sub rd, rs1, rs2
SLL	32'b0000000??????????001?????0110011	Logical left shift, sll rd, rs1, rs2
XOR	32'b0000000??????????100?????0110011	xor rd, rs1, rs2
OR	32'b0000000??????????110?????0110011	or rd, rs1, rs2
JAL	32'b?????????????????????????1101111	jal rd, imm: jump and link
HALT	32'b00000000000100000000000001110011	System break, here we take is as halt

*? Stands for the bit can be any value, which doesn't affect the function.

To make it understandable, I divide the CPU into 7 sub modules with 5 stages as shown in Fig. 2. The **control** module, we can take it as a supervisor which collects the command information from the instruction decoder, and intermediate results that need multiplexing (select via MUX, for some modules' input, there are several possible candidates, all possible candidates sent to the control module to decide which one to be sent out.). Based on the collected information and intermediate results from other modules, then the **control** configure the control bits for each module first, and then trigger the execution of the module by the rising edge of the trigger signal.

For example, the **PC**(program counter) is triggered by the signal *inc_pc* from the **control,** before triggering or updating of the **PC** (change *inc_pc* from 0 to 1 in **control**)**,** the control bit and necessary input to **PC** should be ready first. That is, **control** will set the *jump* and *branch* indicator of the **PC** to the specific value before the stage that changes *inc_pc*. In this example, *jump* and *branch* is the control bit for **PC** and *inc_pc* is the trigger signal for **PC.**

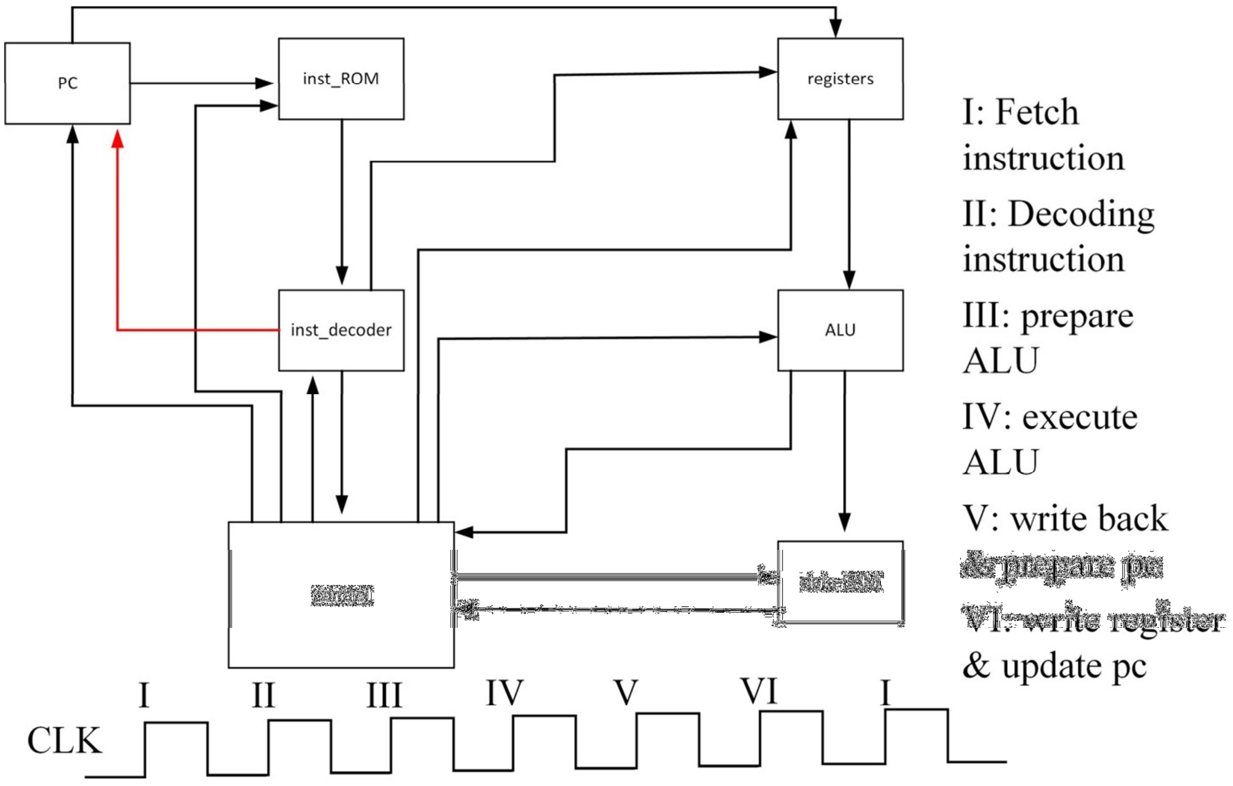


Fig. 2. Sub-modules and the 5 stages we defined in our project.

Generally, the execution of a instruction can be divided into instruction fetch, instruction interpretation, instruction execution, and write-back 4 stages. It seems like 4 clock cycles are enough for one instruction. However, this is just the theoretical situation. We require extra cycles for preparations before the execution. Considering the instruction in this project is greatly simplified, and thus one more clock cycles is sufficient to handle the whole procedures, i.e., 5 clock cycles or 5 stages as in Fig. 2.

The operations of each stage as descripted as follows. Before we start, the initial condition should be clarified. Just as we start a computer or phone, the CPU each time starting up will be reset to the predefined initial state. In this project, after resetting, program counter is set the zero and *state* in the **control** module is set to stage I. So after the resetting, the first clock positive edge triggers the operation of stage I in **control**. In stage I, **control** set the *load_inst* from 0 to 1, which enables the reading of the read only memory that stores your instructions (**inst_ROM**). Once enabled by *load_inst,* **inst_ROM** put the value at the address indexed by the current program counter value (the first time, it is 0), to its output port *inst*. Assuming the instruction reading can be done within one clock cycle(it is good enough, since the reading of the memory is around 1ns or less nowadays, upon this, as long as the clock is slower than 1GHz, the time is sufficient), i.e., instruction to the input of the instruction decoder is ready at the end of the first clock cycle. Second clock edge comes, **control** will set the *dec_en* (*load_inst* should keep at 1) from 0 to 1, which triggers the execution of the **inst_decoder**. Assuming the time for decoding is also within one clock cycle, i.e., decoding is done at the end of second clock edge. Depends on the input instruction binary, the **inst_decoder** produces different outputs. For instance, BEQ, the **inst_decoder** generates the *command* (corresponding to SUB in ALU) to be read by **control,** and *branch_address* sent to **PC**, and the *rs1* and *rs2* for indexing **register**. Once *rs1* and *rs2* changes, **register** reading data output-port *rd1* and *rd2* updates with the corresponding register value with specific delay (assume the delay is within one clock cycle). Because of the **register** updating delay or multiplexing for other instructions (i.e., LW, SLLI, and SW, the second input of **ALU** is from imm part of the instruction which is provide by the *ALU_data2* of the **inst_decoder**), so the next clock cycle (stage III), it is used for the preparing of the **ALU** input (in the provided code, it is called control state). Simultaneously, **control** reads *ALU_data2* and *rd2* respectively from **inst_decoder** and **register**, and, in the meantime, it decides which one to send to **ALU's** second input port according to the command type by an internal MUX. Also, the **control** configures the operation type control bit (*ALUcommand*) of the **ALU** according to the command type (i.e., SUB for BEQ, SL for SLLI, ADD for ADD). So, at the end of third clock cycle (stage III), all the input of the **ALU** is ready. When the forth clock rising edge comes, **control** set the *ALUenable* from 0 to 1, to trigger the execution of **ALU** according to the configuration in clock cycle III. Assume, the operation can be finished within one clock cycle, then at the end of the clock IV, output of **ALU** (*ALUresult*) is ready for reading by other modules. When rising edge of clock V comes, **control** gets into write-back stage for the preparation of register or data memory writing and **PC** updating. For R family instructions, it select *ALUresult* as the data to be written to **register**; for LW, it select the output of the **data_RAM** and triggers the reading of the **data_RAM** by setting *mem_rd* from 0 to 1; for JAL, it select the PC+1(*pc_addr_plus*) and set *jump* to 1 to tell the **PC** that this updating is for jump; for BEQ, it doesn't conduct selection since it requires no **register** writing, but it set *branch* to 1 when the *ALUzero* is 1; for SW, no **register** writing but needs **data_RAM** writing. At the end of clock V, the input data and control signals for **register** and **PC** are ready. Different from **register** writing, write to **data_RAM** for instruction SW can be triggered in this cycle since the inputs are ready (address is *ALUresult* from **ALU** and value is *rd2* from **register**, which are respectively available at the end of clock IV and III). In the last clock cycle, **control** triggers the **register** writing and **PC** updating by setting *regwrite* and *inc_pc* from 0 to1, respectively. Then *state* return to instruction fetch stage again.

Practically, some of the sub modules, such as **ALU**, can be purely combinational logic. In this project, for the easy understanding, we set it as a sequential module triggered by the signal from **control**. This is also useful in the reduction of useless **ALU** operation for the save of power.

Followings are the port definition of each module and the reason behind them.

The JAL will update the **PC** (program counter value) point to target address indicates by imm, and store PC+1 to *rd* register. While, BEQ updates **PC** to the target address only if the value *rs1* is equal to *rs2*, otherwise updates the **PC** with PC+1, and the comparison is achieved by the SUB operation of the ALU by examining ZERO port of the ALU (ZERO will be set to 1 once the SUB results is 0). Other instructions always update the PC by PC+1. That is, the output of PC can be PC+1, *branch_addr* , or *jump_addr*. We need also output PC+1 as one of possible data writing input of the **register** for storing. The input and output definition for **PC** is summarized in Table II.

Table II. Port or pin definitions of module **PC**

Port name	Property	To/From	Description
rst	input	System	Power-on Reset
inc_pc	input	**control**	Trigger the **PC**
jump	input	**control**	Jump control bit
branch	input	**control**	Branch control bit
jump_addr	input	**inst_decoder**	Jump address
branch_addr	input	**inst_decoder**	Branch address
pc_addr	output	**register**	PC value indexing register file
pc_addr_plus	output	**control**	Stored value for JAL instruction

An example code of **PC** is given in Fig. 3.

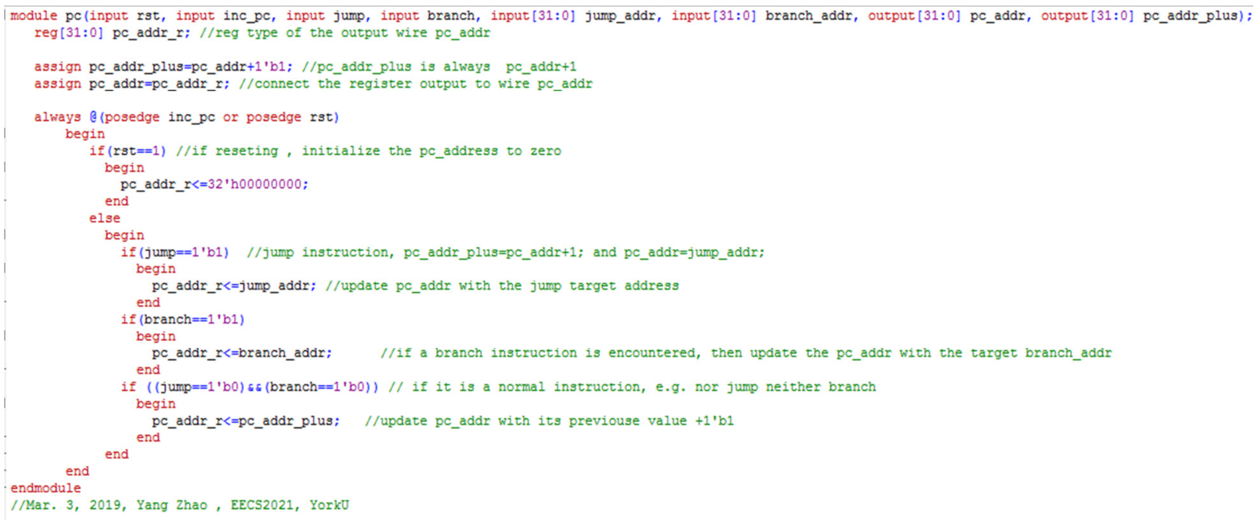

```
module pc(input rst, input inc_pc, input jump, input branch, input[31:0] jump_addr, input[31:0] branch_addr, output[31:0] pc_addr, output[31:0] pc_addr_plus);
   reg[31:0] pc_addr_r; //reg type of the output wire pc_addr

   assign pc_addr_plus=pc_addr+1'b1; //pc_addr_plus is always  pc_addr+1
   assign pc_addr=pc_addr_r; //connect the register output to wire pc_addr

   always @(posedge inc_pc or posedge rst)
       begin
          if(rst==1) //if reseting , initialize the pc_address to zero
            begin
              pc_addr_r<=32'h00000000;
            end
          else
            begin
              if(jump==1'b1)  //jump instruction, pc_addr_plus=pc_addr+1; and pc_addr=jump_addr;
                begin
                  pc_addr_r<=jump_addr; //update pc_addr with the jump target address
                end
              if(branch==1'b1)
                begin
                  pc_addr_r<=branch_addr;      //if a branch instruction is encountered, then update the pc_addr with the target branch_addr
                end
              if ((jump==1'b0)&&(branch==1'b0)) // if it is a normal instruction, e.g. nor jump neither branch
                begin
                  pc_addr_r<=pc_addr_plus;   //update pc_addr with its previouse value +1'b1
                end
            end
       end
endmodule
//Mar. 3, 2019, Yang Zhao , EECS2021, YorkU
```

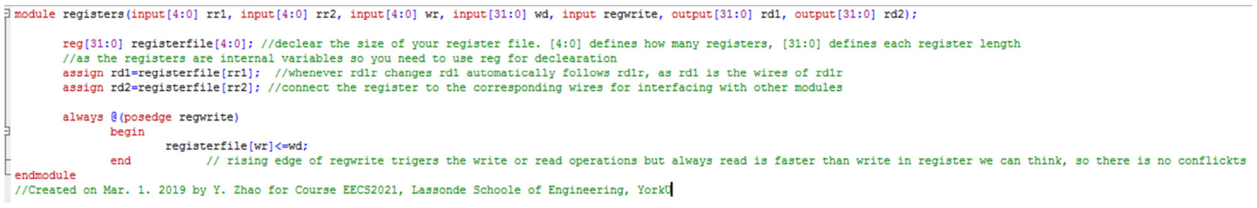
Fig. 3. Example of **PC** in verilog.

Register is triggered by the *regwrite* for writing at address *wr* with value *wd*, and the output of register *rd1* and *rd2* are always reflects the content of the register *rr1* and *rr2*. The ports are summarized in Table III.

Table III. Ports definition of module **register**

Port name	Property	To/From	Description
rr1	input	**inst_decoder**	Read register 1 index
rr2	input	**inst_decoder**	Read register 2 index
wr	input	**inst_decoder**	Write register index
wd	input	**control**	Write value
regwrite	input	**control**	Write trigger signal
rd1	output	**ALU**	Content in Read register 1
rd2	output	**control**	Content in Read register 2

Code example is provided in Fig. 4.



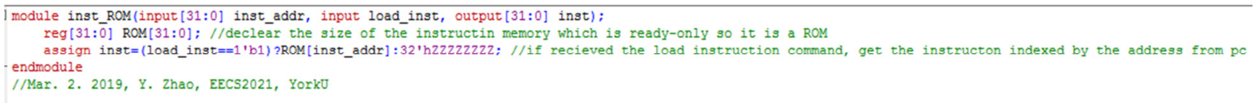
```
module registers(input[4:0] rr1, input[4:0] rr2, input[4:0] wr, input[31:0] wd, input regwrite, output[31:0] rd1, output[31:0] rd2);

       reg[31:0] registerfile[4:0]; //declear the size of your register file. [4:0] defines how many registers, [31:0] defines each register length
       //as the registers are internal variables so you need to use reg for declearation
       assign rd1=registerfile[rr1];  //whenever rd1r changes rd1 automatically follows rd1r, as rd1 is the wires of rd1r
       assign rd2=registerfile[rr2]; //connect the register to the corresponding wires for interfacing with other modules

       always @(posedge regwrite)
              begin
                     registerfile[wr]<=wd;
              end          // rising edge of regwrite trigers the write or read operations but always read is faster than write in register we can think, so there is no conflickts
endmodule
//Created on Mar. 1. 2019 by Y. Zhao for Course EECS2021, Lassonde Schoole of Engineering, YorkU
```

Fig. 4. Example of **register** in verilog.

For the memory to store instructions, it is read only memory. So we call it **inst_ROM.** It involves instruction address, the instruction load and output instruction. The ports definition as in Table IV and the code is in Fig. 5.

Table IV. Ports definition of module **inst_ROM**

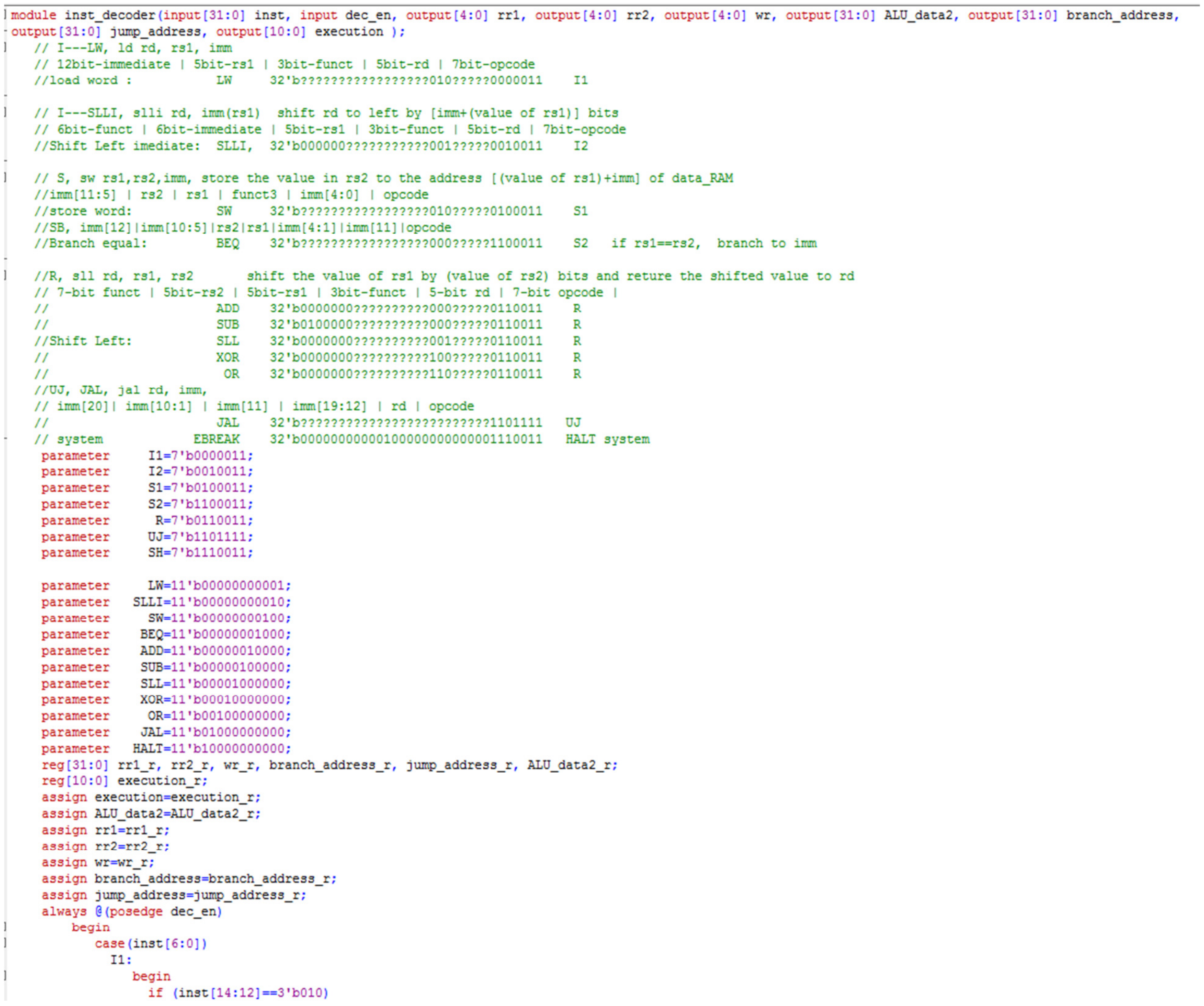
Port name	Property	To/From	Description
inst_addr	input	**PC**	Instruction address always from pc
load_addr	input	**control**	Instruction load enable signal from **control**
inst	output	**inst_decoder**	The instruction sent to **inst_decoder**



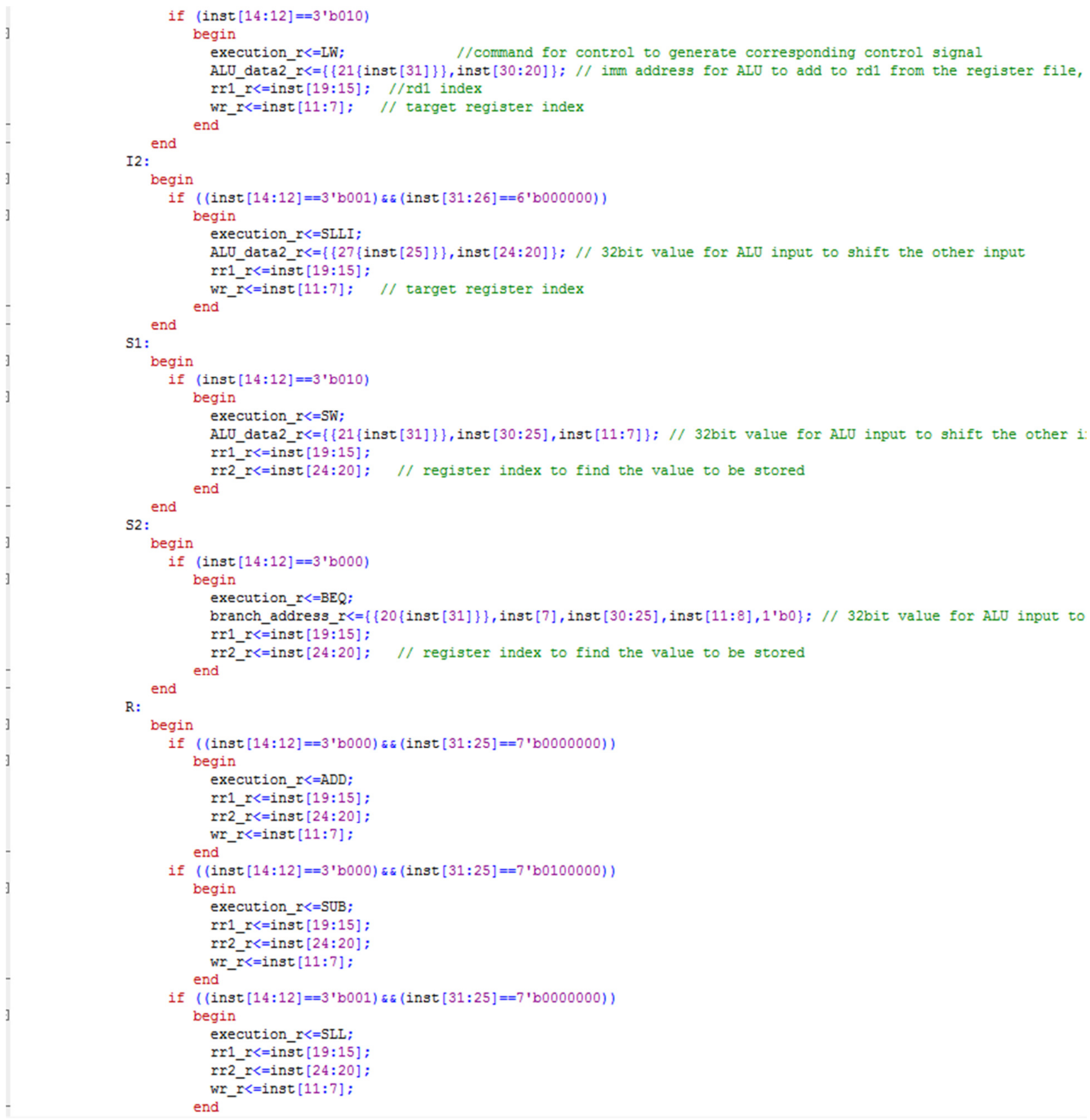
```
module inst_ROM(input[31:0] inst_addr, input load_inst, output[31:0] inst);
    reg[31:0] ROM[31:0]; //declear the size of the instructin memory which is ready-only so it is a ROM
    assign inst=(load_inst==1'b1)?ROM[inst_addr]:32'hZZZZZZZZ; //if recieved the load instruction command, get the instructon indexed by the address from pc
endmodule
//Mar. 2. 2019, Y. Zhao, EECS2021, YorkU
```

Fig. 5. Example of **inst_ROM** in verilog.

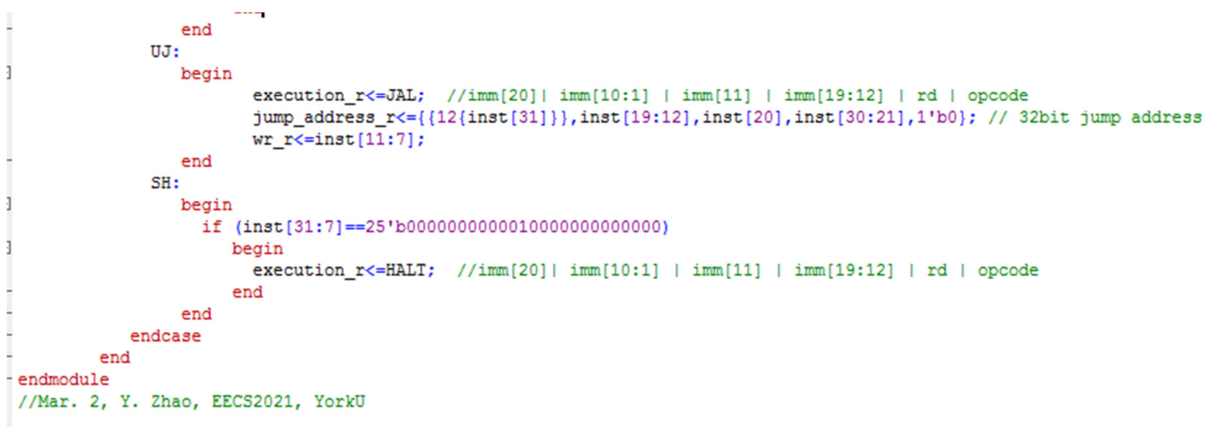
The **inst_decoder** gets the target pc in JAL and possible target pc in BEQ directly from the instruction and also interprets the execution to the format defined in **control** that maps the execution to corresponding arithmetic operations in **ALU**. The definition for each port is given in Table V, and the code example is given in Fig. 6.



```
module inst_decoder(input[31:0] inst, input dec_en, output[4:0] rr1, output[4:0] rr2, output[4:0] wr, output[31:0] ALU_data2, output[31:0] branch_address,
output[31:0] jump_address, output[10:0] execution );
   // I---LW, ld rd, rs1, imm
   // 12bit-immediate | 5bit-rs1 | 3bit-funct | 5bit-rd | 7bit-opcode
   //load word :           LW     32'b?????????????????010?????0000011    I1

   // I---SLLI, slli rd, imm(rs1)  shift rd to left by [imm+(value of rs1)] bits
   // 6bit-funct | 6bit-immediate | 5bit-rs1 | 3bit-funct | 5bit-rd | 7bit-opcode
   //Shift Left imediate:  SLLI,  32'b000000???????????001?????0010011    I2

   // S, sw rs1,rs2,imm, store the value in rs2 to the address [(value of rs1)+imm] of data_RAM
   //imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode
   //store word:           SW     32'b?????????????????010?????0100011    S1
   //SB, imm[12]|imm[10:5]|rs2|rs1|imm[4:1]|imm[11]|opcode
   //Branch equal:         BEQ    32'b?????????????????000?????1100011    S2   if rs1==rs2,  branch to imm

   //R, sll rd, rs1, rs2       shift the value of rs1 by (value of rs2) bits and reture the shifted value to rd
   // 7-bit funct | 5bit-rs2 | 5bit-rs1 | 3bit-funct | 5-bit rd | 7-bit opcode |
   //                      ADD    32'b0000000??????????000?????0110011    R
   //                      SUB    32'b0100000??????????000?????0110011    R
   //Shift Left:           SLL    32'b0000000??????????001?????0110011    R
   //                      XOR    32'b0000000??????????100?????0110011    R
   //                       OR    32'b0000000??????????110?????0110011    R
   //UJ, JAL, jal rd, imm,
   // imm[20]| imm[10:1] | imm[11] | imm[19:12] | rd | opcode
   //                      JAL    32'b?????????????????????????1101111   UJ
   // system            EBREAK    32'b00000000000100000000000001110011   HALT system
    parameter     I1=7'b0000011;
    parameter     I2=7'b0010011;
    parameter     S1=7'b0100011;
    parameter     S2=7'b1100011;
    parameter      R=7'b0110011;
    parameter     UJ=7'b1101111;
    parameter     SH=7'b1110011;

    parameter     LW=11'b00000000001;
    parameter   SLLI=11'b00000000010;
    parameter     SW=11'b00000000100;
    parameter    BEQ=11'b00000001000;
    parameter    ADD=11'b00000010000;
    parameter    SUB=11'b00000100000;
    parameter    SLL=11'b00001000000;
    parameter    XOR=11'b00010000000;
    parameter     OR=11'b00100000000;
    parameter    JAL=11'b01000000000;
    parameter   HALT=11'b10000000000;
    reg[31:0] rr1_r, rr2_r, wr_r, branch_address_r, jump_address_r, ALU_data2_r;
    reg[10:0] execution_r;
    assign execution=execution_r;
    assign ALU_data2=ALU_data2_r;
    assign rr1=rr1_r;
    assign rr2=rr2_r;
    assign wr=wr_r;
    assign branch_address=branch_address_r;
    assign jump_address=jump_address_r;
    always @(posedge dec_en)
        begin
           case(inst[6:0])
             I1:
                begin
                  if (inst[14:12]==3'b010)
```



```
     if (inst[14:12]==3'b010)
        begin
          execution_r<=LW;             //command for control to generate corresponding control signal
          ALU_data2_r<={{21{inst[31]}},inst[30:20]}; // imm address for ALU to add to rd1 from the register file,
          rr1_r<=inst[19:15];  //rd1 index
          wr_r<=inst[11:7];   // target register index
        end
   end
I2:
   begin
     if ((inst[14:12]==3'b001)&&(inst[31:26]==6'b000000))
        begin
          execution_r<=SLLI;
          ALU_data2_r<={{27{inst[25]}},inst[24:20]}; // 32bit value for ALU input to shift the other input
          rr1_r<=inst[19:15];
          wr_r<=inst[11:7];   // target register index
        end
   end
S1:
   begin
     if (inst[14:12]==3'b010)
        begin
          execution_r<=SW;
          ALU_data2_r<={{21{inst[31]}},inst[30:25],inst[11:7]}; // 32bit value for ALU input to shift the other i
          rr1_r<=inst[19:15];
          rr2_r<=inst[24:20];   // register index to find the value to be stored
        end
   end
S2:
   begin
     if (inst[14:12]==3'b000)
        begin
          execution_r<=BEQ;
          branch_address_r<={{20{inst[31]}},inst[7],inst[30:25],inst[11:8],1'b0}; // 32bit value for ALU input to
          rr1_r<=inst[19:15];
          rr2_r<=inst[24:20];   // register index to find the value to be stored
        end
   end
R:
   begin
     if ((inst[14:12]==3'b000)&&(inst[31:25]==7'b0000000))
        begin
          execution_r<=ADD;
          rr1_r<=inst[19:15];
          rr2_r<=inst[24:20];
          wr_r<=inst[11:7];
        end
     if ((inst[14:12]==3'b000)&&(inst[31:25]==7'b0100000))
        begin
          execution_r<=SUB;
          rr1_r<=inst[19:15];
          rr2_r<=inst[24:20];
          wr_r<=inst[11:7];
        end
     if ((inst[14:12]==3'b001)&&(inst[31:25]==7'b0000000))
        begin
          execution_r<=SLL;
          rr1_r<=inst[19:15];
          rr2_r<=inst[24:20];
          wr_r<=inst[11:7];
        end
```



```
            end
          UJ:
            begin
                execution_r<=JAL;  //imm[20]| imm[10:1] | imm[11] | imm[19:12] | rd | opcode
                jump_address_r<={{12{inst[31]}},inst[19:12],inst[20],inst[30:21],1'b0}; // 32bit jump address
                wr_r<=inst[11:7];
            end
          SH:
            begin
              if (inst[31:7]==25'b0000000000001000000000000)
                begin
                execution_r<=HALT;  //imm[20]| imm[10:1] | imm[11] | imm[19:12] | rd | opcode
                end
            end
        endcase
      end
endmodule
//Mar. 2, Y. Zhao, EECS2021, YorkU
```

Fig. 6. Example of **inst_decoder** in verilog.

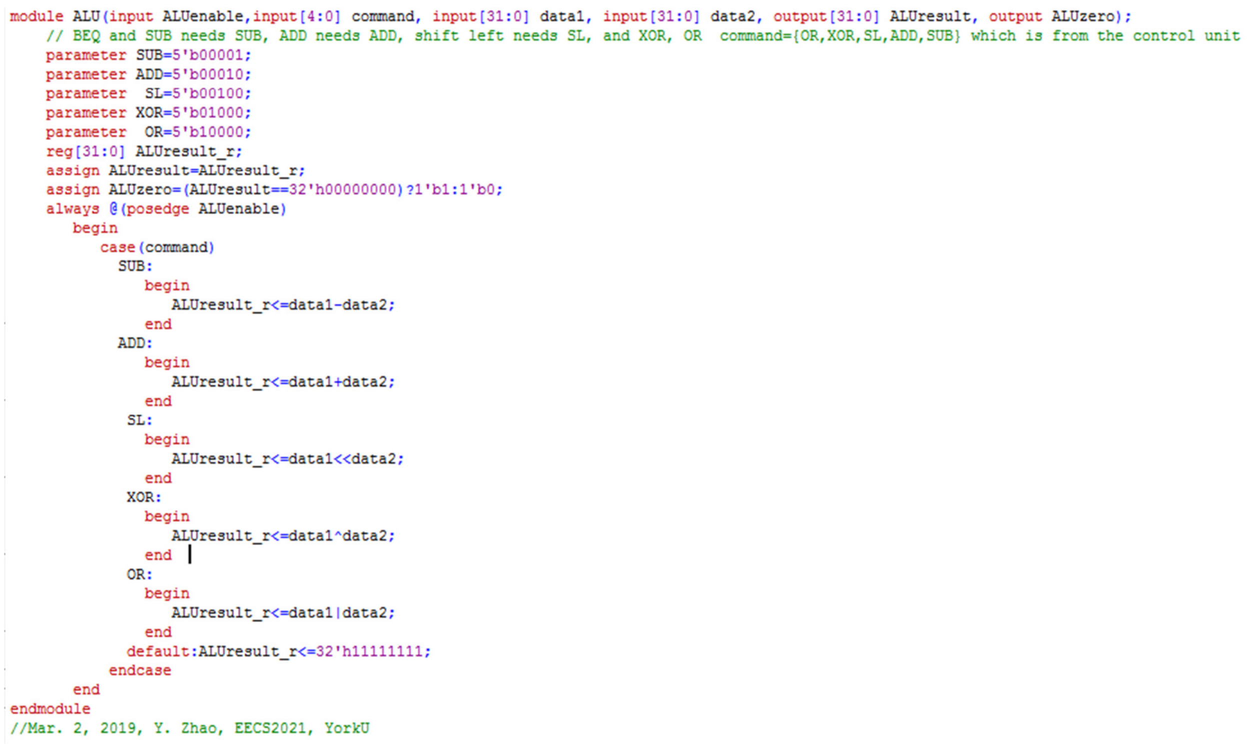
Table V. Ports definition of module **inst_decoder**

Port name	Property	To/From	Description
inst	input	**inst_ROM**	Instruction from ROM
dec_en	input	**control**	Decoding triggering single from **control**
rr1	output	**register**	Index of read register 1
rr2	output	**register**	Index of read register 2
wr	output	**register**	Index of write register 1
ALU_data2	output	**control**	One of the candidates for the second input of **ALU**
branch_address	output	**PC**	Branch address
jump_address	output	**PC**	Jump address
execution	output	**control**	Execution code required for control for deciding

Our ALU need SUB (for instruction SUB and BEQ), ADD (for instruction ADD, LW, SW), SL (shift left for instruction SLLI and SLL), XOR and OR operations to perform the illustrated 11 instructions according to the commands from **control** and output the arithmetic results and the equal indicator.

Table VI. Ports definition of module **ALU**

Port name	Property	To/From	Description
ALUenable	input	**control**	Signal from **control** to trigger the execution of **ALU**
command	input	**control**	**ALU** command for multiplexing
data1	input	**register**	First input always from **register**
data2	input	**control**	Second input is from the control, is *rd2* or *ALU_data2*
ALUresult	output	**control**	result
ALUzero	output	**control**	If equal, equal it is 1 otherwise is 0



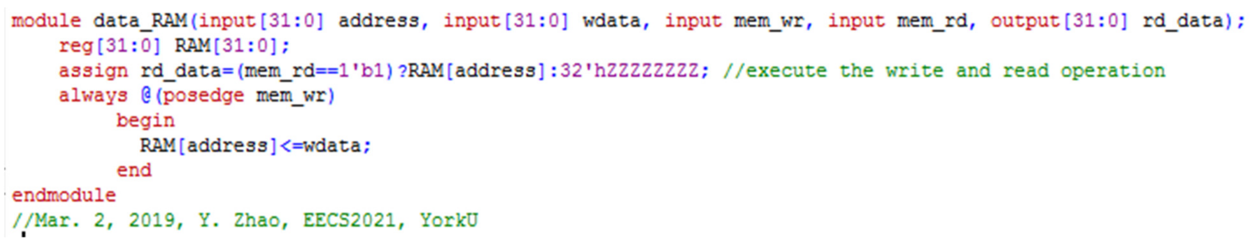
```
module ALU(input ALUenable,input[4:0] command, input[31:0] data1, input[31:0] data2, output[31:0] ALUresult, output ALUzero);
    // BEQ and SUB needs SUB, ADD needs ADD, shift left needs SL, and XOR, OR  command={OR,XOR,SL,ADD,SUB} which is from the control unit
    parameter SUB=5'b00001;
    parameter ADD=5'b00010;
    parameter  SL=5'b00100;
    parameter XOR=5'b01000;
    parameter  OR=5'b10000;
    reg[31:0] ALUresult_r;
    assign ALUresult=ALUresult_r;
    assign ALUzero=(ALUresult==32'h00000000)?1'b1:1'b0;
    always @(posedge ALUenable)
       begin
          case(command)
            SUB:
               begin
                  ALUresult_r<=data1-data2;
               end
            ADD:
               begin
                  ALUresult_r<=data1+data2;
               end
             SL:
               begin
                  ALUresult_r<=data1<<data2;
               end
             XOR:
               begin
                  ALUresult_r<=data1^data2;
               end
             OR:
               begin
                  ALUresult_r<=data1|data2;
               end
             default:ALUresult_r<=32'h11111111;
           endcase
       end
endmodule
//Mar. 2, 2019, Y. Zhao, EECS2021, YorkU
```

Fig. 7. **ALU** Verilog code.

Data memory is both readable and writable, we call it as **data_RAM**. For the LW, it will call the ALU to add imm with rs1 to calculate the target **data_RAM** address, then read the memory content and store the content to register *rd*. For SW, ALU will also be called to add imm with rs1, as the target **data_RAM** address. Then store the value from register *rs2* (i.e. *rd2* of the **register**)to **data_RAM** at the target address. Writing is triggered by the *mem_wr*, and reading is enabled by *mem_rd*. Its address is always from *ALUresult* since we have only LW and SW that require the **data_RAM** operation, and their address need to be calculated by **ALU**. The data written to it comes from *rd2* of **register**. Please refer to Table VII and Fig. 8 for the ports definition and example code.

Table VII. Ports definition of module **data_RAM**

Port name	Property	To/From	Description
address	input	**ALU**	Read or write address
wdata	input	**register**	Data to be written into RAM directly form register *rd2*
mem_wr	input	**control**	RAM write triggering
mem_rd	input	**control**	RAM read enable
rd_data	output	**control**	Output read data to **control** to decide if sent to *wd* of **register**



```
module data_RAM(input[31:0] address, input[31:0] wdata, input mem_wr, input mem_rd, output[31:0] rd_data);
    reg[31:0] RAM[31:0];
    assign rd_data=(mem_rd==1'b1)?RAM[address]:32'hZZZZZZZZ; //execute the write and read operation
    always @(posedge mem_wr)
         begin
           RAM[address]<=wdata;
         end
endmodule
//Mar. 2, 2019, Y. Zhao, EECS2021, YorkU
```

Fig. 8. **data_RAM** Verilog code

The control module operation is provided in Fig. 2. And the ports definitions are in Table VIII. Also the codes is given.

Table VIII. Ports definition of module **data_RAM**

Port name	Property	To/From	Description
clk	Input	System	Clock
rst	Input	System	Reset
execution	Input	**inst_decoder**	Instruction execution code from **inst_decoder**
ALU_data2	Input	**inst_decoder**	*imm*, one possible input of **ALU**
rd2	Input	**register**	*rs2* content, one possible input of **ALU**
ALUzero	Input	**ALU**	Equal indicator from **ALU**
pc_addr_plus	Input	**PC**	PC+1
ALUresult	Input	**ALU**	**ALU** arithmetic results
rd_data	Input	**data_RAM**	Data read from **data_RAM**
inc_pc	output	**PC**	Trigger update of **PC**
load_inst	output	**inst_ROM**	Enable instruction loading
dec_en	output	**inst_decoder**	Trigger decoding
mem_rd	output	**data_RAM**	Enable **RAM** reading
regwrite	output	**register**	Trigger **register** writing
wd	output	**register**	Data to be written into **register**
ALUenable	output	**ALU**	Trigger **ALU** operation
mem_wr	output	**data_RAM**	Trigger RAM writing
jump	output	**PC**	Jump indicator in **PC**
branch	output	**PC**	Branch indicator in **PC**
data2	output	**ALU**	Second input data of **ALU**
ALUcommand	output	**ALU**	Arithmetic configuration for **ALU**

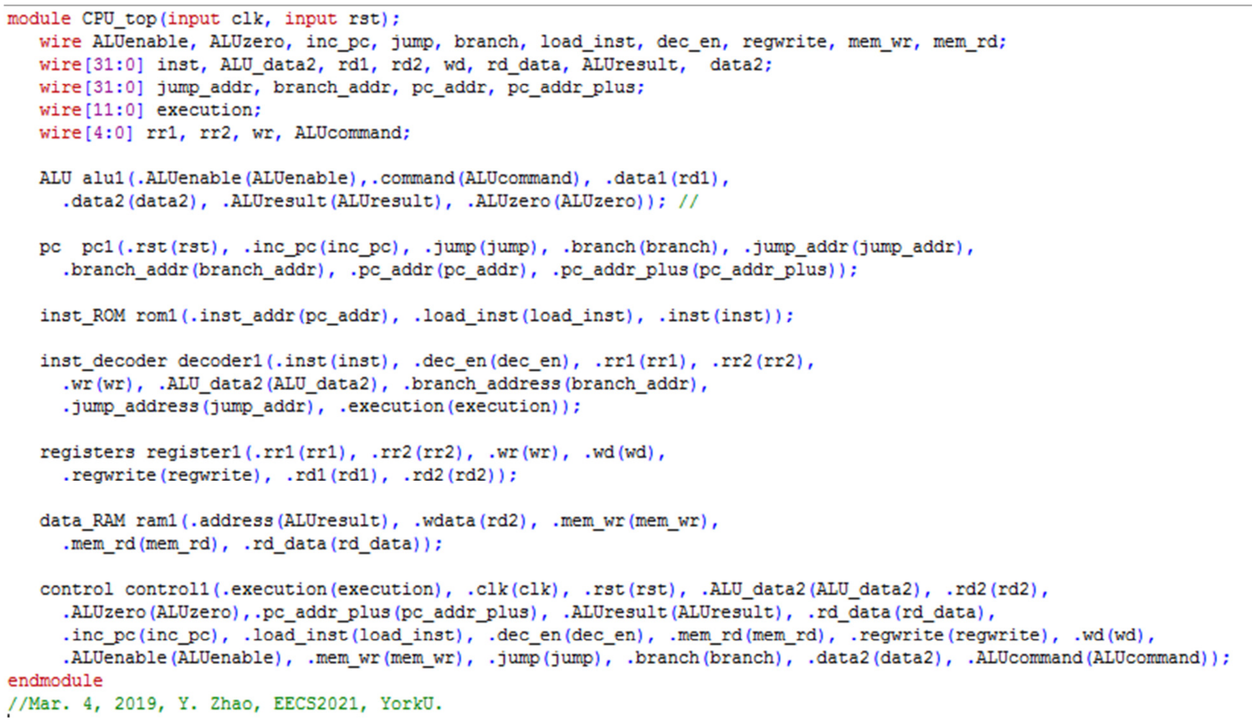
The code for this module is provided here.

```
module control(input[10:0] execution, input clk, input rst, input[31:0] ALU_data2, input[31:0] rd2, input ALUzero,
input[31:0] pc_addr_plus, input[31:0] ALUresult, input[31:0] rd_data,
output inc_pc, output load_inst, output dec_en, output mem_rd, output regwrite, output[31:0] wd,
output ALUenable, output mem_wr, output jump, output branch, output[31:0] data2, output[4:0] ALUcommand);
   parameter ALUSUB=5'b00001;
   parameter ALUADD=5'b00010;
   parameter  ALUSL=5'b00100;
   parameter ALUXOR=5'b01000;
   parameter  ALUOR=5'b10000;

   parameter    LW=11'b00000000001;
   parameter   SLLI=11'b00000000010;
   parameter    SW=11'b00000000100;
   parameter   BEQ=11'b00000001000;
```

```
    parameter    ADD=11'b00000010000;
    parameter    SUB=11'b00000100000;
    parameter    SLL=11'b00001000000;
    parameter    XOR=11'b00010000000;
    parameter     OR=11'b00100000000;
    parameter    JAL=11'b01000000000;
    parameter   HALT=11'b10000000000;

    parameter     fetch=3'b000;
    parameter  decoding=3'b001;
    parameter   control=3'b010;
    parameter executing=3'b011;
    parameter writeback=3'b100;
    parameter change_pc=3'b101;

    reg[2:0] state;
    reg[8:0] op_reg;
    reg[4:0] ALUcommand_r;
    reg[2:0] select_wd;
    reg ALUsrc;

    assign data2=(ALUsrc==1'b1)?ALU_data2:rd2; //mux to selec the second input of the ALU is from ALU_data2 or directly from register i.e. rd2.
    assign {load_inst, dec_en, ALUenable, mem_rd, regwrite, mem_wr, jump, branch, inc_pc}=op_reg;
    assign ALUcommand=ALUcommand_r;
    assign wd=({32{select_wd[2]}}&pc_addr_plus)|({32{select_wd[1]}}&ALUresult)|({32{select_wd[0]}}&rd_data);
    // if select_wd=3'b100  then the input of the register writing is from pc_addr_plus
   // if select_wd=3'b010 from ALUresult, else 3'b001 from rd_data of dataRAM
    always @(posedge clk or posedge rst)
       if (rst==1)
          begin
            state<=3'b000;
            op_reg<=9'b000000000;
            ALUsrc<=1'b0;
            ALUcommand_r<=5'b00000;
            select_wd<=3'b010;
          end
       else
          begin
            case(state)
              fetch:
                begin
                  op_reg<=9'b100000000; //load one instruction from inst_ROM based on the value of PC, initial value of PC is 0,
                  state<=decoding;
                end
           decoding:
                begin
                  op_reg<=9'b110000000; //fetched one instruction and start to send to inst_decoder for decoding
                  state<=control;
                end
            control:
                begin
                  case(execution)
                     LW,SW:                //if the the execution is load word, do the following preparationg, later we will find in this stage LW SW have same operation
                       begin
                         ALUsrc<=1'b1; //select the imm address as the second input of the ALU, noting first input of ALU always from register file i.e. rd1.
                         ALUcommand_r<=ALUADD; //ALU execute add operation during execution state
                         state<=executing; //state to executing stage when next clock comes
                       end
                     SLLI:               //if the the execution is shift left immediately
                       begin
                         ALUsrc<=1'b1; //select the imm value as the second input of the ALU to shift the rd1.
                         ALUcommand_r<=ALUSL; //
                         state<=executing; //state to executing stage when next clock comes
                       end
//                   SW:               //if the the execution is store word
//                       begin
//                         ALUsrc<=1'b1; //select the imm address as the second input of the ALU which will be addted to the adress in rd1 from register file.
//                         ALUcommand_r<=ALUADD; //ALU execute add operation, we can find that during the preparation stage, the LW and SW has same operation, we can joint them
//                       end
                     BEQ, SUB:               //if the the execution is branch when equal, we can find BEQ is same with SUB in this stage, we can joint them
                       begin
                         ALUsrc<=1'b0; //select the rd2 as the second input of ALU.
                         ALUcommand_r<=ALUSUB; //ALU execute sub operation
                         state<=executing; //state to executing stage when next clock comes
                       end
                     ADD:               //if the the execution is addition
                       begin
                         ALUsrc<=1'b0; //select the rd2.
                         ALUcommand_r<=ALUADD; //execute addition
                         state<=executing; //state to executing stage when next clock comes
                       end
                     SLL:               //if the the execution is logical shift left
                       begin
                         ALUsrc<=1'b0; //select the rd2.
                         ALUcommand_r<=ALUSL; //execute shift left
                         state<=executing; //state to executing stage when next clock comes
                       end
                     XOR:               //if the the execution is bit wise xor
                       begin
                         ALUsrc<=1'b0; //select the rd2.
                         ALUcommand_r<=ALUXOR; //execute bitwise xor
                         state<=executing; //state to executing stage when next clock comes
                       end
                     OR:               //if the the execution is bit wise or
                       begin
                         ALUsrc<=1'b0; //select the rd2.
                         ALUcommand_r<=ALUOR; //execute bitwise or
                         state<=executing; //state to executing stage when next clock comes
                       end
                     JAL:               //if the the execution is jump
                       begin
                         //dont need ALU for jump, so dont need ALU preparations
                         state<=executing; //state to executing stage when next clock comes
                       end
                     HALT:              //if the the execution is HALT
                       begin
                         //dont need ALU for jump, so dont need ALU preparations
                         state<=fetch; //state to instruction fetch state, since PC keeps same, instructions fetched is still HALT,
                    //therefore, the CPU will keep looping from the instruction fetch to control, we can take is as shutdown
                end
              endcase
              op_reg<=9'b000000000;
            end
        executing:
            begin
              case(execution)
                LW, SLLI, SW, BEQ, ADD, SUB, SLL, XOR, OR:op_reg<=9'b001000000; //executing ALU when the command requires
              endcase
              state<=writeback;
            end
        writeback:
            begin
              case(execution)
                LW:
                  begin
                    op_reg[5]<=1'b1; //start reading the data ram content at the address indexed by the ALUresult
                    select_wd<=3'b001; //select the output of data ram as the input of register writing
                  end
                SW:
                  begin
                    op_reg[3]<=1'b1; //start writing rd2 to the data ram at the address indexed by the ALUresult
                  end
                SLLI, ADD, SUB, SLL, XOR, OR:
                  begin
                    select_wd<=3'b010; //select the output of ALUresult ram as the input of register writing
                  end
                BEQ:
                  begin
                    op_reg[1]<=(ALUzero==1'b1)?1'b1:1'b0; //if the value is same , then set branch to 1
                  end
                JAL:
                  begin
                    select_wd<=3'b100; //select pc_addr_plus as the input of register writing
                    op_reg[2]<=1'b1; //set the jump to 1 for instruction address updating in pc
                  end
              endcase
              state<=change_pc;
              op_reg[6]<=1'b0; //change ALUenable back to 0
            end
        change_pc:
            begin
              case(execution)
                LW, SLLI, ADD, SUB, SLL, XOR, OR, JAL:op_reg[4]<=1'b1; //write the result to register file
              endcase
              state<=fetch; //finished the command and return to the instruction fetch stage
              op_reg[0]<=1'b1; //for all commands, update the pc trigierd by the rising edge of inc_pc
            end
        default:
            begin
              state<=3'b000;
              op_reg<=9'b000000000;
              ALUsrc<=1'b0;
              ALUcommand_r<=5'b00000;
              select_wd<=3'b010;
            end
        endcase
      end
endmodule
//Mar. 3, 2019, Y. Zhao, EECS2021, YorkU
```

Organize the modules

Based on the ports definition for each module, we can connect them together to achieve the entire design as in Fig. 9.



```
module CPU_top(input clk, input rst);
   wire ALUenable, ALUzero, inc_pc, jump, branch, load_inst, dec_en, regwrite, mem_wr, mem_rd;
   wire[31:0] inst, ALU_data2, rd1, rd2, wd, rd_data, ALUresult,  data2;
   wire[31:0] jump_addr, branch_addr, pc_addr, pc_addr_plus;
   wire[11:0] execution;
   wire[4:0] rr1, rr2, wr, ALUcommand;

   ALU alu1(.ALUenable(ALUenable),.command(ALUcommand), .data1(rd1),
     .data2(data2), .ALUresult(ALUresult), .ALUzero(ALUzero)); //

   pc  pc1(.rst(rst), .inc_pc(inc_pc), .jump(jump), .branch(branch), .jump_addr(jump_addr),
     .branch_addr(branch_addr), .pc_addr(pc_addr), .pc_addr_plus(pc_addr_plus));

   inst_ROM rom1(.inst_addr(pc_addr), .load_inst(load_inst), .inst(inst));

   inst_decoder decoder1(.inst(inst), .dec_en(dec_en), .rr1(rr1), .rr2(rr2),
     .wr(wr), .ALU_data2(ALU_data2), .branch_address(branch_addr),
     .jump_address(jump_addr), .execution(execution));

   registers register1(.rr1(rr1), .rr2(rr2), .wr(wr), .wd(wd),
     .regwrite(regwrite), .rd1(rd1), .rd2(rd2));

   data_RAM ram1(.address(ALUresult), .wdata(rd2), .mem_wr(mem_wr),
     .mem_rd(mem_rd), .rd_data(rd_data));

   control control1(.execution(execution), .clk(clk), .rst(rst), .ALU_data2(ALU_data2), .rd2(rd2),
     .ALUzero(ALUzero),.pc_addr_plus(pc_addr_plus), .ALUresult(ALUresult), .rd_data(rd_data),
     .inc_pc(inc_pc), .load_inst(load_inst), .dec_en(dec_en), .mem_rd(mem_rd), .regwrite(regwrite), .wd(wd),
     .ALUenable(ALUenable), .mem_wr(mem_wr), .jump(jump), .branch(branch), .data2(data2), .ALUcommand(ALUcommand));
endmodule
//Mar. 4, 2019, Y. Zhao, EECS2021, YorkU.
```

Fig. 9. Connections between each module refer to the ports definition and Fig. 2.