8 位 CPU 的 Verilog 实现

Kimchange

2020 年 6 月 6 日

目录

1 设计需求

1.1 设计指标

采用冯诺依曼体系结构，即计算机由控制器，运算器，存储器，和输入/输出设备五部分组成。本文主要考虑设计控制单元 (control unit)，算术逻辑单元 ALU(Arithmetic Logic Unit)，随机存取存储器 RAM(Radom Access Memory) 三个模块。在这三个子模块之上还有顶层模块 (core) 和测试模块 (core_testbench)。

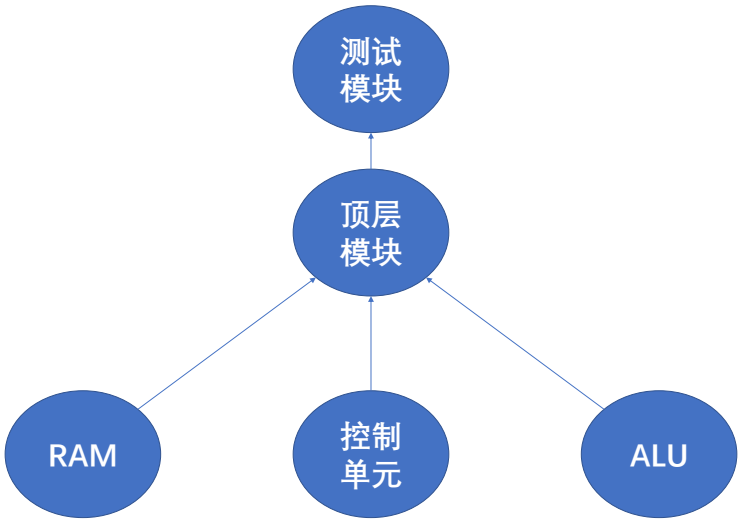

图 1: 各个模块之间关系

各个模块功能描述:

- **控制单元**, 系统的核心，完成取指令，指令译码，读写寄存器和 RAM。
- **算术逻辑单元**, 完成 8 位有符号数的加减运算。
- **随机存取存储器**, 存储指令和数据。
- **顶层模块**, 链接子模块。
- **测试模块**, 向顶层模块加入时钟和复位信号，完成任务。

1.2 指令集

定义的指令集 ISA(Instruction Set Architecture) 指令的 opcode 长度均为 4 位，在 RAM 中整体从低到高存储，指令和数据用 4bit 全 0(HALT 指令) 隔开。每条指令的低四位用来描述寄存器的 ID 或者 RAM 的地址，高四位为 opcode 描述执行的操作。每次取指令都是取 8 位。

表 1: 指令集架构

INSTRUCTION	OPCODE(BIN)	DESCRIPTION	ADDRESS OR REGISTERS
LOAD_A	0010	Read RAM Location into Register A(0)	4-bit RAM Address
LOAD_B	0001	Read RAM Location into Register B(1)	4-bit RAM Address
STORE_A	0100	Write from Register A(0) into RAM Location	4-bit RAM Address
ADD	1000	ADD 2 Registers, Store Value into Second Register	2-bit Register ID 2-bit Register ID
SUB	1001	Subtract 2 Registers, Store Value into Second Register	2-bit Register ID 2-bit Register ID
JUMP	1010	Update PC to New Address (i.e. Jump to Address)	4-bit RAM Address
JUMP_NEG	1011	If ALU Result was Negtive, Update PC to New Address	4-bit RAM Address
HALT	0000	Program Done,Halt Computer	No Address

还定义了宽度和最高有效位。实际 Verilog 的代码如下：

```
parameter HALT    = 4'b0000,
          LOAD_A  = 4'b0010,
          LOAD_B  = 4'b0001,
          STORE_A = 4'b0100,
          ADD     = 4'b1000,
          SUB     = 4'b1001,
          JUMP    = 4'b1010,
          JUMP_NEG = 4'b1011,
          WIDTH   = 8,
          MSB     = WIDTH - 1;
```

2 硬件组成

2.1 存储器

为了将更多注意力放在 CPU 的设计和实现过程上，存储器只使用 RAM。考虑到指令中 opcode 长度固定 4 位，而整体是 8 位处理器，那么 RAM 就有 4 位的寻址空间，地址从低到高为 0-15. 低地址存放指令，高地址存放数据，中间是 HALT 指令用于区分指令和数据。RAM 中每次存一套程序，共 3 套程序。

RAM 与其他模块的交互的总共有 4 个信号，8 位地址线，1 位写使能线和 1 位读使能线，8 位数据线。其中地址线和使能线为输入端口。为了节约线路，数据线为输入输出端口的时分复用，这样也就要求写使能线和读使能线不能同时为高电平。

使用的开源软件软件 iverilog 的仿真波形不支持数组变量的查询，所以额外定义了 wire 型变量 ram_13，便于仿真时验证程序的正确性。

RAM 模块的 Verilog 代码 ram.v:

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////
3  module ram(ram_data, ram_read_en, ram_write_en, ram_address);
4
5  input ram_read_en, ram_write_en;
6  input [3:0] ram_address;
7
8  inout [7:0] ram_data;
9
10 reg [7:0] ram[0:15];
11
12 //仅做仿真时观察波形使用: begin
13 wire [7:0] ram_13;
14 assign ram_13 = ram[13];
15 //仅做仿真时观察波形使用: end
16
17 // //第一个程序, 计算机速成课episode 7
18 // //3+14 = 17
19 // initial
20 //       begin
21 //               ram[0] = 8'b0010_1110; // LOAD_A 14
22 //               ram[1] = 8'b0001_1111; // LOAD_B 15
23 //               ram[2] = 8'b1000_0100; // ADD B A
24 //               ram[3] = 8'b0100_1101; // STORE_A 13
25 //               ram[4] = 8'b0000_0000;
26 //               ram[5] = 8'b0000_0000;
27 //               ram[6] = 8'b0000_0000;
28 //               ram[7] = 8'b0000_0000;
29 //               ram[8] = 8'b0000_0000;
```

```
30 //              ram[9] = 8'b0000_0000;
31 //              ram[10]= 8'b0000_0000;
32 //              ram[11]= 8'b0000_0000;
33 //              ram[12]= 8'b0000_0000;
34 //              ram[13]= 8'b0000_0000;
35 //              ram[14]= 8'b0000_0011; // 3
36 //              ram[15]= 8'b0000_1110; // 14
37 //      end
38
39 // //第二个程序，计算机速成课episode 8
40 // //1+1,2+1,...
41 // initial
42 //      begin
43 //              ram[0] = 8'b0010_1110; // LOAD_A 14
44 //              ram[1] = 8'b0001_1111; // LOAD_B 15
45 //              ram[2] = 8'b1000_0100; // ADD B A
46 //              ram[3] = 8'b0100_1101; // STORE_A 13
47 //              ram[4] = 8'b1010_0010; // JUMP 2
48 //              ram[5] = 8'b0000_0000;
49 //              ram[6] = 8'b0000_0000;
50 //              ram[7] = 8'b0000_0000;
51 //              ram[8] = 8'b0000_0000;
52 //              ram[9] = 8'b0000_0000;
53 //              ram[10]= 8'b0000_0000;
54 //              ram[11]= 8'b0000_0000;
55 //              ram[12]= 8'b0000_0000;
56 //              ram[13]= 8'b0000_0000;
57 //              ram[14]= 8'b0000_0001; // 1
58 //              ram[15]= 8'b0000_0001; // 1
59 //      end
60
61 // 第三个程序，计算机速成课episode 8
62 // 11 mod 5 == 1
63 initial
64         begin
65                 ram[0] = 8'b0010_1110; // LOAD_A 14
66                 ram[1] = 8'b0001_1111; // LOAD_B 15
67                 ram[2] = 8'b1001_0100; // SUB B A
68                 ram[3] = 8'b1011_0101; // JUMP_NEG 5
69                 ram[4] = 8'b1010_0010; // JUMP 2
70                 ram[5] = 8'b1000_0100; // ADD B A
71                 ram[6] = 8'b0100_1101; // STORE_A 13
72                 ram[7] = 8'b0000_0000; // HALT
73                 ram[8] = 8'b0000_0000;
74                 ram[9] = 8'b0000_0000;
```

```
                ram[10]= 8'b0000_0000;
                ram[11]= 8'b0000_0000;
                ram[12]= 8'b0000_0000;
                ram[13]= 8'b0000_0000;
                ram[14]= 8'b0000_1011; // 11
                ram[15]= 8'b0000_0101; // 5
        end

assign ram_data = (ram_read_en)? ram[ram_address]:8'hzz;        // read data from RAM

//尝试了always @(posedge ram_write_en) //失败，发现赋值结果为高阻态
always @(ram_write_en or ram_data) begin        // write data to RAM
        if (ram_write_en) ram[ram_address] <= ram_data;
end

endmodule
```

尝试使用了 always @(posedge ram_write_en) 失败，发现赋值结果为高阻态，猜测是由于 ram_data 是 inout 类型的，赋值过程中发生冲突。

2.2 CPU

2.2.1 控制单元

控制单元采用 (Mealy 型) 有限状态机 (Finite State Machine, FSM) 来实现。考虑到非阻塞赋值的特性，设置了共 3 个状态，其中 SO 状态下为取指令，S1 状态为指令译码 (对号入座) 和执行指令，S2 状态为写入寄存器或 RAM 以及准备复位。状态转移图：

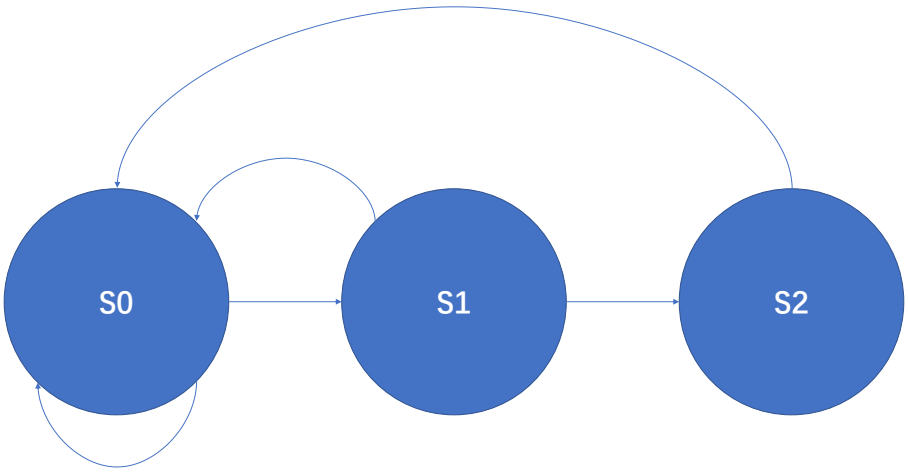

图 2: 控制单元状态转移图

除了状态寄存器外，PC 和通用寄存器也放在控制单元中。8 位指令除去 opcode 的 4 位后，由于 ADD、SUB 操作需要用到两个操作数，则 8 位寄存器只能由 2 位寻址，即只能有 4 个寄存器。(实际本实验只用到了 2 个寄存器就足够了。) 同样由于数组变量的波形信息无法在 iverilog 仿真时显示出来，所以定义了若干 wire 型变量便于观察通用寄存器情况。控制单元同时还定义了 ISA，便于指令译码。

控制单元和其他模块交互的端口信号有：时钟信号 clk，复位信号 rst，ram 读写使能信号，ram 地址线，ram 时分复用的数据线，alu 的两个操作数输入和一个操作数输出，alu 输出操作数是否为 0，是否为负，以及运算是否溢出。

控制单元的 Verilog 代码 control_unit.v:

```
1  `timescale 1ns / 1ps
2  module control_unit(clk, rst, ram_data, ram_read_en, ram_write_en, ram_address, opcode,
       alu_in_A, alu_in_B, alu_out, overflow, zero, negative);
3
4  input clk, rst, overflow, zero, negative ;              // clock, reset
5  input [7:0] alu_out;
6
7  output reg ram_read_en, ram_write_en;
8  output reg [7:0] alu_in_A, alu_in_B;
9  output reg [3:0] opcode;
10 output reg [3:0] ram_address;
11
12 // 相当于control_unit和ram之间的databus
13 inout [7:0] ram_data;
14
15
16 reg [7:0] Register[0:3];
17 reg [3:0] PC;
18 reg [2:0] state;
19
20 // 仅做仿真时观察波形使用: begin
21 wire [7:0] test_Register0, test_Register1, test_Register2, test_Register3;
22 assign test_Register0 = Register[0];
23 assign test_Register1 = Register[1];
24 assign test_Register2 = Register[2];
25 assign test_Register3 = Register[3];
26 // 仅做仿真时观察波形使用: end
27 // 有关仿真时数组再iverilog的显示问题, 参考了
28 // https://stackoverflow.com/questions/20317820/icarus-verilog-dump-memory-array-
       dumpvars
29 // 原因: iverilog 仿真时就是不能 dump 数组
30
31 // 相当于control_unit和ram之间的databus
32 assign ram_data = (ram_write_en)? Register[0]:8'hzz;
33
```

```
34  // instruction code
35  parameter HALT    = 4'b0000,
36          LOAD_A    = 4'b0010,
37          LOAD_B    = 4'b0001,
38          STORE_A   = 4'b0100,
39          ADD       = 4'b1000,
40          SUB       = 4'b1001,
41          JUMP      = 4'b1010,
42          JUMP_NEG  = 4'b1011,
43          WIDTH     = 8,
44          MSB       = WIDTH - 1;
45  // state encoding one-hot
46  parameter       S0 = 3'b001,
47                  S1 = 3'b010,
48                  S2 = 3'b100;
49
50  initial
51          begin
52                  state = S0;
53                  PC = 4'b0000;
54                  ram_address = 4'b0000;
55                  opcode = HALT;
56                  alu_in_A = 8'b0000_0000;
57                  alu_in_B = 8'b0000_0000;
58                  Register[0] = 8'b0000_0000;
59                  Register[1] = 8'b0000_0000;
60                  Register[2] = 8'b0000_0000;
61                  Register[3] = 8'b0000_0000;
62                  ram_read_en = 1'b0;
63                  ram_write_en = 1'b0;
64                  //通用寄存器初始化
65          end
66
67  always@(posedge clk)
68  begin
69          case(state)
70          // 取指令
71          S0: begin
72                  ram_address <= PC;
73                  ram_read_en <= 1'b1;
74                  ram_write_en<= 1'b0;
75                  opcode <= ram_data[7:4];
76                  if (opcode == HALT || rst) begin
77                          state <= S0;
78                  end else begin
```

```
79                      PC <= PC + 1;
80                      state <= S1;
81              end
82      end
83      // 指令译码(对号入座)和执行指令
84      S1: begin
85              if ((opcode == LOAD_A) || (opcode == LOAD_B)) begin
86                      ram_address <= ram_data[3:0];
87                      ram_read_en <= 1'b1;
88                      ram_write_en<= 1'b0;
89                      state <= S2;
90
91              end else if ((opcode == ADD) || (opcode == SUB)) begin
92                      alu_in_B <= Register[ram_data[3:2]];
93                      alu_in_A <= Register[ram_data[1:0]];
94                      state <= S2;
95
96              end else if (opcode == JUMP) begin
97                      PC <= ram_data[3:0];
98                      state <= S2;
99
100             end else if (opcode == JUMP_NEG) begin
101                     PC <= negative ? ram_data[3:0] : (PC);
102                     state <= S2;
103
104             end else if (opcode == STORE_A) begin
105                     ram_address <= ram_data[3:0];
106                     ram_read_en <= 1'b0;
107                     ram_write_en <= 1'b1;
108                     state <= S2;
109
110             end else begin
111                     state <= S0;
112             end
113
114
115     end
116     // 写入寄存器或RAM以及准备复位
117     S2: begin
118             case (opcode)
119                     LOAD_A: begin
120                             Register[0] <= ram_data[7:0];
121                             ram_address <= PC;
122                             state <= S0;
123                     end
```

```
                        LOAD_B: begin
                                Register[1] <= ram_data[7:0];
                                ram_address <= PC;
                                state <= S0;
                        end
                        ADD: begin
                                Register[ram_data[1:0]] <= alu_out;
                                ram_address <= PC;
                                state <= S0;
                        end
                        SUB: begin
                                Register[ram_data[1:0]] <= alu_out;
                                ram_address <= PC;
                                state <= S0;
                        end
                        STORE_A: begin
                                ram_read_en <= 1'b1;
                                ram_write_en <= 1'b0;
                                ram_address <= PC;
                                state <= S0;
                        end
                        JUMP: begin
                                ram_address <= PC;
                                state <= S0;
                        end
                        JUMP_NEG: begin
                                ram_address <= PC;
                                state <= S0;
                        end

                        default: state <= S0;
                endcase
        end

        default: begin
                state <= S0;
                ram_read_en  <= 1'b0;
                ram_write_en <= 1'b0;
                ram_address <= 4'b0000;
                opcode <= HALT;
        end
endcase
end
endmodule
```

2.2.2 算术逻辑单元

算术逻辑单元 ALU，主要计算 ADD 和 SUB 两种运算 (定义的指令集 ISA 其他指令没有用到 ALU)。ALU 按照 8 位有符号整型数进行运算，即运算范围为十进制-128 到 127。

具体加法运算为将操作数和输出都扩展一位后进行运算，保证内部运算的准确性。然后输出时舍去附加的最高位，然后判断结果是否为负，是否为 0，运算是否溢出。减法运算是将减数进行取反，然后进行加法运算。减数为 0 时单独处理。

算术逻辑单元和其他模块交互的端口信号有：来自控制单元的 opcode，两个 8 位操作数。输出信号有计算结果的输出，计算结果是否为负，是否为 0，运算是否溢出。

算术逻辑单元的 Verilog 代码 alu.v:

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////
3  module alu(opcode, alu_in_A, alu_in_B, alu_out, overflow, zero, negative);//
       arithmetic logic unit
4  input [3:0] opcode;
5  input [7:0] alu_in_A, alu_in_B;
6  output reg [7:0] alu_out;
7  output reg overflow;
8  output reg zero;
9  output reg negative;
10
11 reg extra;
12
13 parameter HALT    = 4'b0000,
14          LOAD_A   = 4'b0010,
15          LOAD_B   = 4'b0001,
16          STORE_A  = 4'b0100,
17          ADD      = 4'b1000,
18          SUB      = 4'b1001,
19          JUMP     = 4'b1010,
20          JUMP_NEG = 4'b1011,
21          WIDTH    = 8,
22          MSB      = WIDTH - 1;
23
24 initial
25          begin
26                  alu_out = 8'b00000000;
27                  overflow = 1'b0;
28                  zero = 1'b0;
29                  negative = 1'b0;
30                  extra = 1'b0;
31                  //通用寄存器初始化
32
33          end
```

```
always @(*) begin
                case (opcode)
                ADD:
                begin
                        {extra, alu_out} = {alu_in_A[MSB], alu_in_A} + {alu_in_B[MSB],
                            alu_in_B};

                        overflow  = ({extra, alu_out[MSB]} == 2'b01) || ({extra,
                            alu_out[MSB]} == 2'b10);

                        zero = alu_out ? 0 : 1;

                        negative = alu_out[MSB];
                end
                SUB:
                begin
                        if (!alu_in_B) begin
                                alu_out = alu_in_A;
                                overflow  = 1'b0;
                                zero = alu_out ? 0 : 1;
                                negative = alu_out[MSB];
                        end else begin
                                {extra, alu_out} = {alu_in_A[MSB], alu_in_A} + ~{
                                    alu_in_B[MSB], alu_in_B} + 9'b000000001;
                                overflow  = ({extra, alu_out[MSB]} == 2'b01) || ({extra
                                    , alu_out[MSB]} == 2'b10);
                                zero = alu_out ? 0 : 1;
                                negative = alu_out[MSB];
                        end
                end

                default:        alu_out = 8'bzzzz_zzzz;
                endcase
end

endmodule
```

2.3 顶层设计

模块化编程的顶层设计 Verilog 代码 core.v:

```
`timescale 1ns / 1ps
module core(clk, rst);  // Top-level entity(except core-tb)
input clk, rst;

wire ram_read_en, ram_write_en;
wire overflow, zero, negative;

wire[7:0] ram_data;
wire[7:0] alu_in_A, alu_in_B, alu_out;
wire[3:0] opcode, ram_address;

//module alu(opcode, alu_in_A, alu_in_B, alu_out, overflow, zero, negative);
alu ALU(.opcode(opcode), .alu_in_A(alu_in_A), .alu_in_B(alu_in_B), .alu_out(alu_out), .
    overflow(overflow), .zero(zero), .negative(negative));

//module ram(ram_data, ram_read_en, ram_write_en, ram_address);
ram RAM(.ram_data(ram_data), .ram_read_en(ram_read_en), .ram_write_en(ram_write_en), .
    ram_address(ram_address));


//module control_unit(clk, rst, ram_data, ram_read_en, ram_write_en, ram_address,
    opcode, alu_in_A, alu_in_B, alu_out, overflow, zero, negative);
control_unit CONTROL_UNIT(.clk(clk),
                                        .rst(rst),
                                        .ram_data(ram_data),
                                        .ram_read_en(ram_read_en),
                                        .ram_write_en(ram_write_en),
                                        .ram_address(ram_address),
                                        .opcode(opcode),
                                        .alu_in_A(alu_in_A),
                                        .alu_in_B(alu_in_B),
                                        .alu_out(alu_out),
                                        .overflow(overflow),
                                        .zero(zero),
                                        .negative(negative)
                                        );


endmodule
```

根据顶成设计可知总体内部结构，见附录B。(点击可直接跳转)

2.4 测试平台

测试平台相当于加入了时钟信号 clk 和复位信号 rst。由于测试平台中将顶层设计模块实例化，那么逻辑上测试平台其实位于最顶层。测试平台的 Verilog 代码 core_testbench.v:

```
`timescale 1ps / 1ps
module core_testbench;

reg rst;
reg clk;

initial
        begin
    $dumpfile("test.vcd");
    $dumpvars(0,core1);
        end

initial
  begin
    clk <= 1'b0;
    # 150 ;
  repeat(9999)
  begin
    clk  = 1'b1;
    #50  clk  = 1'b0;
    #50  ;
  end
  clk  = 1'b1;
  # 50 ;
  end
initial
  begin
          rst  = 1'b1  ;
          # 100;
          rst=1'b0;
          # 9000 ;
  end
core core1(.clk(clk),.rst(rst));

initial
#1000000 $finish;
endmodule
```

3 实验结果

由于 S0 在取完指令后，PC 的值直接就加一，因此仿真时的波形 PC 的值应该比当前执行指令的 PC 值大 1。这么设计虽然直觉上不太合适，但是确实省下了一级流水线，使得问题简单化。至于 PC 的值为什么不在后两个状态改变，理由是放在 S1 会影响到跳转指令的执行，而放在 S2 时则由于非阻塞赋值并行的特性，PC 值只能在下一个状态才能传递给 ram 地址线，如果在 S0 传递给 ram 地址线又会影响下一次的取指令。于是 PC 加一放在了 S0 状态，这里 PC 波形信号的值是一种设定而非故障。

3.1 程序一：两数相加

将 RAM 初始化为第一个程序 (见 ram.v), 得到仿真波形的截屏如下：

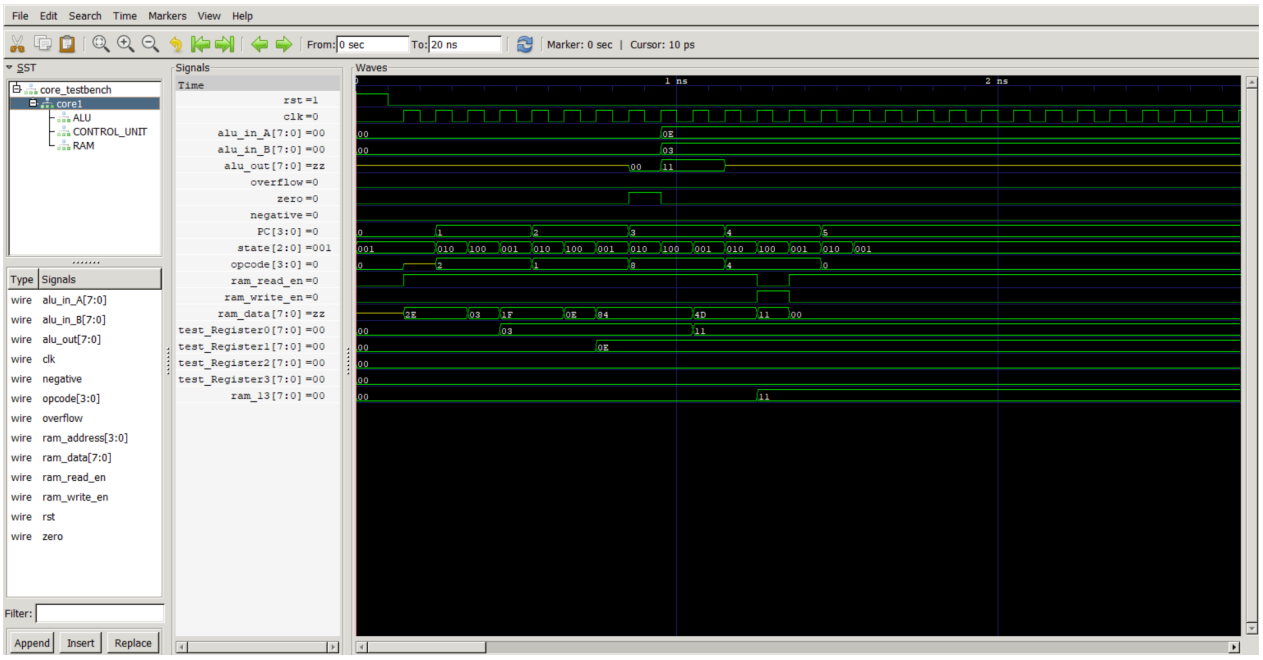

图 3: 程序一 3 + 14 = 17 仿真波形

注意: 因为波形标注的数字是十六进制格式，因而最终得到的结果是十六进制的'11', 对应十进制的'17'。从图中还可以看出 ram_data 端口的时分复用情况，相比于将 ram 的输入数据和输出数据分开，节约了 8 根线。

3.2 程序二：无限加一

从累加初始值 1 无限加一，由于 ALU 是执行有符号数的运算，那么结果会从 1 到 127(8'b0111_1111)，然后溢出到-128(8'b1000_0000)，再到 1 循环不止。

将 RAM 初始化为第二个程序 (见 ram.v), 得到仿真波形的截屏如下：

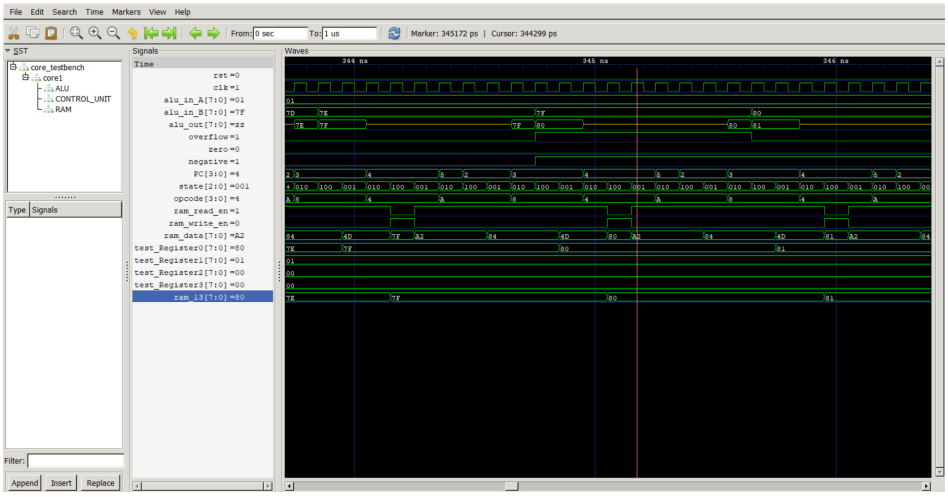

图 4: 程序二无限加一 ALU 运算溢出时的仿真波形

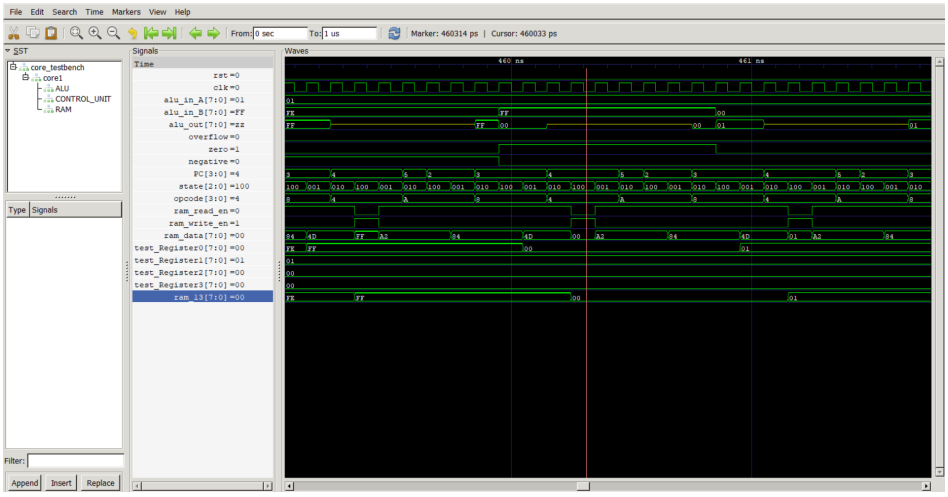

图 5: 程序二无限加一 ALU 输出为 0 时的仿真波形

3.3 程序三：求余数

将 RAM 初始化为第三个程序 (见 ram.v), 得到仿真波形的截屏如下：

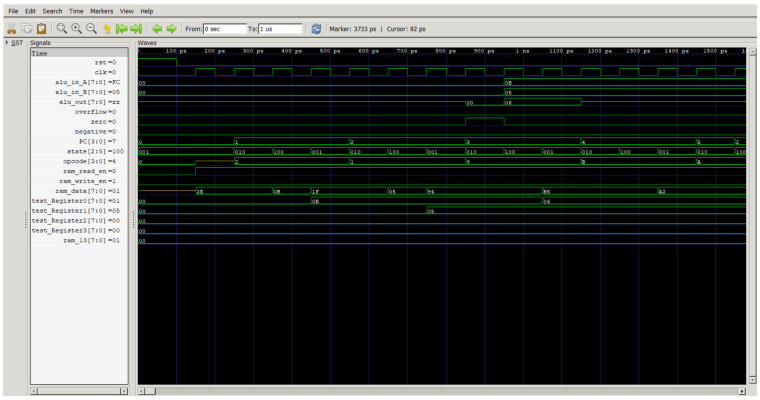

图 6: 程序三 11 mod 5 = 1 的仿真波形 1

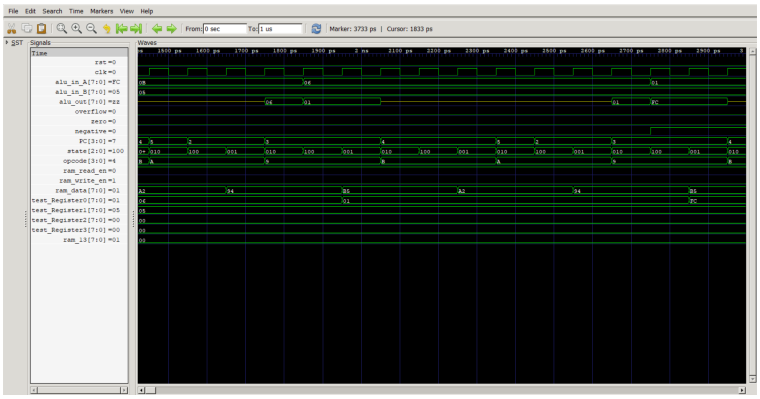

图 7: 程序三 11 mod 5 = 1 的仿真波形 2

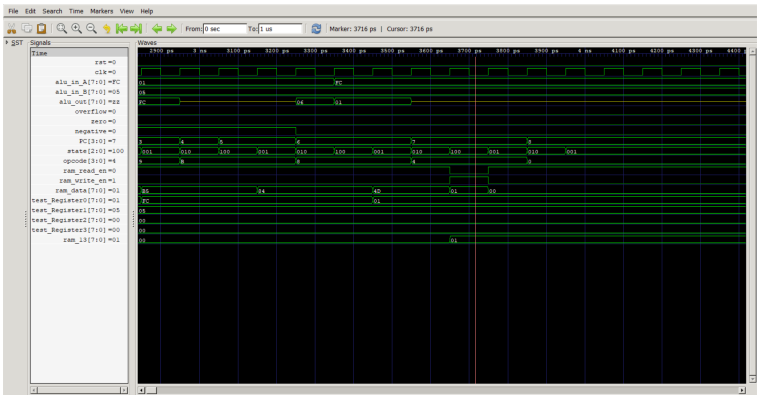

图 8: 程序三 11 mod 5 = 1 的仿真波形 3

4 实验心得

由于本人事先没有学过 Verilog 等硬件描述语言，更多的是现学现用而且是速成，很多语法还不是很熟悉，所以写的程序比较死板，但完全遵循 KISS(Keep it Simple and Stupid) 原则。原本想把通用寄存器作为一个单独的 module 来写，但控制单元的连线已经很复杂，端口也很多，尝试用 inout 端口时分复用来减少连线，但 inout 端口用得并不熟悉，因为 inout 端口只能是 wire 型变量，使用时赋值不当出了很多问题，在上边花了很多时间。

一开始就采用分模块设计的思想，每个模块写玩之后只单独编译，直到编译通过为止 (但是不知道能否真的实现功能)。在把所有模块写好之后，通过改 RAM 中的指令，然后观察仿真波形，找出是哪个模块出了问题，然后继续 debug。

全部代码已经共享在 github:

https://github.com/Eigenterm/8-bit-CPU-Verilog

此代码的结构参考了:

https://github.com/liuqidev/8-bits-RISC-CPU-Verilog

代码的 ISA 参考了 youtube 计算机速成课:

Instructions & Programs: Crash Course Computer Science #8

A 附录：windows 下仿真说明

本实验在 windows10 操作系统下使用 iverilog+gtkwave 环境仿真。添加到环境变量后，在命令行下进入所有源代码所在路径，以本此实验为例，依次执行：

```
cd E:\code\Verilog\JMC
iverilog -y E:/code/Verilog/JMC -o test.out core.v core_testbench.v
vvp test.out
gtkwave test.vcd
```

即可编译并仿真全部源码。命令行记录已经整理并上传至:

https://github.com/Eigenterm/8-bit-CPU-Verilog

B 附录：内部结构 RTL

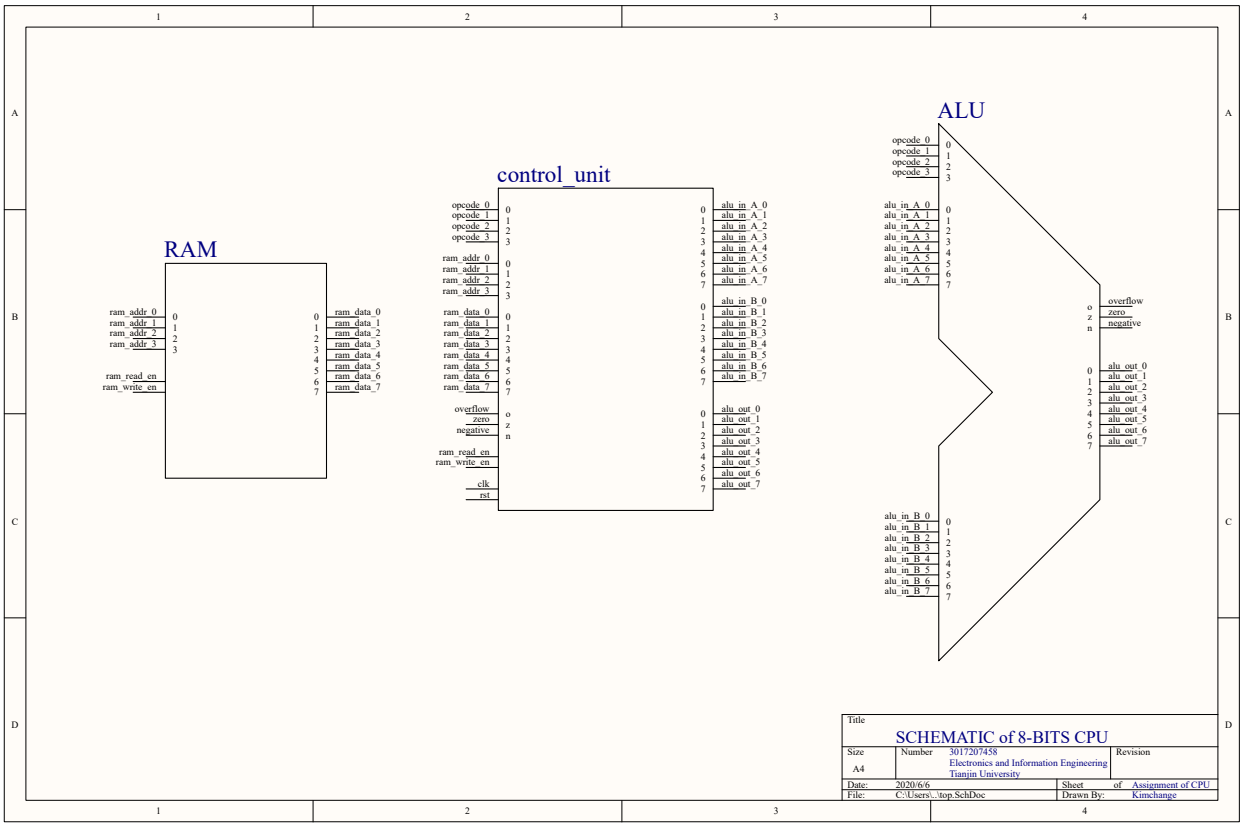

图 9: 总体内部结构

Schdoc 文件与 Schlib 文件同样上传至:
https://github.com/Eigenterm/8-bit-CPU-Verilog