FAET630004: AI-Core and RISC Architecture **(Due: 4/2/21)**

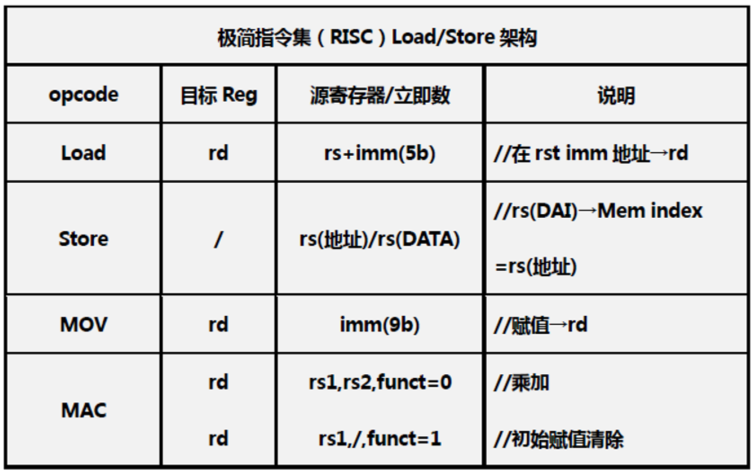
Homework Assignment #1

Instructor: Chixiao Chen *Name:* Chunyu Wang, *FudanID:* 20210860017

- This HW counts 15% of your final score, please treat it carefully.
- Please submit the electronic copy via mail: faet_english@126.com before 03/23/2019 11:59pm.
- It is encouraged to use LaTeX to edit it, the source code of the assignment is available via: https://www.overleaf.com/read/gmxszwbypznk
- You can also open it by Office Word, and save it as a .doc file for easy editing. Also, you can print it out, complete it and scan it by your cellphone.
- The assignment needs verilog/SV simulation. It is suggested to use Vivado from Xilinx to complete the simulation. If you do not want to install a local verilog simulator, please use an online tool: https://www.edaplayground.com/, you need register for save.
- For students who prefer C rather than Verilog, a C-based hardware description method is also recommended. It is called high level synthesis (HLS) supported by Vivado as well. A HLS tutorial can be found via: https://www.bilibili.com/video/BV11b411e7m3. Also, an HLS example of RISC RISC is available by: https://gitlab.cs.washington.edu/cse599s/hls-tutorials/tree/master/part3.
- You can answer the assignment either in Chinese or English

Problem 1: Implement a simple RISC Core (2+5+8=15 points)

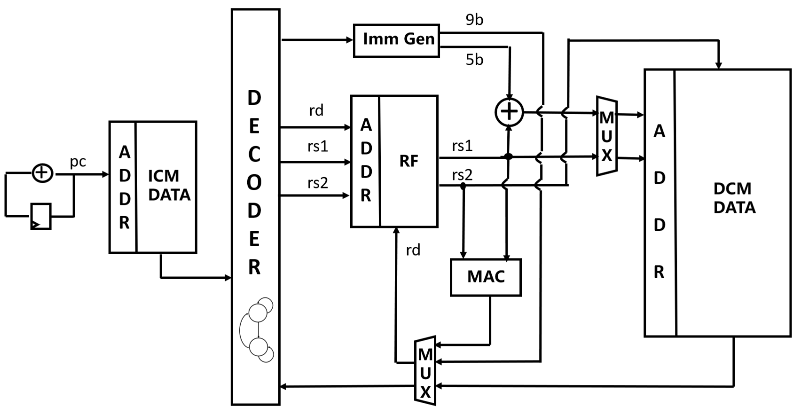
On class, we define an extremely simple RISC ISA and its hardware organization, which are both shown below. For this homework assignment, we need to design/implement/simulate this core.



极简指令集（RISC）Load/Store 架构			
opcode	目标 Reg	源寄存器/立即数	说明
Load	rd	rs+imm(5b)	//在 rst imm 地址→rd
Store	/	rs(地址)/rs(DATA)	//rs(DAI)→Mem index =rs(地址)
MOV	rd	imm(9b)	//赋值→rd
MAC	rd	rs1,rs2,funct=0	//乘加
	rd	rs1,/,funct=1	//初始赋值清除

(a) Which HDL you want to use ?
verilog.

(b) For Verilog-players, please write a top-level structural verilog file to describe the system.
For C-players, please complete a header filer and .cc file , compatible to Vivado HLS, to describe the system.



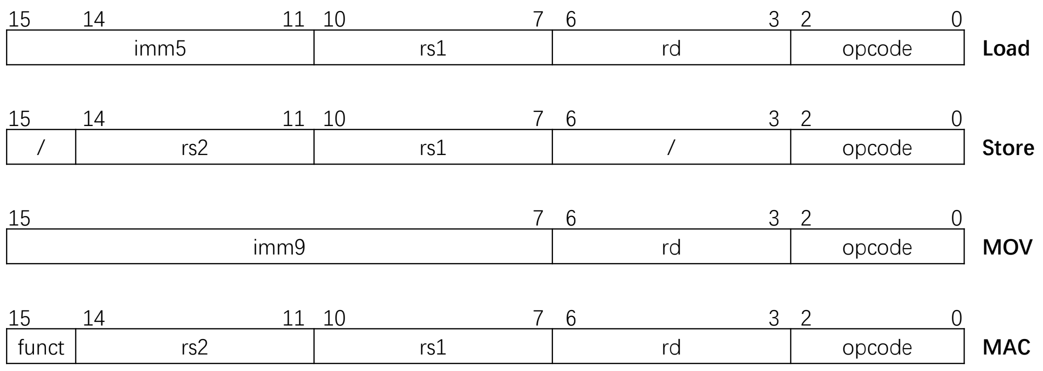
(Hint: Please submit your script as well)

The top-level structual verilog file is implemented in .srcs/sources_1/new/RISCV.v.

(c) For Verilog-players, please write a testbench verilog file to simulate the system. It should complete a 4-MAC (neuron) computing.
For C-players, please complete a _test.cc file , compatible to Vivado HLS, to simulate the system. It should complete a 4-MAC (neuron) computing.
(Hint: Please submit your script and simulated waveform, highlight the final results and compare it with the theoretical value.)

The testbench verilog file is implemented in .srcs/sim_1/new/RISCV.v.

1 Design and Implementation



(a)

	Load	Store	MOV	MAC
Opcode	001	010	011	100

(b)

Figure 1: Instruction Format

The instruction format of this simple RISC ISA is shown in Fig.1. Field **opcode** ocuppies the lowest 3 bits of a instruction, and the binaries defined as Fig.1(b).

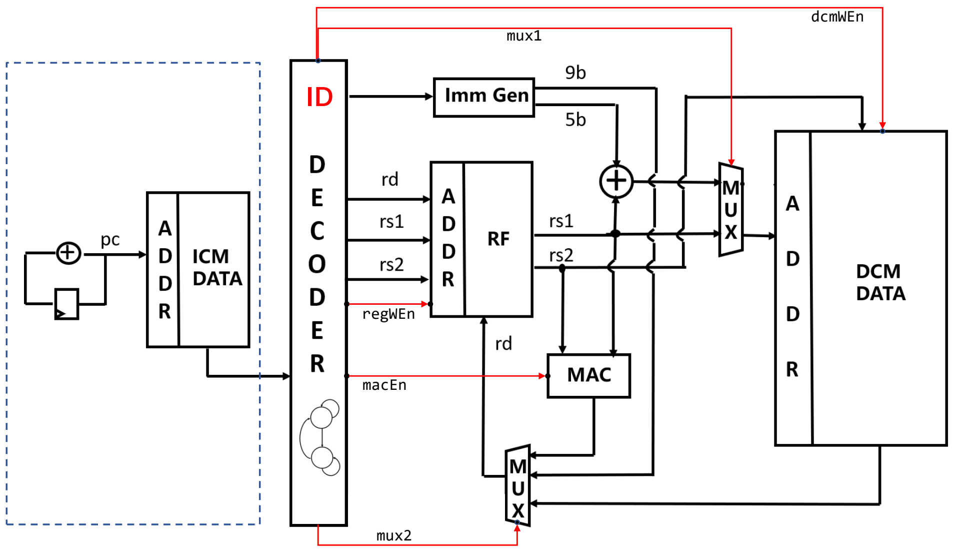
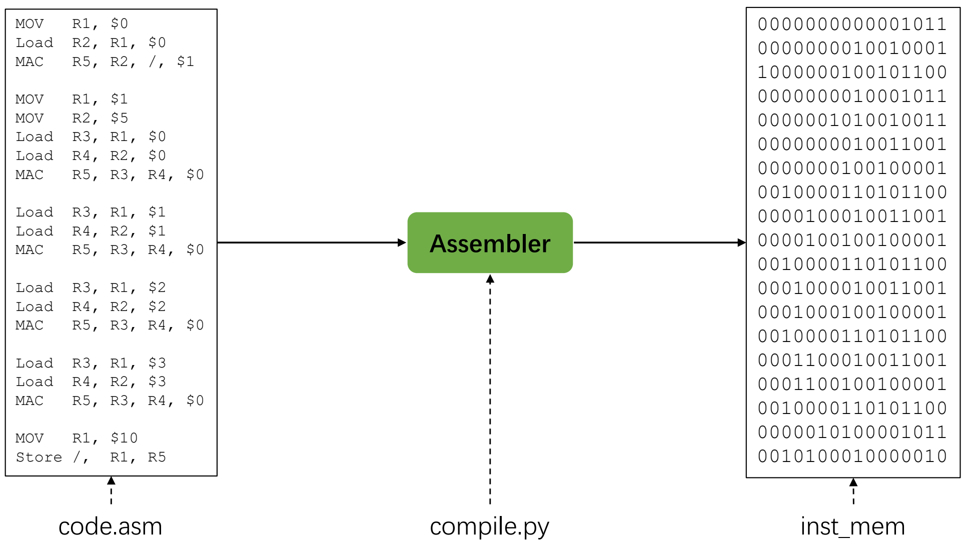


Figure 2: Hardware orgnization

As shown in Fig.2, there are several modules connected by wires. The ID(Instruction Decoder) module will decode a instruction by opcode and then generate control signals. Respectively, regWEn, macEn and dcmWEn will enable regfile writing, MAC and DCM writing; mux1, mux2 will control MUXes as selective signals.

All of the modules are combined together in a top-level structual verilog file in RISCV.v to discribe the system. And the top-level structual verilog file is implemented in **.srcs/sources_1/new/RISCV.v**.



```
MOV   R1, $0
Load  R2, R1, $0
MAC   R5, R2, /, $1

MOV   R1, $1
MOV   R2, $5
Load  R3, R1, $0
Load  R4, R2, $0
MAC   R5, R3, R4, $0

Load  R3, R1, $1
Load  R4, R2, $1
MAC   R5, R3, R4, $0

Load  R3, R1, $2
Load  R4, R2, $2
MAC   R5, R3, R4, $0

Load  R3, R1, $3
Load  R4, R2, $3
MAC   R5, R3, R4, $0

MOV   R1, $10
Store /,  R1, R5
```

```
0000000000001011
0000000010010001
1000000100101100
0000000010001011
0000001010010011
0000000010011001
0000000100100001
0010000110101100
0000100010011001
0000100100100001
0010000110101100
0001000010011001
0001000100100001
0010000110101100
0001100010011001
0001100100100001
0010000110101100
0000010100001011
0010100010000010
```

Figure 3: Tool: translate assembly language to machine language(binary code)

It is hard for us to write machine code to test our system directly, so I wrote python scrips to translate assembly language to machine language(binary code) which strictly observes our instruction format above, just as shown in Fig.3. And you can also find this python scrip in **.srcs/sim_1/new/compile.py**.

For the same reason, I created a generator to generate data for the MAC opration including bias(b), inputs(x_1, x_2, x_3, x_4) and weights(w_1, w_2, w_3, w_4). Inputs of data_gen.py are decimals and outputs which saved in data memory are their binary complements just as shown in Fig.4.
And you can also find this python scrip in **.srcs/sim_1/new/data_gen.py**.

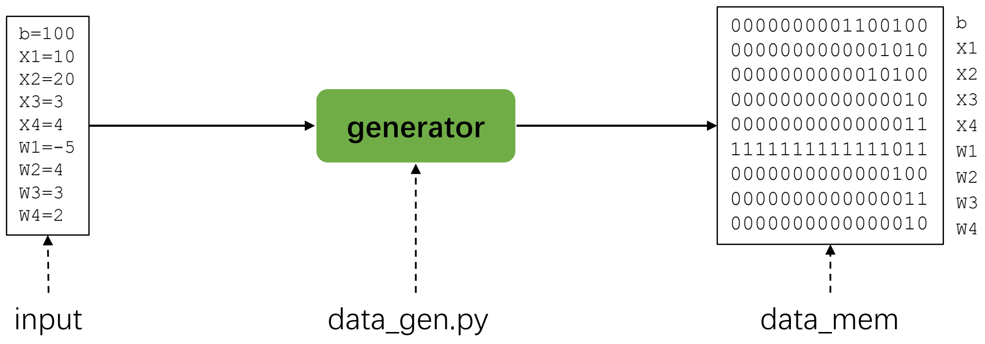


Figure 4: Tool: generate testing data from decimals to binaries(16bit complement)

2 Simulation and Testing

We can write assemble code for 4MAC oparation as below, input data in data memory is orgnized like Fig.4 and this assemble code will be traslated to machine code using my tool above.

```
MOV  R1, $0
Load R2, R1, $0
MAC  R5, R2, /, $1

MOV  R1, $1
MOV  R2, $5
Load R3, R1, $0
Load R4, R2, $0
MAC  R5, R3, R4, $0

Load R3, R1, $1
Load R4, R2, $1
MAC  R5, R3, R4, $0

Load R3, R1, $2
Load R4, R2, $2
MAC  R5, R3, R4, $0

Load R3, R1, $3
Load R4, R2, $3
MAC  R5, R3, R4, $0

MOV  R1, $10
Store /, R1, R5
```

Now, we can start our simulation in vivado. We will run 2 tests to test our system.

2.1 Test 1

Input data and simulation waveforms screenshot of test1 are shown in Table.1 and Fig.5.
You can also open the .wcfg file to view waveforms.

Table 1: Test 1 inputs for MAC operation

	decimal	**Binary**
b	100	0000000001100100
x_1	10	0000000000001010
x_2	20	0000000000010100
x_3	2	0000000000000010
x_4	3	0000000000000011
w_1	-5	1111111111111011
w_2	4	0000000000000100
w_3	3	0000000000000011
w_4	2	0000000000000010
Expected Result	142	0000000010001110

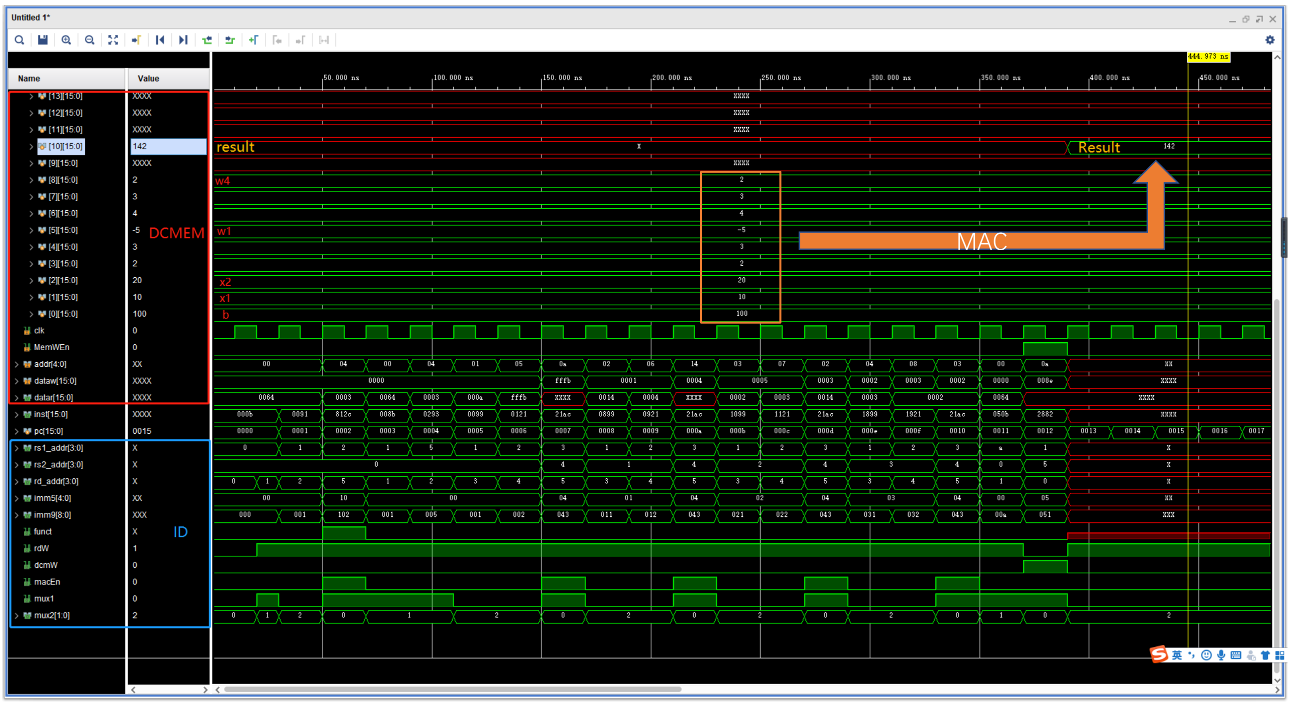


Figure 5: Test 1 simulation waveform screenshot)

2.2 Test 2

Input data and simulation waveform screenshot of test2 are shown in Table.2 and Fig.6.

Table 2: Test 2 inputs for MAC operation

	decimal	**Binary**
b	200	0000000011001000
x_1	123	0000000001111011
x_2	321	0000000101000001
x_3	1	0000000000000001
x_4	1	0000000000000001
w_1	1	0000000000000001
w_2	-1	1111111111111111
w_3	-2	1111111111111110
w_4	-3	1111111111111101
Expected Result	-3	1111111111111101

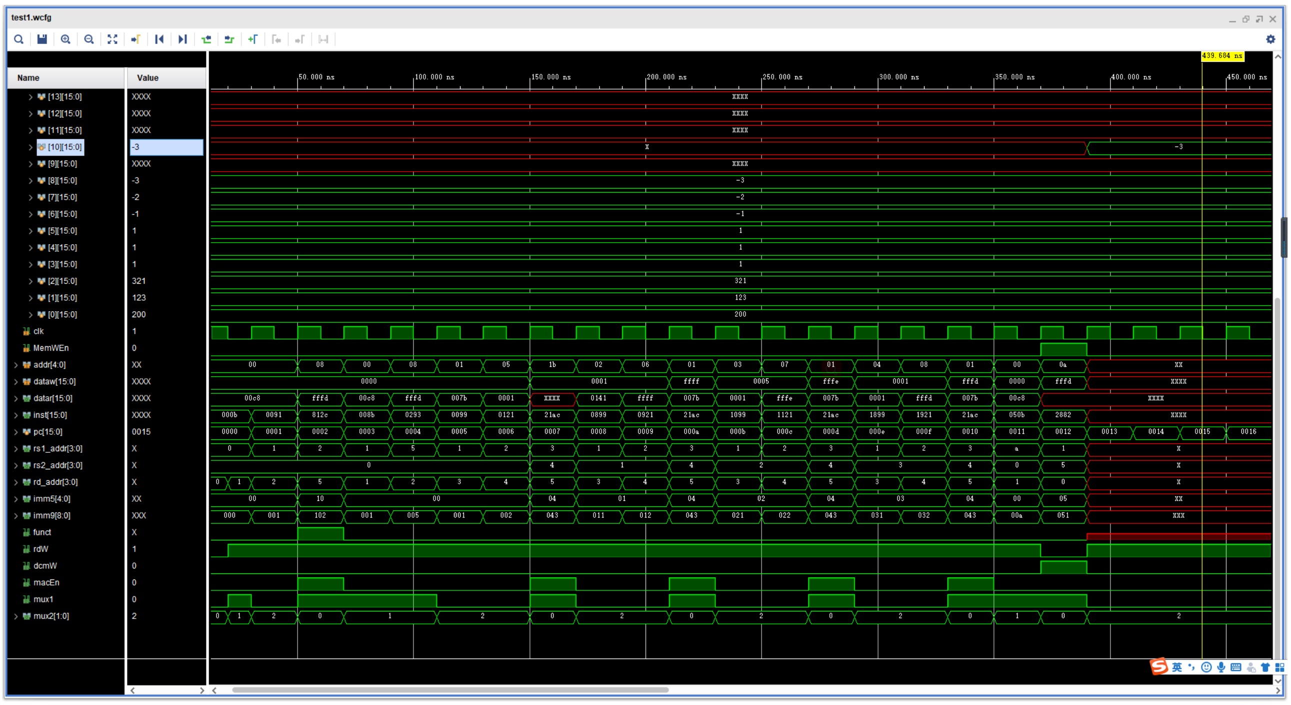


Figure 6: Test 1 simulation waveform screenshot)

3 Source Code

You can find all source code in my github via: https://github.com/yiwangchunyu/RISC-V.

3.1 Python

3.1.1 compile.py

This script is used to translate assemble code to machine code.

```
import re

def itob(x:int, width:int):
    s = bin(x)[2:]
    s=s.rjust(width, '0')
    return s

def process_line(line:str):
    line=line.upper().strip()
    bcode=''
    if line.startswith('LOAD'):
        opcode='001'
        line=line[4:].strip()
        rd,_,_,rs,_,_,imm,_ =
            re.search('^R(\d+)(\s*),(\s*)R(\d+)(\s*),(\s*)\$(\d+)(.*)$',line).groups()
        bcode='%s%s%s%s'%(itob(int(imm),5),itob(int(rs),4),itob(int(rd),4),opcode)
        return bcode+'\n'
    elif line.startswith('STORE'):
        opcode='010'
        line = line[5:].strip()
        _, _, rs1, _, _, rs2, _= re.search('^/(\s*),(\s*)R(\d+)(\s*),(\s*)R(\d+)(.*)$', line).groups()
        bcode = '%s%s%s%s' % (itob(int(rs2), 5), itob(int(rs1), 4), itob(0, 4), opcode)
        return bcode + '\n'
    elif line.startswith('MOV'):
        opcode='011'
        line = line[3:].strip()
        rd, _, _, imm, _= re.search('^R(\d+)(\s*),(\s*)\$(\d+)(.*)$', line).groups()
        bcode = '%s%s%s' % (itob(int(imm), 9), itob(int(rd), 4), opcode)
        return bcode + '\n'
    elif line.startswith('MAC'):
        opcode='100'
        line = line[3:].strip()
        _,funct,_=re.search('^(.*)\$(\d+)(.*)$',line).groups()
        funct=int(funct)
        rs2='0'
        rs1='0'
        if(funct==1):
            rd, _, _, rs1, _ = re.search('^R(\d+)(\s*),(\s*)R(\d+)(.*)$', line).groups()
        elif(funct==0):
            rd, _, _, rs1, _, _, rs2, _ = re.search('^R(\d+)(\s*),(\s*)R(\d+)(\s*),(\s*)R(\d+)(.*)$',
                line).groups()
        else:
            pass
        bcode = '%s%s%s%s%s' % (itob(funct, 1), itob(int(rs2), 4), itob(int(rs1), 4), itob(int(rd),
            4), opcode)
        return bcode + '\n'
    else:
        pass
    return bcode

def compile(sfilename:str,bfilename:str):
    fb=open(bfilename,'w')
    with open(sfilename,'r') as fs:
        line_num=1
        for line in fs:
            try:
                bcode=process_line(line)
            except:
                print('Errors in line %d: %s'%(line_num,line))
                fb.close()
                exit(-2)
            end='\n' if len(bcode)<16 else ''
            print("%-3d: %s"%(line_num,bcode),end=end)
            fb.write(bcode)
            line_num+=1
    fb.close()

if __name__=="__main__":
    compile("code.asm","inst_mem.txt")
```

3.1.2 data_gen.py

This script is used to generate MAC data which is writen in binary.

```
import numpy as np
def itob(x:int):
    if x>=0:
        s = bin(x)[2:]
        s=s.rjust(16, '0')
    else:
        x=x&0xffff
        s=bin(x)[2:]
    return s

b = input('input bias:')
xs = input('input xs from x1~xn seprated by space:')
ws = input('input ws from w1~wn seprated by space:')
b=int(b)
```

```
xs=list(map(int,xs.split()))
ws=list(map(int,ws.split()))

with open('data_mem.txt','w') as f:
   f.write(itob(b)+'\n')
   for x in xs:
      f.write(itob(x) + '\n')
   for w in ws:
      f.write(itob(w) + '\n')

# compute mac result
res=b+np.dot(xs,ws)
print('b+xs.dot(ws) = %d'%res)
```

3.2 Verilog

3.2.1 RISCV.v

This a top-level structual verilog file.

```
`timescale 1ns / 1ps

module RISCV(
   input clk,rst
   );
   wire   [15:0]  inst;

  wire [3:0] rs1_addr;
  wire [3:0] rs2_addr;
  wire [3:0] rd_addr;
  wire [15:0]  dcm_addr;
  wire [15:0]  dcm_o;
  wire [15:0]  rs1_data;
  wire [15:0]  rs2_data;
  wire [15:0]  rd_data;
  wire [15:0]  imm_add_sum;
  wire [15:0]  mac_res;
  wire       rdW;
  wire       dcmW;
  wire [4:0] imm5;
  wire [8:0] imm9;
  wire       funct;
  wire       mux1;
  wire [1:0] mux2;
  wire       macEn;

  icmem icmem_0(
     .clk(clk),
     .rst(rst),
     .inst(inst)
  );
  dcmem dcmem_0(
     .clk(clk),
     .MemWEn(dcmW),
     .addr(dcm_addr),
     .dataw(rs2_data),
     .datar(dcm_o)
  );
  ID id_0(
     .rst(rst),
     .inst(inst),
     .rs1_addr(rs1_addr),
     .rs2_addr(rs2_addr),
     .rd_addr(rd_addr),
     .rdW(rdW),
     .dcmW(dcmW),
     .imm5(imm5),
     .imm9(imm9),
     .funct(funct),
     .mux1(mux1),
     .mux2(mux2),
     .macEn(macEn)
  );
  regfile regfile_0(
     .clk(clk),
     .rst(rst),
     .rs1_addr(rs1_addr),
     .rs2_addr(rs2_addr),
     .rd_addr(rd_addr),
     .RegWEn(rdW),
     .rs1_data(rs1_data),
     .rs2_data(rs2_data),
     .rd_data(rd_data)
  );
  IMM_ADD imm_add_0(
     .imm(imm5),
     .rs1(rs1_data),
     .dst(imm_add_sum)
  );
  MAC_ALU mac_alu_0(
     .clk(clk),
     .funct(funct),
     .macEn(macEn),
     .rs1(rs1_data),
     .rs2(rs2_data),
     .rd(mac_res)
  );
  MUX2 mux1_0(
     .S(mux1),
     .A(imm_add_sum),
     .B(rs1_data),
     .Y(dcm_addr)
  );
  MUX3 mux2_0(
     .S(mux2),
     .A(mac_res),
     .B(imm9),
     .C(dcm_o),
     .Y(rd_data)
  );
endmodule
```

3.2.2 icmem.v

```
`timescale 1ns / 1ps

module icmem #(
  parameter PC_WIDTH=16,
  parameter ISA_WIDTH=16,
  parameter MEM_NUMBER=100
   )(
    input                  clk,
    input                  rst,
    output  [ISA_WIDTH-1:0]       inst
  );
  reg[PC_WIDTH-1:0] pc;
  reg [PC_WIDTH-1:0] RAM[MEM_NUMBER-1:0];

  initial $readmemb("inst_mem.txt",RAM);

  always@(posedge clk or negedge rst)
    if (!rst)
        pc<=0;
    else
        pc<=pc+1;

  assign inst = RAM[pc];
endmodule
```

3.2.3 dcmem.v

```
`timescale 1ns / 1ps

module dcmem #(
  parameter MEM_ADDR_WIDTH=5,
  parameter MEM_DATA_WIDTH=16,
  parameter MEM_NUMBER=32
   )(
    input                   clk,
    input                   MemWEn, //memory write enabel
    input   [MEM_ADDR_WIDTH-1:0] addr,
    input [MEM_DATA_WIDTH-1:0] dataw,
    output  [MEM_DATA_WIDTH-1:0] datar

  );
  reg [MEM_DATA_WIDTH-1:0] RAM[MEM_NUMBER-1:0];
  initial $readmemb("data_mem.txt",RAM);

  always@(posedge clk)begin
    if(MemWEn)begin
      RAM[addr]<=dataw;
    end
  end

   assign datar = RAM[addr];
endmodule
```

3.2.4 ID.v

```
`timescale 1ns / 1ps

module ID#(
   parameter ISA_WIDTH=16,
   parameter REG_ADDR_WIDTH=4,
  parameter RED_DATA_WIDTH=16
)(
   input wire                        rst,
   input wire    [ISA_WIDTH-1:0]     inst,

   output reg    [REG_ADDR_WIDTH-1:0] rs1_addr,
   output reg    [REG_ADDR_WIDTH-1:0] rs2_addr,
  output reg    [REG_ADDR_WIDTH-1:0] rd_addr,
   output reg                      rdW,
  output reg                      dcmW,
   output reg    [4:0]             imm5,
   output reg    [8:0]             imm9,
  output reg                      funct,
  output reg                 mux1,
  output reg    [1:0]           mux2,
  output reg                      macEn
   );

   always @ (*) begin
       if(!rst)
        rs1_addr<=4'b0;
       else
        rs1_addr<=inst[10:7];
   end

  always @ (*) begin
       if(!rst)
        rs2_addr<=4'b0;
       else
        rs2_addr<=inst[14:11];
   end

  always @ (*) begin
       if(!rst)
        rd_addr<=4'b0;
       else
        rd_addr<=inst[6:3];
   end
  always @ (*) begin
       if(!rst)
        imm5<=5'b0;
       else
        imm5<=inst[15:11];
   end
  always @ (*) begin
       if(!rst)
        imm9<=9'b0;
       else
        imm9<=inst[15:7];
   end
  always @ (*) begin
       if(!rst)
        funct<=1'b0;
       else
        funct<=inst[15];
   end
  always @ (*) begin
       if(!rst)
        rdW<=1'b0;
       else begin
        casex(inst)
           16'bxxxxxxxxxxxxx001: rdW<=1'b1; //load
           16'bxxxxxxxxxxxxx010: rdW<=1'b0; //store
           16'bxxxxxxxxxxxxx011: rdW<=1'b1; //mov
           16'bxxxxxxxxxxxxx100: rdW<=1'b1; //mac
           default: rdW<=1'b0;
        endcase
     end
   end
  always @ (*) begin
       if(!rst)
```

```
         dcmW<=1'b0;
        else begin
         casex(inst)
            16'bxxxxxxxxxxxxx001: dcmW<=1'b0; //load
            16'bxxxxxxxxxxxxx010: dcmW<=1'b1; //store
            16'bxxxxxxxxxxxxx011: dcmW<=1'b0; //mov
            16'bxxxxxxxxxxxxx100: dcmW<=1'b0; //mac
            default: dcmW<=1'b0;
         endcase
      end
    end
   always @ (*) begin
        if(!rst)
         mux1<=1'b0;
        else begin
         casex(inst)
            16'bxxxxxxxxxxxxx001: mux1<=1'b0; //load
            16'bxxxxxxxxxxxxx010: mux1<=1'b1; //store
            16'bxxxxxxxxxxxxx011: mux1<=1'b1; //mov
            16'bxxxxxxxxxxxxx100: mux1<=1'b1; //mac
            default: mux1<=1'b0;
         endcase
      end
    end
   always @ (*) begin
        if(!rst)
         mux2<=2'b00;
        else begin
         casex(inst)
            16'bxxxxxxxxxxxxx001: mux2<=2'b10; //load
            16'bxxxxxxxxxxxxx010: mux2<=2'b0; //store
            16'bxxxxxxxxxxxxx011: mux2<=2'b01; //mov
            16'bxxxxxxxxxxxxx100: mux2<=2'b00; //mac
            default: mux2<=2'b0;
         endcase
      end
    end
    always @ (*) begin
        if(!rst)
         macEn<=1'b0;
        else begin
         casex(inst)
            16'bxxxxxxxxxxxxx001: macEn<=1'b0; //load
            16'bxxxxxxxxxxxxx010: macEn<=1'b0; //store
            16'bxxxxxxxxxxxxx011: macEn<=1'b0; //mov
            16'bxxxxxxxxxxxxx100: macEn<=1'b1; //mac
            default: macEn<=1'b0;
         endcase
      end
    end
endmodule
```

3.2.5 regfile.v

```
`timescale 1ns / 1ps

module regfile#(
      parameter REG_ADDR_WIDTH=4,
      parameter RED_DATA_WIDTH=16,
      parameter REG_NUMBER=16
   )(
      input [REG_ADDR_WIDTH-1:0] rs1_addr,
      input [REG_ADDR_WIDTH-1:0] rs2_addr,
      input [REG_ADDR_WIDTH-1:0] rd_addr,
      input [RED_DATA_WIDTH-1:0] rd_data,
      input RegWEn,
      input clk,rst,
      output [RED_DATA_WIDTH-1:0] rs1_data,
      output [RED_DATA_WIDTH-1:0] rs2_data
     );
   reg [RED_DATA_WIDTH-1:0] rf[REG_NUMBER-1:0];

   assign rs1_data = rs1_addr==0?0:rf[rs1_addr];
   assign rs2_data = rs2_addr==0?0:rf[rs2_addr];
     integer i;
   always@(posedge clk or negedge rst)begin
      if(!rst) begin
        for(i=0;i<REG_NUMBER;i=i+1)begin
          rf[i]<=0;
        end
      end
      else if(RegWEn&&rd_addr!=0)
        rf[rd_addr]<=rd_data;
   end
endmodule
```

3.2.6 MAC_ALU.v

```
`timescale 1ns / 1ps

module MAC_ALU#(
    parameter REG_DATA_WIDTH=16
  )(
    input funct, clk, macEn,
    input [REG_DATA_WIDTH-1:0] rs1,rs2,
    output [REG_DATA_WIDTH-1:0] rd
   );
  wire [REG_DATA_WIDTH-1:0] product;
  wire [REG_DATA_WIDTH-1:0] addend1,addend2;
  reg [REG_DATA_WIDTH-1:0] psum;

  assign product = $signed(rs1)*$signed(rs2);
  assign addend1 = funct?0:product;
  assign addend2 = funct?rs1:psum;
  assign rd    = addend1+addend2;

  always@(posedge clk) begin
    if(macEn)
        psum<=rd;
  end
endmodule
```

3.2.7 IMM_ADD.v

```
`timescale 1ns / 1ps

module IMM_ADD(
   input [4:0] imm,
   input [15:0] rs1,
   output reg[15:0] dst
   );
   always@(*)begin
      dst<=imm+rs1;
   end
endmodule
```

3.2.8 MUX2.v

```
`timescale 1ns / 1ps

module MUX2(
   input S,
   input [15:0] A,B,
   output [15:0] Y
    );
   assign Y=S==0?A:B;
endmodule
```

3.2.9 MUX3.v

```
`timescale 1ns / 1ps

module MUX3(
   input [1:0]S,
   input [15:0] A,C,
   input [8:0] B,
   output [15:0] Y
    );
   assign Y=(S==2'b00)?A:((S==2'b01)?B:C);
endmodule
```

3.2.10 RISCV_tb.v

This is a testbench verilog.

```
`timescale 1ns / 1ps
module RISCV_tb(

    );
   reg clk,rst;

   initial begin
      clk=1'b0;
      forever #10 clk=~clk;
   end
   initial begin
      rst=1'b0;
      #20 rst=1'b1;

      #1000  $stop;

   end

   RISCV riscv(
      .clk(clk),
      .rst(rst)
   );
endmodule
```