

# Lab 6 综合设计

姓名:张芷苒 学号: PB21081601 实验日期: 2023-6-14

## 1 实验题目

综合设计

## 2 实验目的

- 在流水线 CPU 上针对 RV32I 指令集进行扩展, 使 CPU 支持 RV32I 中除 FENCE, CSR 与环境调用和断点指令外的全部 37 条指令.
- 在流水线 CPU 上增加了分支预测模块, 实现 2-bit 动态分支预测.

## 3 实验平台

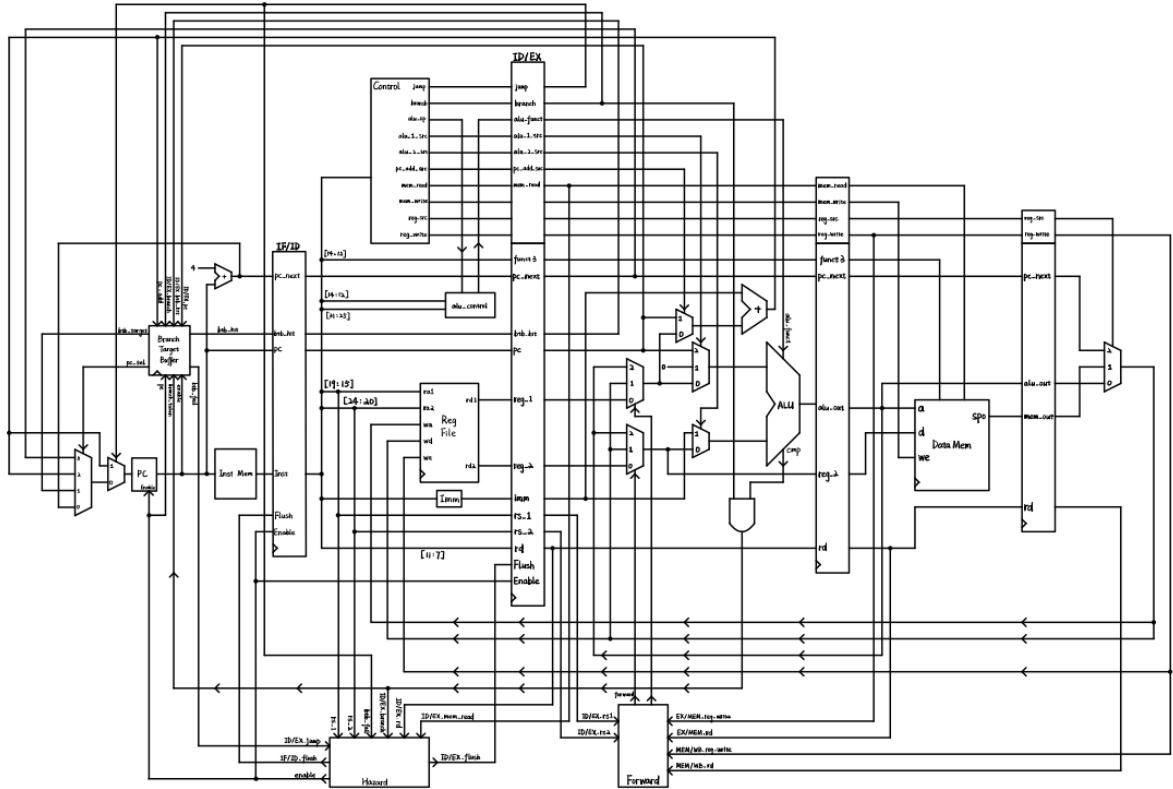
- Xilinx Vivado v2019.1
- Microsoft Visual Studio Code
- FPGAOL

## 4 实验过程

整个项目结构如下

```
1 main (Main.v)
2 |— PDU: pdu (PDU.v)
3 |— CPU: cpu (CPU.v)
4 |   |— Inst_Mem: rom (rom.xci)
5 |   |— Data_Mem: data_mem (Data_Mem.v)
6 |   |   |— Data_Mem: dual_port_ram (dual_port_ram.xci)
7 |   |— Control: control (Ctrl.v)
8 |   |— ALU_control: alu_control (ALU_Ctrl.v)
9 |   |— Reg_File: regfile (Reg_File.v)
10 |   |— Imm_Gen: imm_gen (Imm_Gen.v)
11 |   |— ALU: alu (ALU.v)
12 |   |— Forward: forward (forward.v)
13 |   |— Hazard: hazard (hazard.v)
14 |   |— BTB: btb (BTB.v)
15 |   |— IF_ID: if_id (IF_ID.v)
16 |   |— ID_EX: id_ex (ID_EX.v)
17 |   |— EX_MEM: ex_mem (EX_MEM.v)
18 |   |— MEM_WB: mem_wb (MEM_WB.v)
```

CPU 的数据通路如下, 其中没有画出 I/O 端口.



### 4.1 扩展指令集

针对指令集扩展, CPU 主要在 ID, EX, MEM 段进行修改.

#### 4.1.1 ID 段

在 ID 段中, 修改有两部分

- 对 control 模块的输出信号进行了一些修改
- 增加了 alu\_control 模块以生成 alu\_funct 信号, 控制 EX 段 ALU 的行为.

##### 4.1.1.1 control 模块

在 control 模块中, 输出信号有如下改动

- 增加了 alu\_1\_src 信号, 用于控制 EX 段 ALU 的第一个操作数的来源. 当指令为 auipc 或 lui 时, 分别输出 10 和 01
- 原来的 alu\_src 重命名为 alu\_2\_src
- 增加了 pc\_add\_src 信号, 控制 PC 加法器的 PC 来源 (pc / reg\_1), 当指令为 jalr 时, 输出 1.
- alu\_op 信号不再直接控制 ALU, 而是作为 alu\_control 的输入, 指示当前指令的格式. (branch/imm/reg/其他)

改动后的 control 模块输出信号如下表

输出信号	位数	含义
jump	1	是否跳转指令
branch	1	是否分支指令
alu_op	2	alu_control 的输入信号
alu_1_src	2	ALU 第一操作数选择信号
alu_2_src	1	ALU 第二操作数选择信号
pc_add_src	1	PC 加法器的 PC 选择信号
mem_read	1	指令是否需要读取内存
mem_write	1	指令是否需要写入内存
reg_src	2	寄存器写回值选择信号
reg_write	1	指令是否需要写回寄存器

具体的 Verilog 代码见[附录: Ctrl.v](#)

#### 4.1.1.2 alu\_control 模块

alu\_control 模块接受 `alu_op`, `funct3` (`inst[14:12]`) 和 `funct7` (`inst[30]`) 信号, 输出 `alu_funcnt` 信号, 控制 ALU 的行为. 其中 `alu_funcnt` 信号的低 3 位为 `funct3`, 第四位视情况而定, 具体如下

alu_funcnt	算术操作	比较操作
4'b0000	加	等于
4'b0001	逻辑左移	不等于
4'b0010	小于时置位	-
4'b0011	小于时置位 (无符号)	-
4'b0100	按位异或	小于
4'b0101	逻辑右移	大于等于
4'b0110	按位或	小于 (无符号)
4'b0111	按位与	大于等于 (无符号)
4'b1000	减	-
4'b1101	算术右移	-
4'b1111	加	-

具体的 Verilog 代码见[附录: ALU\\_ctrl.v](#)

#### 4.1.2 EX 段

EX 段的修改主要有

- 完善了 ALU 的功能, 与上文中的 `alu_funcnt` 信号对应, 具体可见[附录: ALU.v](#)
- 在 EX 段的数据通路中增加了两个 MUX, 如下
  - ALU 的第一个操作数前增加了一个 MUX, 选择信号为 `IDEX_alu_1_src`, 在 `EX_reg_1` (前递后的 `reg_1`), `0`, `IDEX_pc` 之间选择, 以实现 `lui`, `auipc` 指令, 如下

```

1  always @(*) begin
2      case (IDEX_alu_1_src)
3          2'b00: alu_in_1 = EX_reg_1;
4          2'b01: alu_in_1 = 32'h0;
5          2'b10: alu_in_1 = IDEX_pc;
6          default: alu_in_1 = 32'h0;
7      endcase
8  end

```

- PC 加法器前增加了一个 MUX, 选择信号为 `IDEX_pc_add_src`, 在 `IDEX_pc` 和 `EX_reg_1` 之间选择, 以实现 `jalr` 指令, 如下

```

1  assign pc_add = (IDEX_pc_add_src ? EX_reg_1 : IDEX_pc) +
    IDEX_imm;

```

### 4.1.3 MEM 段

MEM 段主要对数据存储器进行修改, 使其能够进行非对齐存取, 以实现 `load/store` 指令.

- 若需要读取数据, 首先将对应的字取出, 根据 `funct3` 信号判断需要取得的数据长度, 并根据地址取对应的数据段, 如下

```

1  always @(*) begin
2      if (mem_read) begin
3          case (funct3)
4              3'b000:
5                  case (addr[1:0])
6                      2'b00: mem_data = {{24{data_out[7]}}},
7                          data_out[7:0]};
8                      2'b01: mem_data = {{24{data_out[15]}}},
9                          data_out[15:8]};
10                     2'b10: mem_data = {{24{data_out[23]}}},
11                         data_out[23:16]};
12                     2'b11: mem_data = {{24{data_out[31]}}},
13                         data_out[31:24]};
14                 endcase
15             3'b001:
16                 case (addr[1])
17                     1'b0: mem_data = {{16{data_out[15]}}},
18                         data_out[15:0]};
19                     1'b1: mem_data = {{16{data_out[31]}}},
20                         data_out[31:16]};
21                 endcase
22             3'b010: mem_data = data_out;
23             3'b100:
24                 case (addr[1:0])
25                     2'b00: mem_data = {24'h0, data_out[7:0]};
26                     2'b01: mem_data = {24'h0, data_out[15:8]};
27                     2'b10: mem_data = {24'h0, data_out[23:16]};

```

```

22         2'b11: mem_data = {24'h0, data_out[31:24]};
23     endcase
24     3'b101:
25     case (addr[1])
26         1'b0: mem_data = {16'h0, data_out[15:0]};
27         1'b1: mem_data = {16'h0, data_out[31:16]};
28     endcase
29     default: mem_data = data_out;
30 endcase
31 end
32 else begin
33     mem_data = data_out;
34 end
35 end

```

- 若需要存数据, 同样地需要先将对应的字取出, 根据 `funct3` 信号判断需要存的数据长度, 再根据地址替换取得字的对应段, 如下

```

1  always @(*) begin
2      if (mem_we) begin
3          case(funct3)
4              3'b000:
5                  case(addr[1:0])
6                      2'b00: data_in = {data_out[31:8], data[7:0]};
7                      2'b01: data_in = {data_out[31:16], data[7:0],
data_out[7:0]};
8                      2'b10: data_in = {data_out[31:24], data[7:0],
data_out[15:0]};
9                      2'b11: data_in = {data[7:0], data_out[23:0]};
10                 endcase
11             3'b001:
12             case (addr[1])
13                 1'b0: data_in = {data_out[31:16], data[15:0]};
14                 1'b1: data_in = {data[15:0], data_out[15:0]};
15             endcase
16             3'b010: data_in = data;
17             default: data_in = data_out;
18         endcase
19     end
20     else begin
21         data_in = data_out;
22     end
23 end

```

## 4.2 分支预测

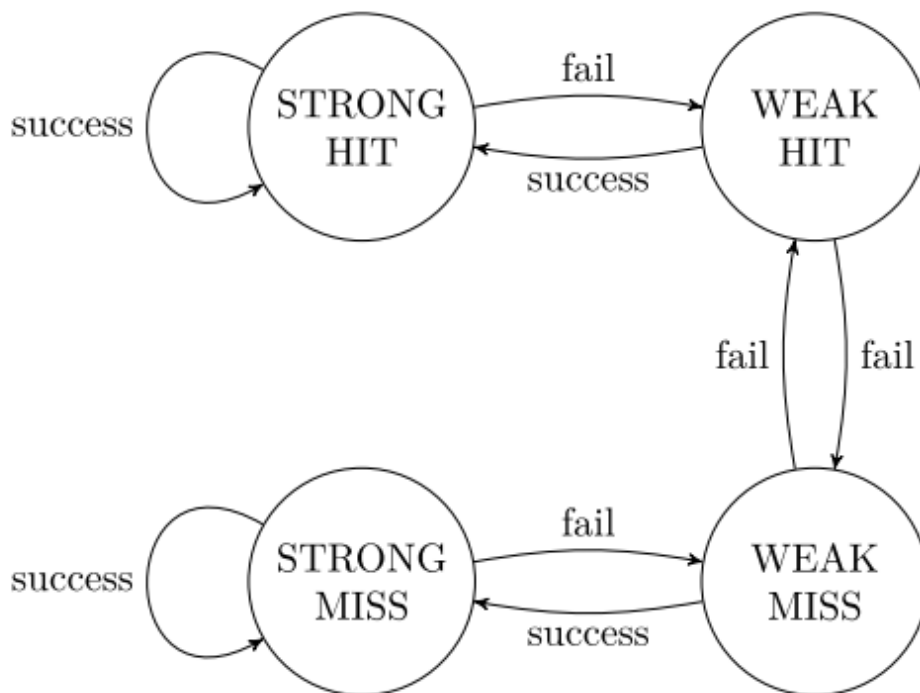
本次实验在 IF 段增加了一个分支预测模块, 其中有一个 cache 存储分支指令的相关信息, 每个 cache 数据块有四个部分, 分别为:

- 标签 tag: 用于判断传入的 pc 是否与该数据块的 pc 匹配
- 目标 target: 用于存放分支指令的目标 pc
- 有效位 valid: 用于判断该数据块是否有效 (被启用)
- 状态 state: 指示该分支指令的预测状态

其 Verilog 代码部分如下:

```
1 // Buffer Register
2 reg [TAG_LEN - 1 : 0] tag    [BUF_SIZE - 1 : 0];
3 reg [31:0]             target [BUF_SIZE - 1 : 0];
4 reg                   valid   [BUF_SIZE - 1 : 0];
5 reg [1:0]              state  [BUF_SIZE - 1 : 0];
```

其中 state 是两位的, 有四个状态, 根据上一次预测是否成功进行更新, 状态转移图如下



HIT 代表预测跳转, MISS 代表预测不跳转.

当 cache 数据块标签与传入的 pc 标签段匹配时, 若该数据块有效, 且 state 为 HIT, 则预测跳转, 并输出 target 作为下一个 pc 值, 否则预测不跳转, 使用  $pc + 4$  作为下一个 pc 值. 这部分的 Verilog 代码如下

```
1 // Generate Target
2 always @(*) begin
3     if (rst) begin
4         btb_hit = 1'b0;
```

```

5         btb_target = 32'h0;
6     end
7     else if (valid[pc_idx] &&
8             (pc_tag == tag[pc_idx]) &&
9             state[pc_idx][1]) begin
10         btb_hit      = 1'b1;
11         btb_target = target[pc_idx];
12     end
13     else begin
14         btb_hit      = 1'b0;
15         btb_target = 32'h0;
16     end
17 end

```

在 EX 段, 需要根据 IF 段的预测结果和实际需要跳转情况来对 cache 进行更新

- 如果对应 cache 数据块标签段与 EX 段 pc 标签段匹配, 则根据预测是否成功, 按照上面的状态机对 state 段进行更新
- 如果对应 cache 数据块无效或标签不匹配, 则直接将整个数据段替换掉, 默认 state 为 WEAK\_MISS

这一部分的 Verilog 代码如下

```

1  integer i = 0;
2  always @(posedge clk or posedge rst) begin
3      if (rst) begin
4          for (i = 0; i < BUF_SIZE; i = i + 1) begin
5              tag[i]      ≤ 0;
6              target[i] ≤ 32'h0;
7              valid[i]   ≤ 1'b0;
8              state[i]   ≤ WEAK_MISS;
9          end
10     end
11     else if (enable && IDEX_branch) begin
12         // tag matched, update state
13         if ((tag[IDEX_pc_idx] == IDEX_pc_tag) && valid[IDEX_pc_idx])
14         begin
15             case (state[IDEX_pc_idx])
16             STRONG_HIT:
17                 state[IDEX_pc_idx] ≤ btb_fail ? WEAK_HIT :
18                 STRONG_HIT;
19             WEAK_HIT:
20                 state[IDEX_pc_idx] ≤ btb_fail ? WEAK_MISS :
21                 STRONG_HIT;
22             WEAK_MISS:
23                 state[IDEX_pc_idx] ≤ btb_fail ? WEAK_HIT :
24                 STRONG_MISS;
25             STRONG_MISS:

```

```

22         state[IDEX_pc_idx] ≤ btb_fail ? WEAK_MISS :
STRONG_MISS;
23     endcase
24 end
25 // tag not matched, change buffer
26 else begin
27     tag[IDEX_pc_idx] ≤ IDEX_pc_tag;
28     target[IDEX_pc_idx] ≤ IDEX_branch_target;
29     valid[IDEX_pc_idx] ≤ 1'b1;
30     state[IDEX_pc_idx] ≤ WEAK_MISS;
31 end
32 end
33 end

```

预测模块同时还要生成 pc 的选择信号, 以应对预测跳转/不跳转和预测失败的情况, 分如下四种情况

- 预测失败, IF 段预测跳转但 EX 段计算出不跳转, 选择 EX 段  $pc + 4$
- 预测失败, IF 段预测不跳转但 EX 段计算出跳转, 选择 EX 段计算出的跳转地址
- 预测跳转, 选择 cache 中的目标
- 预测不跳转, 选择  $pc + 4$

这一部分的 Verilog 代码如下

```

1  reg btb_p_nt; // branch predicted but not taken
2  reg btb_np_t; // not branch predicted but taken
3
4  assign btb_fail = btb_p_nt | btb_np_t;
5
6  always @(*) begin
7      if (rst) begin
8          btb_p_nt = 1'b0;
9          btb_np_t = 1'b0;
10     end
11     else begin
12         btb_p_nt = (IDEX_btb_hit) & (~branch_taken);
13         btb_np_t = (~IDEX_btb_hit) & (branch_taken);
14     end
15 end
16 // PC Controller
17 always @(*) begin
18     if (btb_p_nt) pc_sel ≤ 2'b11; // IDEX_pc_next
19     else if (btb_np_t) pc_sel ≤ 2'b10; // pc_add
20     else if (btb_hit) pc_sel ≤ 2'b01; // btb_target
21     else pc_sel ≤ 2'b00; // pc_next
22 end

```



最后, 这个预测模块中还有一个计数单元, 用于记录预测成功/失败次数以及分支指令执行数, 这些数据会直接连接到 PDU, 替换原先 ctrl 系列信号, 以便在 FPGAOL 上显示出来. Verilog 代码如下

```
1 // Counter Part
2 always @(posedge clk or posedge rst) begin
3     if (rst) begin
4         branch_cnt    ≤ 32'h0;
5         btb_succ_cnt ≤ 32'h0;
6         btb_fail_cnt ≤ 32'h0;
7     end
8     else begin
9         if (IDEX_branch) begin
10            branch_cnt    ≤ branch_cnt    + 32'h1;
11            if (btb_fail) btb_fail_cnt ≤ btb_fail_cnt + 32'h1;
12            else          btb_succ_cnt ≤ btb_succ_cnt + 32'h1;
13        end
14    end
15 end
16 end
```

整个分支预测模块的代码见[附录 BTB.v](#).

为了实现分支预测, 还要对数据通路进行少量修改

- pc 前需要增加一个 MUX, 若 EX 为跳转指令, 则选择跳转地址传入 pc, 否则选择由上一个 MUX (由分支预测模块输出的选择信号控制) 选择出的 pc, 代码如下

```
1 always @(*) begin
2     if (IDEX_jump) pc_in = pc_add;
3     else begin
4         case (pc_sel)
5             2'b11: pc_in = IDEX_pc_next;
6             2'b10: pc_in = pc_add;
7             2'b01: pc_in = btb_target;
8             2'b00: pc_in = pc_next;
9         endcase
10    end
11 end
```

- hazard 模块的输入 `pc_src` 现在被拆分为 `btb_fail` 和 `IDEX_jump`, 二者任意一个有效就会冲刷流水线.

## 5 实验结果

### 5.1 测试指令集

为了测试指令集, 我编写了如下的 RISC-V 汇编代码, 其中不同指令写回的寄存器均不同, 若所有指令都正确执行, 则寄存器值应该与注释中相同, 汇编如下, 在执行到 `jal x0, start` 指令后会重新跳回第一条继续执行, 以测试分支预测模块

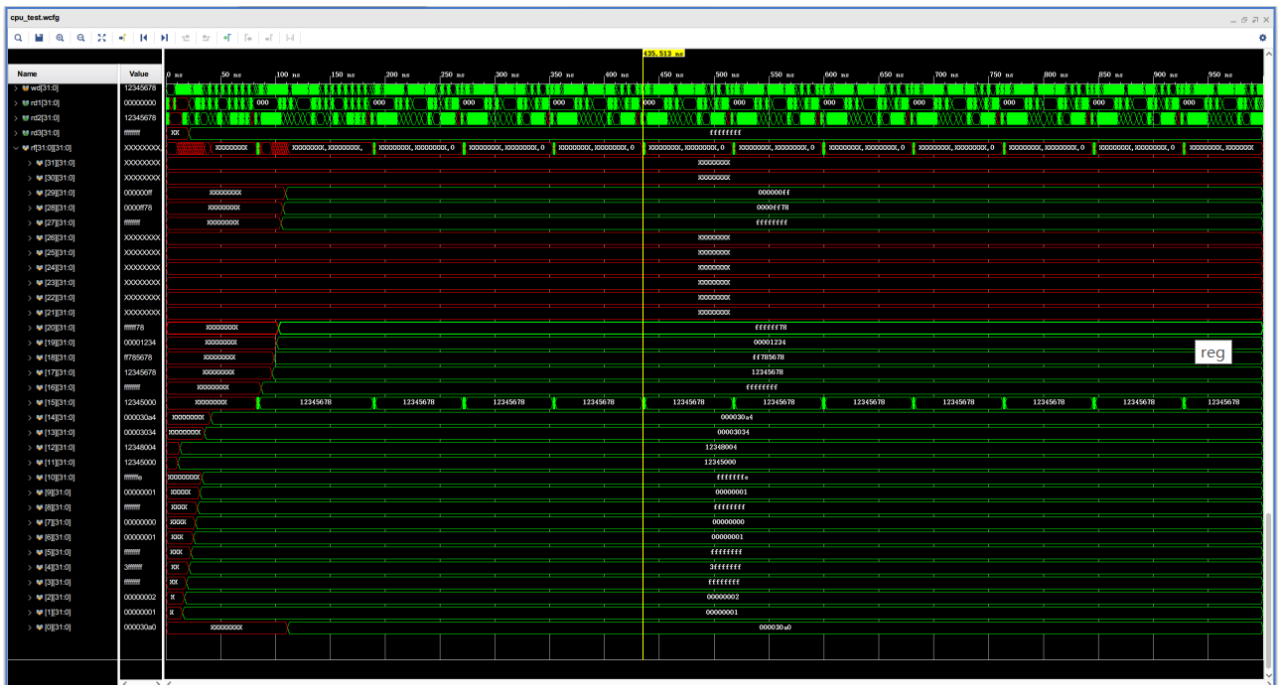
```
1  # begins at 0x3000
2  start:
3      # test for lui & auipc
4      lui    x11, 0x12345 # x11 = 0x12345000 305,418,240
5      auipc  x12, 0x12345 # x12 = 0x12348004 305,430,532
6
7      # test for arithmetic & logical operations
8      addi   x1,  x0, 1 # x1 = 0x00000001
9      slli   x2,  x1, 1 # x2 = 0x00000002
10     sub    x3,  x1, x2 # x3 = 0xFFFFFFFF
11     srl    x4,  x3, x2 # x4 = 0x3FFFFFFF
12     srai   x5,  x3, 2 # x5 = 0xFFFFFFFF
13     slt    x6,  x5, x4 # x6 = 0x00000001
14     sltu   x7,  x5, x4 # x7 = 0x00000000
15     or     x8,  x5, x1 # x8 = 0xFFFFFFFF
16     and    x9,  x5, x1 # x9 = 0x00000001
17     xor    x10, x5, x1 # x10 = 0xFFFFFFFFE
18
19     # test for jal & jalr
20     jal x13, jal_test # x13 = 0x00003034
21
22     # test for branch
23     beq    x1, x1, bne_test
24     addi   x21, x0, 1 # if beq failed, x21 = 1
25 bne_test:
26     bne    x1, x0, blt_test
27     addi   x22, x0, 1 # if bne failed, x22 = 1
28 blt_test:
29     blt    x0, x1, bge_test
30     addi   x23, x0, 1 # if blt failed, x23 = 1
31 bge_test:
32     bge    x1, x0, bltu_test
33     addi   x24, x0, 1 # if bge failed, x24 = 1
34 bltu_test:
35     bltu   x4, x3, bgeu_test
36     addi   x25, x0, 1 # if bltu failed, x25 = 1
37 bgeu_test:
38     bgeu   x3, x4, ls_test
39     addi   x26, x0, 1 # if bgeu failed, x26 = 1
40
```

```

41 # test for load & store
42 ls_test:
43     li    x15, 0x12345678
44     li    x16, 0xFFFFFFFF
45     sw    x15, 0x000(x0) # mem[0] = 0x12345678
46     sh    x15, 0x004(x0)
47     sb    x15, 0x006(x0)
48     sb    x16, 0x007(x0) # mem[1] = 0xFF785678
49     lw    x17, 0x000(x0) # x17 = 0x12345678
50     lw    x18, 0x004(x0) # x18 = 0xFF785678
51     lh    x19, 0x002(x0) # x19 = 0x00001234
52     lh    x20, 0x006(x0) # x20 = 0xFFFFFFFF
53     lb    x27, 0x007(x0) # x27 = 0xFFFFFFFF
54     lhu   x28, 0x006(x0) # x28 = 0x0000FF78
55     lbu   x29, 0x007(x0) # x29 = 0x000000FF
56
57 # test for btb
58     jal   x0, start
59
60 jal_test:
61     jalr  x14, 0(x13) # x14 = 0x000030A4

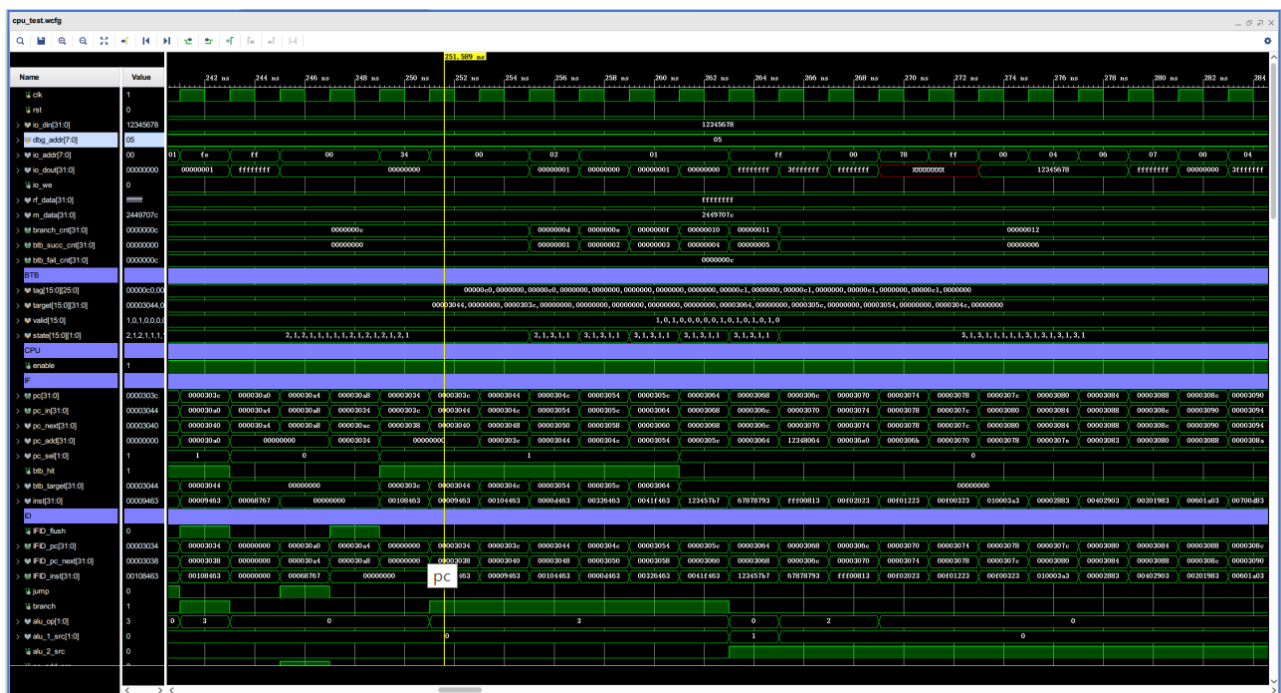
```

其中所有的分支跳转指令都会跳转, 因此 `x21` ~ `x26` 的赋值指令不会执行. 在 Vivado 中进行模拟, 寄存器堆如下, 可以看到所有的寄存器值都是正确的.

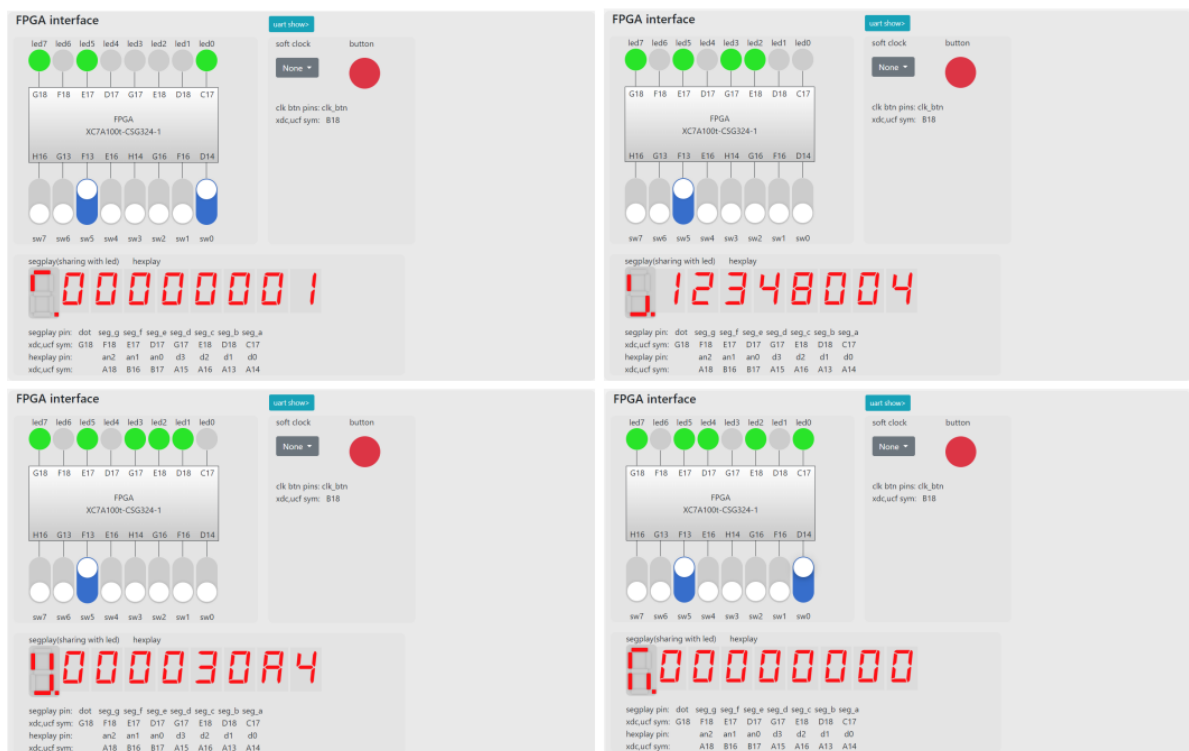


而观察 pc 部分如下, 可以看到进行分支预测 (`btb_hit` 有效) 时, pc 并不是每次都加 4, 而且没有冲刷流水线的痕迹, 观察计数器, 预测失败的次数在最后稳定到了 `0x0000000c` (12), 与预期也相同: 六条分支跳转指令, 第一轮添加到 cache, 默认状态为 WEAK\_MISS, 所以第二轮预测不跳转, 共 12

次失败预测。



生成 bit 流烧写到 FPGAOL 平台上, 执行结果与模拟相同, 这里放部分截图, 其中最后三张分别为分支指令执行行数, 预测成功数, 预测失败数。





## 5.2 测试分支预测

为了测试分支预测, 我编写了如下的冒泡排序程序, 对数据存储器中的 256 个字进行升序排序, 排序结束不停执行 `jal` 指令. 在 FPGAOL 平台上, 排序结束后数码管显示会清零, 以显示排序结束. 汇编程序如下

```

1  # memory starts at 0x000, ends at 0x3FF
2      li    t0, 1024    # t0 represents for size
3      add   x1, x0, x0  # x1 = 0, represents for i
4
5  loop1:
6      mv     x2, x1      # x2 = i, represents for j
7      lw     t1, 0(x1)   # t1 = mem[i]
8
9  loop2:
10     lw     t2, 0(x2)    # t2 = mem[j]
11     bge    t2, t1, loop2_end # if mem[i] ≥ mem[j], jump to loop2_end
12     sw     t1, 0(x2)    # mem[j] = mem[i]
13     sw     t2, 0(x1)    # mem[i] = mem[j]
14     add    t1, x0, t2   # t1 = new mem[i]
15 loop2_end:
16     addi   x2, x2, 4    # j++
17     blt    x2, t0, loop2
18
19     addi   x1, x1, 4    # i++
20     blt    x1, t0, loop1
21
22     # Get the counters

```

```

23      li    s4, 0x800
24      lw    s0, 0(s4)    # s0 = branch_count
25      lw    s1, 4(s4)    # s1 = success_count
26      lw    s2, 8(s4)    # s2 = fail_count
27
28      # Signal the end of the program
29      sw    x0, 0x408(x0) # segplay echoes 0
30
31  end:
32      jal   x0, end

```

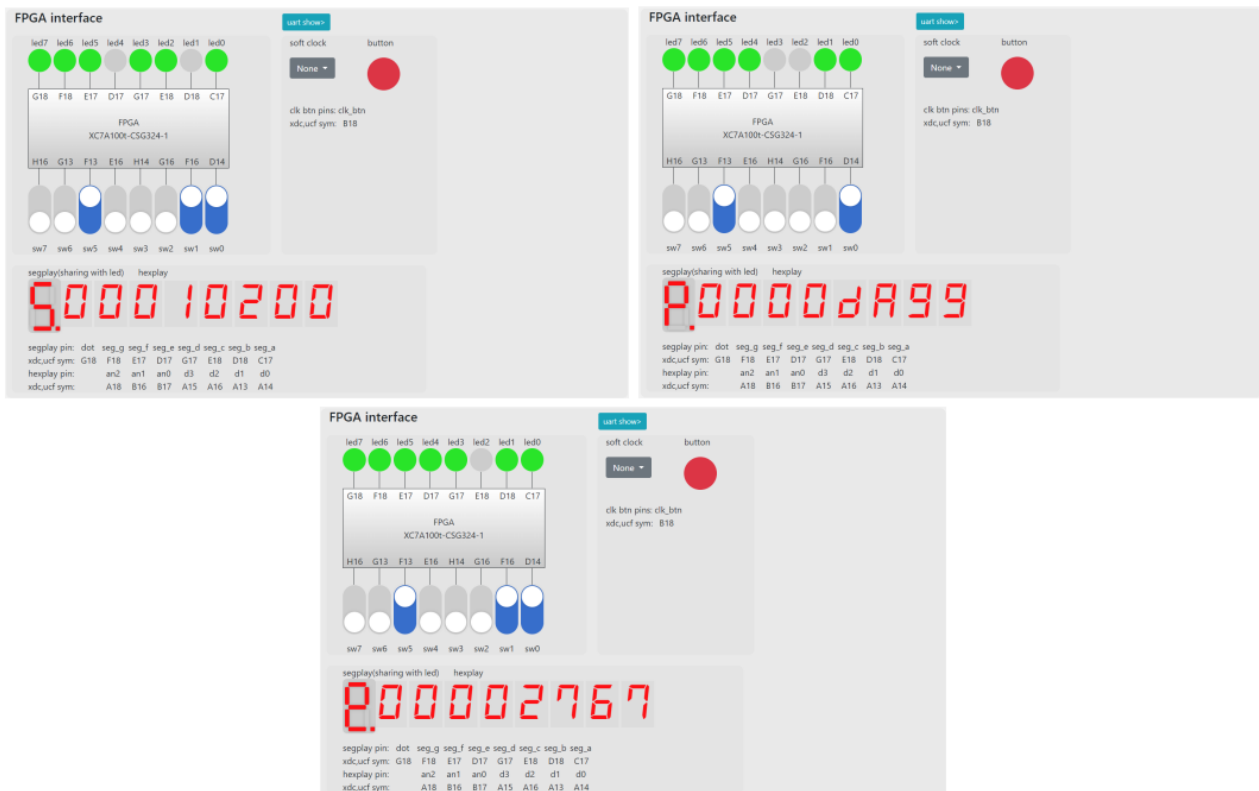
其中数据存储器的 COE 文件由下面的 Python 脚本生成, 生成的数均为正数。

```

1  from random import randrange
2  import sys
3  with open('.\\CPU\\CPU.srcs\\sources_1\\coes\\random_mem.coe', 'w') as
    f:
4      print('memory_initialization_radix=16;', file=f)
5      print('memory_initialization_vector=', file=f)
6
7      for i in range(256):
8          print('%x' % randrange(2147483647), file=f)
9
10     print(';', file=f)

```

烧写至 FPGAOL 平台上, 运行后得到的分支指令执行数, 预测成功数, 预测失败数如下



而有

$$\frac{0x0000DA99}{0x00010200} = \frac{55961}{66048} = 84.7\%$$

即对冒泡排序程序, 分支预测的成功率在 80% 左右.

## 6 附录: 整个项目的 Verilog 代码

### 6.1 Main.v

```

1  module main(
2      input      clk,      // clk_100MHz
3      input      step,     // btn
4      input      rst,      // sw[7]
5      input      run,      // sw[6]
6      input      valid,    // sw[5]
7      input [4:0] in,      // sw[4:0]
8      output     ready,    // led[7]
9      output [1:0] check,  // led[6:5]
10     output [4:0] out_0,   // led[4:0]
11     output [2:0] an,      // segplay_an
12     output [3:0] seg      // segplay_data
13 );
14     wire clk_cpu;
15
16     // IO_BUS
17     wire io_we;
18     wire [7:0] io_addr;
19     wire [31:0] io_din;
20     wire [31:0] io_dout;
21
22     // Debug_BUS
23     wire [31:0] rf_data;
24     wire [31:0] m_data;
25     wire [7:0] dbg_addr;
26
27     // BTB Counter
28     wire [31:0] branch_cnt;
29     wire [31:0] btb_succ_cnt;
30     wire [31:0] btb_fail_cnt;
31
32     // Pipeline Register
33     wire [31:0] pc_in;
34     wire [31:0] pc;
35     wire [31:0] IFID_pc;
36     wire [31:0] IFID_inst;
37     wire [31:0] IDEX_pc;
38     wire [31:0] IDEX_reg_1;
39     wire [31:0] IDEX_reg_2;
40     wire [31:0] IDEX_imm;

```

```

41     wire [4:0] IDEX_rd;
42     wire [31:0] EXMEM_alu_out;
43     wire [31:0] EXMEM_reg_2;
44     wire [4:0] EXMEM_rd;
45     wire [31:0] MEMWB_alu_out;
46     wire [31:0] MEMWB_mem_out;
47     wire [4:0] MEMWB_rd;
48
49     // Wire Mapping
50     cpu CPU(
51         .clk          (clk_cpu),
52         .rst          (rst),
53         .io_we        (io_we),
54         .io_addr      (io_addr),
55         .io_din       (io_din),
56         .io_dout      (io_dout),
57         .rf_data      (rf_data),
58         .m_data       (m_data),
59         .dbg_addr     (dbg_addr),
60         .branch_cnt   (branch_cnt),
61         .btb_succ_cnt (btb_succ_cnt),
62         .btb_fail_cnt (btb_fail_cnt),
63         .pc_in        (pc_in),
64         .pc           (pc),
65         .IFID_pc      (IFID_pc),
66         .IFID_inst    (IFID_inst),
67         .IDEX_pc      (IDEX_pc),
68         .IDEX_reg_1   (IDEX_reg_1),
69         .IDEX_reg_2   (IDEX_reg_2),
70         .IDEX_imm     (IDEX_imm),
71         .IDEX_rd      (IDEX_rd),
72         .EXMEM_alu_out (EXMEM_alu_out),
73         .EXMEM_reg_2   (EXMEM_reg_2),
74         .EXMEM_rd     (EXMEM_rd),
75         .MEMWB_alu_out (MEMWB_alu_out),
76         .MEMWB_mem_out (MEMWB_mem_out),
77         .MEMWB_rd     (MEMWB_rd)
78     );
79
80     pdu PDU(
81         .clk          (clk),
82         .rst          (rst),
83         .run          (run),
84         .step         (step),
85         .clk_cpu      (clk_cpu),
86         .valid        (valid),
87         .in           (in),
88         .ready        (ready),

```



```

89         .check      (check),
90         .out0        (out_0),
91         .an           (an),
92         .seg          (seg),
93         .io_we        (io_we),
94         .io_addr      (io_addr),
95         .io_din       (io_din),
96         .io_dout      (io_dout),
97         .rf_data      (rf_data),
98         .m_data       (m_data),
99         .m_rf_addr    (dbg_addr),
100        .pcin         (pc_in),
101        .pc            (pc),
102        .pcd           (IFID_pc),
103        .pce           (IDEX_pc),
104        .ir            (IFID_inst),
105        .imm           (IDEX_imm),
106        .mdr           (MEMWB_mem_out),
107        .a             (IDEX_reg_1),
108        .b             (IDEX_reg_2),
109        .y             (EXMEM_alu_out),
110        .bm            (EXMEM_reg_2),
111        .yw            (MEMWB_alu_out),
112        .rd            (IDEX_rd),
113        .rdm           (MEMWB_rd),
114        .rdw           (EXMEM_rd),
115        .br_cnt        (branch_cnt),
116        .succ_cnt      (btb_succ_cnt),
117        .fail_cnt      (btb_fail_cnt)
118    );
119 endmodule

```

## 6.2 PDU.v

```

1 module pdu(
2     input clk,
3     input rst,
4
5     //选择CPU工作方式;
6     input run,
7     input step,
8     output clk_cpu,
9
10    //输入switch的端口
11    input valid,
12    input [4:0] in,
13
14    //输出led和seg的端口
15    output [1:0] check, //led6-5:查看类型

```

```

16     output [4:0] out0,    //led4-0
17     output [2:0] an,      //8个数码管
18     output [3:0] seg,
19     output ready,        //led7
20
21     //IO_BUS
22     input      io_we,
23     input [7:0] io_addr,
24     input [31:0] io_dout,
25     output [31:0] io_din,
26
27     //Debug_BUS
28     input [31:0] rf_data,
29     input [31:0] m_data,
30     output reg [7:0] m_rf_addr,
31
32     //增加流水线寄存器调试接口
33     input [31:0] pcin, pc, pcd, pce,
34     input [31:0] ir, imm, mdr,
35     input [31:0] a, b, y, bm, yw,
36     input [4:0] rd, rdm, rdw,
37     input [31:0] br_cnt, succ_cnt, fail_cnt
38
39     // 串口输入输出
40 );
41
42     reg [4:0] in_r, in_2r;    //同步外部输入用，为信号in增加一级寄存器
43     reg run_r, step_r, step_2r, valid_r, valid_2r;
44     wire step_p, valid_pn;    //取边沿信号
45     wire pre_pn,next_pn;      //增加取边沿信号
46
47     reg clk_cpu_r;            //寄存器输出CPU时钟
48     reg [4:0] out0_r;         //输出外设端口
49     reg [31:0] out1_r;
50     reg ready_r;
51     reg [19:0] cnt;           //刷新计数器，刷新频率约为95Hz
52     reg [1:0] check_r;        //查看信息类型，00-运行结果，01-寄存器堆，10-存储器，11-plr
53
54     reg [7:0] io_din_a; //_a表示为满足组合always描述要求定义的，下同
55     reg [4:0] out0_a;
56     reg [31:0] out1_a;
57     reg [3:0] seg_a;
58
59     //增加pre,next取边沿计数器
60     reg [4:0] cnt_m_rf;       //寄存器堆和存储器地址计数器
61     reg [1:0] cnt_ah_plr;     //流水线寄存器高两位地址计数器
62     reg [2:0] cnt_al_plr;     //流水线寄存器低三位地址计数器

```

```

63
64 //增加流水线寄存器地址和数据选择输入
65 wire [4:0] addr_plr ;
66 reg [31:0] plr_data;
67
68 assign clk_cpu = clk_cpu_r;
69 assign io_din = io_din_a;
70 assign check = check_r;
71 assign out0 = out0_a;
72 assign ready = ready_r;
73 assign seg = seg_a;
74 assign an = cnt[19:17];
75 assign step_p = step_r & ~step_2r; //取上升沿
76 assign valid_pn = valid_r ^ valid_2r; //取上升沿或下降沿
77 assign pre_pn = in_r[1] ^ in_2r[1]; //增加pre取上升或下降沿信号
78 assign next_pn = in_r[0] ^ in_2r[0]; //增加next取上升或下降沿信号
79
80 //同步输入信号
81 always @(posedge clk) begin
82     run_r ≤ run;
83     step_r ≤ step;
84     step_2r ≤ step_r;
85     valid_r ≤ valid;
86     valid_2r ≤ valid_r;
87     in_r ≤ in;
88     in_2r ≤ in_r; //为信号in增加一级寄存器
89 end
90
91 //CPU工作方式
92 always @(posedge clk, posedge rst) begin
93     if(rst)
94         clk_cpu_r ≤ 0;
95     else if (run_r)
96         clk_cpu_r ≤ ~clk_cpu_r;
97     else
98         clk_cpu_r ≤ step_p;
99 end
100
101 //读外设端口
102 always @(*) begin
103     case (io_addr)
104         8'h0c:
105             io_din_a = {{27{1'b0}}, in_r};
106         8'h10:
107             io_din_a = {{31{1'b0}}, valid_r};
108         default:
109             io_din_a = 32'h0000_0000;
110     endcase

```

```

111     end
112
113     //写外设端口
114     always @(posedge clk, posedge rst) begin
115         if (rst) begin
116             out0_r ≤ 5'h1f;
117             out1_r ≤ 32'h1234_5678;
118             ready_r ≤ 1'b1;
119         end
120         else if (io_we)
121             case (io_addr)
122                 8'h00:
123                     out0_r ≤ io_dout[4:0];
124                 8'h04:
125                     ready_r ≤ io_dout[0];
126                 8'h08:
127                     out1_r ≤ io_dout;
128                 default:
129                     ;
130             endcase
131     end
132
133     //增加寄存器堆和存储器地址计数：依靠pre,next边沿计数使能
134     always @(posedge clk, posedge rst) begin
135         if (rst)
136             cnt_m_rf ≤ 5'b0_0000;
137         else if (step_p)
138             cnt_m_rf ≤ 5'b0_0000;
139         else if (next_pn)
140             cnt_m_rf ≤ cnt_m_rf + 5'b0_0001;
141         else if (pre_pn)
142             cnt_m_rf ≤ cnt_m_rf - 5'b0_0001;
143     end
144
145     //增加流水寄存器地址计数，流水线寄存器高两位地址依靠pre边沿计数，低三位地址
    依靠next边沿计数
146     always @(posedge clk, posedge rst) begin
147         if (rst)
148             cnt_ah_plr ≤ 2'b00;
149         else if (step_p)
150             cnt_ah_plr ≤ 2'b00;
151         else if (pre_pn)
152             cnt_ah_plr ≤ cnt_ah_plr + 2'b01;
153     end
154
155     always @(posedge clk, posedge rst) begin
156         if (rst)
157             cnt_al_plr ≤ 3'b000;

```

```

158     else if (step_p)
159         cnt_al_plr ≤ 3'b000;
160     else if (next_pn)
161         if (cnt_ah_plr==2'b01)
162             if (cnt_al_plr == 3'b101)
163                 cnt_al_plr ≤ 3'b000;
164             else
165                 cnt_al_plr ≤ cnt_al_plr + 3'b001;
166         else begin
167             cnt_al_plr [2] ≤ 1'b0;
168             cnt_al_plr [1:0] ≤ cnt_al_plr[1:0] + 2'b01;
169         end
170     end
171
172     assign  addr_plr = {cnt_ah_plr,cnt_al_plr}; //增加流水线寄存器地址
173
174     //寄存器堆和存储器地址输出选择
175     //下面的always块也可以用assign m_rf_addr = {in_r[4:2],cnt_m_rf};代替
    因为寄存器堆只需要低5位就可以了，不关心高3位
176     always @(*) begin
177         case (check_r[1])
178             1'b0:
179                 m_rf_addr = {3'b000,cnt_m_rf};
180             1'b1:
181                 m_rf_addr = {in_r[4:2],cnt_m_rf};
182         endcase
183     end
184
185     //流水线寄存器数据选择输入
186     always @(*) begin
187         case (cnt_ah_plr)
188             //PC/IF/ID
189             2'b00:
190                 case (cnt_al_plr[1:0])
191                     2'b00:
192                         plr_data = pc;
193                     2'b01:
194                         plr_data = pcd;
195                     2'b10:
196                         plr_data = ir;
197                     2'b11:
198                         plr_data = pcin;
199                 endcase
200             //ID/EX
201             2'b01: begin
202                 case (cnt_al_plr)
203                     3'b000:
204                         plr_data = pce;

```

```

205         3'b001:
206             plr_data = a;
207         3'b010:
208             plr_data = b;
209         3'b011:
210             plr_data = imm;
211         3'b100:
212             plr_data = {{27{1'b0}},rd};
213         3'b101:
214             plr_data = br_cnt;
215         default:
216             plr_data = pce;
217     endcase
218 end
219 //EX/MEM
220 2'b10:
221 case (cnt_al_plr[1:0])
222     2'b00:
223         plr_data = y;
224     2'b01:
225         plr_data = bm;
226     2'b10:
227         plr_data = {{27{1'b0}},rdm};
228     2'b11:
229         plr_data = succ_cnt;
230 endcase
231 //MEM/WB
232 2'b11:
233 case (cnt_al_plr[1:0])
234     2'b00:
235         plr_data = yw;
236     2'b01:
237         plr_data = mdr;
238     2'b10:
239         plr_data = {{27{1'b0}},rdw};
240     2'b11:
241         plr_data = fail_cnt;
242 endcase
243 endcase
244 end
245
246 //LED和数码管查看类型
247 always @(posedge clk, posedge rst) begin
248     if(rst)
249         check_r ≤ 2'b00;
250     else if(run_r)
251         check_r ≤ 2'b00;
252     else if (step_p)

```

```

253         check_r ≤ 2'b00;
254     else if (valid_pn)
255         check_r ≤ check - 2'b01;
256 end
257
258 //LED和数码管显示内容
259 always @(*) begin
260     case (check_r)
261         2'b00: begin
262             out0_a = out0_r;
263             out1_a = out1_r;
264         end
265         2'b01: begin
266             out0_a = cnt_m_rf;
267             out1_a = rf_data;
268         end
269         2'b10: begin
270             out0_a = cnt_m_rf;
271             out1_a = m_data;
272         end
273         2'b11: begin
274             out0_a = addr_plr;
275             out1_a = plr_data; //更改为流水线寄存器地址和数据显示
276         end
277     endcase
278 end
279
280 //扫描数码管
281 always @(posedge clk, posedge rst) begin
282     if (rst)
283         cnt ≤ 20'h0_0000;
284     else
285         cnt ≤ cnt + 20'h0_0001;
286 end
287
288 always @* begin
289     case (an)
290         3'd0:
291             seg_a = out1_a[3:0];
292         3'd1:
293             seg_a = out1_a[7:4];
294         3'd2:
295             seg_a = out1_a[11:8];
296         3'd3:
297             seg_a = out1_a[15:12];
298         3'd4:
299             seg_a = out1_a[19:16];
300         3'd5:

```

```

301         seg_a = out1_a[23:20];
302     3'd6:
303         seg_a = out1_a[27:24];
304     3'd7:
305         seg_a = out1_a[31:28];
306     default:
307         ;
308     endcase
309 end
310
311 endmodule

```

### 6.3 CPU.v

```

1  module cpu(
2      input clk,
3      input rst,
4
5      // IO_BUS
6      input    [31:0] io_din,    // 来自 sw 的输入数据
7      output   [7:0]  io_addr,   // led 和 seg 的地址
8      output   [31:0] io_dout,   // 输出 led 和 seg 的数据
9      output                   io_we, // 输出 led 和 seg 数据时的使能信号
10
11     // Debug_BUS
12     input    [7:0] dbg_addr, // 存储器 (MEM) 或寄存器堆 (RF) 的调试
读口地址
13     output   [31:0] rf_data, // 从RF读取的数据
14     output   [31:0] m_data, // 从MEM读取的数据
15
16     // BTB Counter
17     output   [31:0] branch_cnt,
18     output   [31:0] btb_succ_cnt,
19     output   [31:0] btb_fail_cnt,
20
21     // PC/IF/ID 流水段寄存器
22     output reg [31:0] pc,
23     output reg [31:0] pc_in,
24     output   [31:0] IFID_pc,
25     output   [31:0] IFID_inst,
26
27     // ID/EX 流水段寄存器
28     output   [31:0] IDEX_pc,
29     output   [31:0] IDEX_reg_1,
30     output   [31:0] IDEX_reg_2,
31     output   [31:0] IDEX_imm,
32     output   [4:0]  IDEX_rd,
33
34     // EX/MEM 流水段寄存器

```



```

35     output    [31:0] EXMEM_alu_out,
36     output    [31:0] EXMEM_reg_2,
37     output    [4:0]  EXMEM_rd,
38
39     // MEM/WB 流水段寄存器
40     output    [31:0] MEMWB_alu_out,
41     output    [31:0] MEMWB_mem_out,
42     output    [4:0]  MEMWB_rd
43 );
44     // Control Signal
45     wire      enable;
46
47     // IF: pc, inst_mem, btb
48     //     reg [31:0] pc;
49     //     reg [31:0] pc_in;
50     wire [31:0] pc_next;
51     wire [31:0] pc_add;
52     wire [31:0] inst;
53     wire [1:0]  pc_sel;
54     wire      btb_hit;
55     wire [31:0] btb_target;
56     wire      btb_fail;
57     //     wire [31:0] branch_cnt;
58     //     wire [31:0] btb_succ_cnt;
59     //     wire [31:0] btb_fail_cnt;
60
61
62     // IF/ID
63     wire      IFID_flush;
64     //     wire [31:0] IFID_pc;
65     wire [31:0] IFID_pc_next;
66     //     wire [31:0] IFID_inst;
67     wire      IFID_btb_hit;
68
69     // ID
70     wire      jump;
71     wire      branch;
72     wire [1:0] alu_op;
73     wire [1:0] alu_1_src;
74     wire      alu_2_src;
75     wire      pc_add_src;
76     wire      mem_read;
77     wire      mem_write;
78     wire [1:0] reg_src;
79     wire      reg_write;
80     wire [3:0] alu_funct;
81     wire [2:0] funct3;
82     wire [31:0] reg_1;

```

```

83     wire [31:0] reg_2;
84     wire [4:0]  rs_1;
85     wire [4:0]  rs_2;
86     wire [4:0]  rd;
87     wire [31:0] imm;
88
89     // ID/EX
90     wire        IDEX_flush;
91     wire        IDEX_jump;
92     wire        IDEX_branch;
93     wire [3:0]  IDEX_alu_funct;
94     wire [1:0]  IDEX_alu_1_src;
95     wire        IDEX_alu_2_src;
96     wire        IDEX_pc_add_src;
97     wire        IDEX_mem_read;
98     wire        IDEX_mem_write;
99     wire [1:0]  IDEX_reg_src;
100    wire        IDEX_reg_write;
101    wire [2:0]  IDEX_funct3;
102    //     wire [31:0] IDEX_pc;
103    wire [31:0] IDEX_pc_next;
104    wire        IDEX_btb_hit;
105    //     wire [31:0] IDEX_reg_1;
106    //     wire [31:0] IDEX_reg_2;
107    //     wire [31:0] IDEX_imm;
108    //     wire [4:0]  IDEX_rd;
109    wire [4:0]  IDEX_rs_1;
110    wire [4:0]  IDEX_rs_2;
111
112    // EX
113    wire [1:0]  forward_1;
114    wire [1:0]  forward_2;
115    reg  [31:0] EX_reg_1;
116    reg  [31:0] EX_reg_2;
117    reg  [31:0] alu_in_1;
118    wire [31:0] alu_in_2;
119    wire [31:0] alu_out;
120    wire        cmp;
121    wire        branch_taken;
122
123    // EX/MEM
124    wire [31:0] EXMEM_pc_next;
125    wire        EXMEM_mem_read;
126    wire        EXMEM_mem_write;
127    wire [1:0]  EXMEM_reg_src;
128    wire        EXMEM_reg_write;
129    wire [2:0]  EXMEM_funct3;
130    //     wire [31:0] EXMEM_alu_out;

```

```

131 //    wire [31:0] EXMEM_reg_2;
132 //    wire [4:0]  EXMEM_rd;
133
134 // MEM
135 wire      mem_we;
136 wire [31:0] mem_out;
137 wire [31:0] mem_data;
138 // MEM/WB
139 wire [1:0]  MEMWB_reg_src;
140 wire      MEMWB_reg_write;
141 wire [31:0] MEMWB_pc_next;
142 //    wire [31:0] MEMWB_mem_out;
143 //    wire [31:0] MEMWB_alu_out;
144 //    wire [4:0]  MEMWB_rd;
145 // WB
146 reg  [31:0] WB_data;
147
148 // IF: pc, inst_mem
149 assign pc_next = pc + 32'h4;
150
151 always @(*) begin
152     if (IDEX_jump) pc_in = pc_add;
153     else begin
154         case (pc_sel)
155             2'b11: pc_in = IDEX_pc_next;
156             2'b10: pc_in = pc_add;
157             2'b01: pc_in = btb_target;
158             2'b00: pc_in = pc_next;
159         endcase
160     end
161 end
162
163 always @(posedge clk or posedge rst) begin
164     if (rst)      pc ≤ 32'h3000;
165     else if (enable) pc ≤ pc_in;
166 end
167
168 btb #(BUF_LEN(4)) BTB (
169     .clk      (clk),
170     .rst      (rst),
171     .enable   (enable),
172     .pc       (pc[31:2]),
173     .IDEX_pc  (IDEX_pc[31:2]),
174     .IDEX_branch (IDEX_branch),
175     .branch_taken (branch_taken),
176     .IDEX_branch_target (pc_add),
177     .IDEX_btb_hit (IDEX_btb_hit),
178     .pc_sel    (pc_sel),

```

```

179     .btb_hit          (btb_hit),
180     .btb_target       (btb_target),
181     .btb_fail         (btb_fail),
182     .branch_cnt       (branch_cnt),
183     .btb_succ_cnt     (btb_succ_cnt),
184     .btb_fail_cnt     (btb_fail_cnt)
185 );
186
187 rom Inst_Mem (
188     .a    (pc[9:2]),
189     .spo  (inst)
190 );
191 // IF/ID Register
192 if_id IF_ID (
193     .clk      (clk),
194     .rst      (rst),
195     .en       (enable),
196     .clr      (IFID_flush),
197     .pc       (pc),
198     .pc_next  (pc_next),
199     .btb_hit  (btb_hit),
200     .inst     (inst),
201     .IFID_pc  (IFID_pc),
202     .IFID_pc_next (IFID_pc_next),
203     .IFID_btb_hit (IFID_btb_hit),
204     .IFID_inst (IFID_inst)
205 );
206 // ID: hazard, ctrl, regfile, imm_gen
207 assign funct3 = IFID_inst[14:12];
208 assign rs_1   = IFID_inst[19:15];
209 assign rs_2   = IFID_inst[24:20];
210 assign rd     = IFID_inst[11:7];
211
212 hazard Hazard (
213     .rs_1      (rs_1),
214     .rs_2      (rs_2),
215     .btb_fail  (btb_fail),
216     .IDEX_jump (IDEX_jump),
217     .IDEX_rd   (IDEX_rd),
218     .IDEX_mem_read (IDEX_mem_read),
219     .enable    (enable),
220     .IFID_flush (IFID_flush),
221     .IDEX_flush (IDEX_flush)
222 );
223
224 control Control (
225     .inst      (IFID_inst[6:0]),
226     .jump      (jump),

```

```

227     .branch      (branch),
228     .alu_op      (alu_op),
229     .alu_1_src   (alu_1_src),
230     .alu_2_src   (alu_2_src),
231     .pc_add_src  (pc_add_src),
232     .mem_read    (mem_read),
233     .mem_write   (mem_write),
234     .reg_src     (reg_src),
235     .reg_write   (reg_write)
236 );
237
238 alu_control ALU_control (
239     .alu_op      (alu_op),
240     .funct3      (funct3),
241     .funct7      (IFID_inst[30]),
242     .alu_funct   (alu_funct)
243 );
244
245 regfile Reg_File (
246     .clk (clk),
247     .we  (MEMWB_reg_write),
248     .ra1 (rs_1),
249     .ra2 (rs_2),
250     .ra3 (dbg_addr[4:0]),
251     .wa  (MEMWB_rd),
252     .wd  (WB_data),
253     .rd1 (reg_1),
254     .rd2 (reg_2),
255     .rd3 (rf_data)
256 );
257
258 imm_gen Imm_Gen (
259     .inst (IFID_inst),
260     .imm  (imm)
261 );
262 // ID/EX Register
263 id_ex ID_EX (
264     .clk          (clk),
265     .rst          (rst),
266     .clr          (IDEX_flush),
267     .jump         (jump),
268     .branch       (branch),
269     .alu_funct     (alu_funct),
270     .alu_1_src     (alu_1_src),
271     .alu_2_src     (alu_2_src),
272     .pc_add_src    (pc_add_src),
273     .mem_read      (mem_read),
274     .mem_write     (mem_write),

```

```

275     .reg_src      (reg_src),
276     .reg_write    (reg_write),
277     .IFID_pc      (IFID_pc),
278     .IFID_pc_next  (IFID_pc_next),
279     .IFID_btb_hit  (IFID_btb_hit),
280     .funct3       (funct3),
281     .reg_1        (reg_1),
282     .reg_2        (reg_2),
283     .imm          (imm),
284     .rs_1         (rs_1),
285     .rs_2         (rs_2),
286     .rd           (rd),
287     .IDEX_jump    (IDEX_jump),
288     .IDEX_branch   (IDEX_branch),
289     .IDEX_alu_funct (IDEX_alu_funct),
290     .IDEX_alu_1_src (IDEX_alu_1_src),
291     .IDEX_alu_2_src (IDEX_alu_2_src),
292     .IDEX_pc_add_src (IDEX_pc_add_src),
293     .IDEX_mem_read  (IDEX_mem_read),
294     .IDEX_mem_write (IDEX_mem_write),
295     .IDEX_reg_src   (IDEX_reg_src),
296     .IDEX_reg_write (IDEX_reg_write),
297     .IDEX_pc       (IDEX_pc),
298     .IDEX_pc_next   (IDEX_pc_next),
299     .IDEX_btb_hit   (IDEX_btb_hit),
300     .IDEX_funct3    (IDEX_funct3),
301     .IDEX_reg_1     (IDEX_reg_1),
302     .IDEX_reg_2     (IDEX_reg_2),
303     .IDEX_imm       (IDEX_imm),
304     .IDEX_rs_1      (IDEX_rs_1),
305     .IDEX_rs_2      (IDEX_rs_2),
306     .IDEX_rd        (IDEX_rd)
307 );
308 // EX: forward, alu, pc_adder
309 assign pc_add      = (IDEX_pc_add_src ? EX_reg_1 : IDEX_pc) +
IDEX_imm;
310 assign branch_taken = IDEX_branch & cmp;
311 always @(*) begin
312     case (forward_1)
313         2'b00: EX_reg_1 = IDEX_reg_1;
314         2'b01: EX_reg_1 = WB_data;
315         2'b10: EX_reg_1 = EXMEM_alu_out;
316         default: EX_reg_1 = 32'h0;
317     endcase
318 end
319
320 always @(*) begin
321     case (forward_2)

```

```

322         2'b00: EX_reg_2 = IDEX_reg_2;
323         2'b01: EX_reg_2 = WB_data;
324         2'b10: EX_reg_2 = EXMEM_alu_out;
325         default: EX_reg_2 = 32'h0;
326     endcase
327 end
328
329 always @(*) begin
330     case (IDEX_alu_1_src)
331         2'b00: alu_in_1 = EX_reg_1;
332         2'b01: alu_in_1 = 32'h0;
333         2'b10: alu_in_1 = IDEX_pc;
334         default: alu_in_1 = 32'h0;
335     endcase
336 end
337 assign alu_in_2 = IDEX_alu_2_src ? IDEX_imm : EX_reg_2;
338
339 forward Forward (
340     .IDEX_rs_1      (IDEX_rs_1),
341     .IDEX_rs_2      (IDEX_rs_2),
342     .EXMEM_reg_write (EXMEM_reg_write),
343     .EXMEM_rd        (EXMEM_rd),
344     .MEMWB_reg_write (MEMWB_reg_write),
345     .MEMWB_rd        (MEMWB_rd),
346     .forward_1       (forward_1),
347     .forward_2       (forward_2)
348 );
349
350 alu ALU (
351     .in_1      (alu_in_1),
352     .in_2      (alu_in_2),
353     .funct      (IDEX_alu_funct),
354     .arithmetic (alu_out),
355     .comparision (cmp)
356 );
357
358 // EX/MEM Register
359 ex_mem EX_MEM (
360     .clk      (clk),
361     .rst      (rst),
362     .IDEX_mem_read (IDEX_mem_read),
363     .IDEX_mem_write (IDEX_mem_write),
364     .IDEX_reg_src   (IDEX_reg_src),
365     .IDEX_reg_write (IDEX_reg_write),
366     .IDEX_funct3    (IDEX_funct3),
367     .alu_out        (alu_out),
368     .IDEX_pc_next   (IDEX_pc_next),
369     .EX_reg_2       (EX_reg_2),

```

```

370     .IDEX_rd      (IDEX_rd),
371     .EXMEM_mem_read (EXMEM_mem_read),
372     .EXMEM_mem_write (EXMEM_mem_write),
373     .EXMEM_reg_src  (EXMEM_reg_src),
374     .EXMEM_reg_write (EXMEM_reg_write),
375     .EXMEM_funct3   (EXMEM_funct3),
376     .EXMEM_alu_out  (EXMEM_alu_out),
377     .EXMEM_pc_next  (EXMEM_pc_next),
378     .EXMEM_reg_2    (EXMEM_reg_2),
379     .EXMEM_rd       (EXMEM_rd)
380 );
381
382 // MEM: data_mem
383 assign io_we    = EXMEM_mem_write & EXMEM_alu_out[10];
384 assign io_addr  = EXMEM_alu_out[7:0];
385 assign io_dout  = EXMEM_reg_2;
386 assign mem_we   = EXMEM_mem_write & (~EXMEM_alu_out[10]);
387 assign mem_out  = EXMEM_alu_out[10] ? io_din : mem_data;
388
389 data_mem Data_Mem (
390     .clk      (clk),
391     .funct3   (EXMEM_funct3),
392     .addr     (EXMEM_alu_out[9:0]),
393     .data     (EXMEM_reg_2),
394     .mem_read (EXMEM_mem_read),
395     .mem_we   (mem_we),
396     .dbg_addr (dbg_addr),
397     .mem_data (mem_data),
398     .dbg_data (m_data)
399 );
400
401 // MEM/WB Register
402 mem_wb MEM_WB (
403     .clk      (clk),
404     .rst      (rst),
405     .EXMEM_reg_src  (EXMEM_reg_src),
406     .EXMEM_reg_write (EXMEM_reg_write),
407     .mem_out        (mem_out),
408     .EXMEM_pc_next  (EXMEM_pc_next),
409     .EXMEM_alu_out  (EXMEM_alu_out),
410     .EXMEM_rd       (EXMEM_rd),
411     .MEMWB_reg_src  (MEMWB_reg_src),
412     .MEMWB_reg_write (MEMWB_reg_write),
413     .MEMWB_mem_out  (MEMWB_mem_out),
414     .MEMWB_pc_next  (MEMWB_pc_next),
415     .MEMWB_alu_out  (MEMWB_alu_out),
416     .MEMWB_rd       (MEMWB_rd)
417 );

```



```

418 // WB
419 always @(*) begin
420     case (MEMWB_reg_src)
421         2'b00: WB_data = MEMWB_alu_out;
422         2'b01: WB_data = MEMWB_mem_out;
423         2'b10: WB_data = MEMWB_pc_next;
424         default: WB_data = 32'h0;
425     endcase
426 end
427 endmodule

```

## 6.4 Data\_Mem.v

```

1 module data_mem(
2     input          clk,
3     input  [2:0]   funct3,
4     input  [9:0]   addr,
5     input  [31:0]  data,
6     input          mem_read,
7     input          mem_we,
8     input  [7:0]   dbg_addr,
9     output reg [31:0] mem_data,
10    output  [31:0]  dbg_data
11 );
12    reg  [31:0] data_in;
13    wire [31:0] data_out;
14
15    always @(*) begin
16        if (mem_we) begin
17            case(funct3)
18                3'b000:
19                    case(addr[1:0])
20                        2'b00: data_in = {data_out[31:8], data[7:0]};
21                        2'b01: data_in = {data_out[31:16], data[7:0],
data_out[7:0]};
22                        2'b10: data_in = {data_out[31:24], data[7:0],
data_out[15:0]};
23                        2'b11: data_in = {data[7:0], data_out[23:0]};
24                    endcase
25                3'b001:
26                    case (addr[1])
27                        1'b0: data_in = {data_out[31:16], data[15:0]};
28                        1'b1: data_in = {data[15:0], data_out[15:0]};
29                    endcase
30                3'b010: data_in = data;
31                default: data_in = data_out;
32            endcase
33        end
34    else begin

```

```

35         data_in = data_out;
36     end
37 end
38
39 always @(*) begin
40     if (mem_read) begin
41         case (funct3)
42             3'b000:
43                 case (addr[1:0])
44                     2'b00: mem_data = {{24{data_out[7]}}},
45                     data_out[7:0]];
46                     2'b01: mem_data = {{24{data_out[15]}}},
47                     data_out[15:8]];
48                     2'b10: mem_data = {{24{data_out[23]}}},
49                     data_out[23:16]];
50                     2'b11: mem_data = {{24{data_out[31]}}},
51                     data_out[31:24]];
52                 endcase
53             3'b001:
54                 case (addr[1])
55                     1'b0: mem_data = {{16{data_out[15]}}},
56                     data_out[15:0]];
57                     1'b1: mem_data = {{16{data_out[31]}}},
58                     data_out[31:16]];
59                 endcase
60             3'b010: mem_data = data_out;
61             3'b100:
62                 case (addr[1:0])
63                     2'b00: mem_data = {24'h0, data_out[7:0]};
64                     2'b01: mem_data = {24'h0, data_out[15:8]};
65                     2'b10: mem_data = {24'h0, data_out[23:16]};
66                     2'b11: mem_data = {24'h0, data_out[31:24]};
67                 endcase
68             3'b101:
69                 case (addr[1])
70                     1'b0: mem_data = {16'h0, data_out[15:0]};
71                     1'b1: mem_data = {16'h0, data_out[31:16]};
72                 endcase
73             default: mem_data = data_out;
74         endcase
75     end
76 else begin
77     mem_data = data_out;
78 end
79 end
80
81 dual_port_ram Data_Mem(
82     .clk    (clk),

```

```

77     .we    (mem_we),
78     .a     (addr[9:2]),
79     .dpra  (dbg_addr),
80     .d     (data_in),
81     .spo   (data_out),
82     .dpo   (dbg_data)
83 );
84 endmodule

```

## 6.5 Ctrl.v

关于 Ctrl.v 的介绍见 [control 模块](#)

```

1  module control(
2      input  [6:0] inst,
3      output      jump,
4      output      branch,
5      output [1:0] alu_op,      // 2'b11 for branch, 2'b10 for imm, 2'b01
for reg
6      output [1:0] alu_1_src,  // 2'b10 for auipc, 2'b01 for lui
7      output      alu_2_src,
8      output      pc_add_src, // 1'b1 for jalr
9      output      mem_read,
10     output      mem_write,
11     output [1:0] reg_src,
12     output      reg_write
13 );
14     wire lui;
15     wire auipc;
16     wire load;
17     wire store;
18     wire reg_arith;
19     wire imm_arith;
20     assign lui      = (inst == 7'b0110111);
21     assign auipc    = (inst == 7'b0010111);
22     assign load     = (inst == 7'b0000011);
23     assign store    = (inst == 7'b0100011);
24     assign reg_arith = (inst == 7'b0110011);
25     assign imm_arith = (inst == 7'b0010011);
26
27     assign jump      = (inst == 7'b1101111) | (inst == 7'b1100111);
28     assign branch    = (inst == 7'b1100011);
29     assign alu_op    = {(imm_arith | branch), (reg_arith | branch)};
30     assign alu_1_src = {auipc, lui};
31     assign alu_2_src = (imm_arith | load | store | lui | auipc);
32     assign pc_add_src = (inst == 7'b1100111);
33     assign mem_read  = (load);
34     assign mem_write = (store);
35     assign reg_src   = {jump, load};

```

```

36     assign reg_write = (reg_arith | imm_arith | load | auipc | lui |
    jump);
37 endmodule

```

## 6.6 ALU\_Ctrl.v

关于 ALU\_control.v 的介绍见 [alu\\_control 模块](#)

```

1 module alu_control(
2     input    [1:0] alu_op,
3     input    [2:0] funct3,
4     input    funct7,
5     output reg [3:0] alu_funct
6 );
7     always @(*) begin
8         case (alu_op)
9             2'b00: alu_funct = 4'b1111;
10            2'b01: alu_funct = {funct7, funct3};
11            2'b10: alu_funct = {((funct3 == 3'b101) & funct7),
    funct3};
12            2'b11: alu_funct = {1'b0, funct3};
13            default: alu_funct = 4'b1111;
14        endcase
15    end
16 endmodule

```

## 6.7 Reg\_File.v

```

1 module regfile (
2     input    clk,
3     input    we,
4     input    [4:0] ra1,
5     input    [4:0] ra2,
6     input    [4:0] ra3,
7     input    [4:0] wa,
8     input    [31:0] wd,
9     output reg [31:0] rd1,
10    output reg [31:0] rd2,
11    output reg [31:0] rd3
12 );
13    reg [31:0] rf [31:0];
14
15    always @(posedge clk) begin
16        if (we) rf[wa] ≤ wd;
17    end
18
19    always @(*) begin
20        if (ra1 == 5'h0) rd1 = 32'h0;
21        else if (ra1 == wa) rd1 = wd;

```

```

22         else                rd1 = rf[ra1];
23     end
24
25     always @(*) begin
26         if (ra2 == 5'h0)      rd2 = 32'h0;
27         else if (ra2 == wa)   rd2 = wd;
28         else                  rd2 = rf[ra2];
29     end
30
31     always @(*) begin
32         if (ra3 == 5'h0)      rd3 = 32'h0;
33         else if (ra3 == wa)   rd3 = wd;
34         else                  rd3 = rf[ra3];
35     end
36 endmodule

```

## 6.8 Imm\_Gen.v

```

1  module imm_gen(
2      input    [31:0] inst,
3      output reg [31:0] imm
4  );
5      always @(*) begin
6          case (inst[6:0])
7              7'b0110111, // lui
8              7'b0010111: // auipc
9                  imm = {inst[31:12], 12'h0000};
10
11              7'b1101111: // jal
12                  imm = {{12{inst[31]}}, inst[19:12], inst[20],
inst[30:21], 1'b0};
13
14              7'b1100011: // branch
15                  imm = {{20{inst[31]}}, inst[7], inst[30:25],
inst[11:8], 1'b0};
16
17              7'b1100111, // jalr
18              7'b0000011: // load
19                  imm = {{21{inst[31]}}, inst[30:20]};
20
21              7'b0010011: // imm arithmetic
22                  imm = (inst[13:12] == 2'b01) ?
23                      {27'h0, inst[24:20]} :
24                      {{21{inst[31]}}, inst[30:20]};
25
26              7'b0100011: // store
27                  imm = {{21{inst[31]}}, inst[30:25], inst[11:7]};
28
29              default:

```

```

30         imm = 32'h0;
31     endcase
32 end
33 endmodule

```

## 6.9 ALU.v

```

1  module alu(
2      input    [31:0] in_1,
3      input    [31:0] in_2,
4      input    [3:0]  funct,
5      output reg [31:0] arithmetic,
6      output reg      comparision
7  );
8      always @(*) begin
9          case (funct)
10             4'b0000: arithmetic = in_1 + in_2;
11             4'b0001: arithmetic = in_1 << in_2;
12             4'b0010: arithmetic = $signed(in_1) < $signed(in_2);
13             4'b0011: arithmetic = in_1 < in_2;
14             4'b0100: arithmetic = in_1 ^ in_2;
15             4'b0101: arithmetic = in_1 >> in_2;
16             4'b0110: arithmetic = in_1 | in_2;
17             4'b0111: arithmetic = in_1 & in_2;
18             4'b1000: arithmetic = in_1 - in_2;
19             4'b1101: arithmetic = $signed(in_1) >>> in_2;
20             4'b1111: arithmetic = in_1 + in_2;
21             default: arithmetic = 32'h0;
22         endcase
23     end
24
25     always @(*) begin
26         case (funct)
27             4'b0000: comparision = in_1 == in_2;
28             4'b0001: comparision = in_1 != in_2;
29             4'b0100: comparision = $signed(in_1) < $signed(in_2);
30             4'b0101: comparision = $signed(in_1) ≥ $signed(in_2);
31             4'b0110: comparision = in_1 < in_2;
32             4'b0111: comparision = in_1 ≥ in_2;
33             default: comparision = 1'b0;
34         endcase
35     end
36 endmodule

```

## 6.10 Forward.v

```
1 module forward(
2     input      [4:0]  IDEX_rs_1,
3     input      [4:0]  IDEX_rs_2,
4     input      EXMEM_reg_write,
5     input      [4:0]  EXMEM_rd,
6     input      MEMWB_reg_write,
7     input      [4:0]  MEMWB_rd,
8     output reg [1:0]  forward_1,
9     output reg [1:0]  forward_2
10 );
11     always @(*) begin
12         if (EXMEM_reg_write & (EXMEM_rd ≠ 5'h0) & (EXMEM_rd ==
13 IDEX_rs_1))
14             forward_1 = 2'b10;
15         else if (MEMWB_reg_write & (MEMWB_rd ≠ 5'h0) & (MEMWB_rd ==
16 IDEX_rs_1))
17             forward_1 = 2'b01;
18         else
19             forward_1 = 2'b00;
20     end
21     always @(*) begin
22         if (EXMEM_reg_write & (EXMEM_rd ≠ 5'h0) & (EXMEM_rd ==
23 IDEX_rs_2))
24             forward_2 = 2'b10;
25         else if (MEMWB_reg_write & (MEMWB_rd ≠ 5'h0) & (MEMWB_rd ==
26 IDEX_rs_2))
27             forward_2 = 2'b01;
28         else
29             forward_2 = 2'b00;
30     end
31 endmodule
```

## 6.11 Hazard.v

```
1 module hazard(
2     input [4:0] rs_1,
3     input [4:0] rs_2,
4     input      btb_fail,
5     input      IDEX_jump,
6     input [4:0] IDEX_rd,
7     input      IDEX_mem_read,
8     output reg enable,
9     output      IFID_flush,
10    output      IDEX_flush
11 );
12    always @(*) begin
13        if ((IDEX_mem_read) & (IDEX_rd ≠ 5'h0) &
```

```

14         (IDEX_rd == rs_1 | IDEX_rd == rs_2)
15     )
16     enable = 1'b0;
17     else
18         enable = 1'b1;
19 end
20 assign IFID_flush = btb_fail | IDEX_jump;
21 assign IDEX_flush = ~enable | btb_fail | IDEX_jump;
22 endmodule

```

## 6.12 BTB.v

关于 BTB.v 的介绍见[分支预测](#)

```

1 module btb #(
2     parameter BUF_LEN = 4
3 ) (
4     input          clk,
5     input          rst,
6     input          enable,
7     input [29:0]   pc,
8     input [29:0]   IDEX_pc,
9     input          IDEX_branch,
10    input          branch_taken,
11    input [31:0]    IDEX_branch_target,
12    input          IDEX_btb_hit,
13    output reg [1:0] pc_sel,
14    output reg      btb_hit,
15    output reg [31:0] btb_target,
16    output          btb_fail,
17
18    // Counters
19    output reg [31:0] branch_cnt,
20    output reg [31:0] btb_succ_cnt,
21    output reg [31:0] btb_fail_cnt
22 );
23     localparam TAG_LEN  = 30 - BUF_LEN;
24     localparam BUF_SIZE = 1 << BUF_LEN;
25
26     localparam STRONG_HIT  = 2'b11;
27     localparam WEAK_HIT   = 2'b10;
28     localparam WEAK_MISS  = 2'b01;
29     localparam STRONG_MISS = 2'b00;
30
31     // Buffer Register
32     reg [TAG_LEN - 1 : 0] tag    [BUF_SIZE - 1 : 0];
33     reg [31:0]            target [BUF_SIZE - 1 : 0];
34     reg                  valid  [BUF_SIZE - 1 : 0];
35     reg [1:0]             state  [BUF_SIZE - 1 : 0];

```



```

36
37 // PC tag and index
38 wire [TAG_LEN - 1 : 0] pc_tag;
39 wire [BUF_LEN - 1 : 0] pc_idx;
40 wire [TAG_LEN - 1 : 0] IDEX_pc_tag;
41 wire [BUF_LEN - 1 : 0] IDEX_pc_idx;
42
43 assign {pc_tag,      pc_idx}      = pc;
44 assign {IDEX_pc_tag, IDEX_pc_idx} = IDEX_pc;
45
46 // Generate Target
47 always @(*) begin
48     if (rst) begin
49         btb_hit      = 1'b0;
50         btb_target = 32'h0;
51     end
52     else if (valid[pc_idx] &&
53             (pc_tag == tag[pc_idx]) &&
54             state[pc_idx][1]) begin
55         btb_hit      = 1'b1;
56         btb_target = target[pc_idx];
57     end
58     else begin
59         btb_hit      = 1'b0;
60         btb_target = 32'h0;
61     end
62 end
63
64 // Buffer Update
65 reg btb_p_nt; // branch predicted but not taken
66 reg btb_np_t; // not branch predicted but taken
67
68 assign btb_fail = btb_p_nt | btb_np_t;
69
70 always @(*) begin
71     if (rst) begin
72         btb_p_nt = 1'b0;
73         btb_np_t = 1'b0;
74     end
75     else begin
76         btb_p_nt = (IDEX_btb_hit) & (~branch_taken);
77         btb_np_t = (~IDEX_btb_hit) & (branch_taken);
78     end
79 end
80
81 integer i = 0;
82 always @(posedge clk or posedge rst) begin
83     if (rst) begin

```

```

84         for (i = 0; i < BUF_SIZE; i = i + 1) begin
85             tag[i]      ≤ 0;
86             target[i]   ≤ 32'h0;
87             valid[i]    ≤ 1'b0;
88             state[i]    ≤ WEAK_MISS;
89         end
90     end
91     else if (enable && IDEX_branch) begin
92         // tag matched, update state
93         if ((tag[IDEX_pc_idx] == IDEX_pc_tag) &&
valid[IDEX_pc_idx]) begin
94             case (state[IDEX_pc_idx])
95                 STRONG_HIT:
96                     state[IDEX_pc_idx] ≤ btb_fail ? WEAK_HIT :
STRONG_HIT;
97                 WEAK_HIT:
98                     state[IDEX_pc_idx] ≤ btb_fail ? WEAK_MISS :
STRONG_HIT;
99                 WEAK_MISS:
100                     state[IDEX_pc_idx] ≤ btb_fail ? WEAK_HIT :
STRONG_MISS;
101                 STRONG_MISS:
102                     state[IDEX_pc_idx] ≤ btb_fail ? WEAK_MISS :
STRONG_MISS;
103             endcase
104         end
105         // tag not matched, change buffer
106         else begin
107             tag[IDEX_pc_idx]      ≤ IDEX_pc_tag;
108             target[IDEX_pc_idx]   ≤ IDEX_branch_target;
109             valid[IDEX_pc_idx]    ≤ 1'b1;
110             state[IDEX_pc_idx]    ≤ WEAK_MISS;
111         end
112     end
113 end
114 // PC Controller
115 always @(*) begin
116     if (btb_p_nt)      pc_sel ≤ 2'b11; // IDEX_pc_next
117     else if (btb_np_t) pc_sel ≤ 2'b10; // pc_add
118     else if (btb_hit)  pc_sel ≤ 2'b01; // btb_target
119     else               pc_sel ≤ 2'b00; // pc_next
120 end
121
122 // Counter Part
123 always @(posedge clk or posedge rst) begin
124     if (rst) begin
125         branch_cnt    ≤ 32'h0;
126         btb_succ_cnt  ≤ 32'h0;

```

```

127         btb_fail_cnt ≤ 32'h0;
128     end
129     else begin
130         if (IDEX_branch) begin
131             branch_cnt ≤ branch_cnt + 32'h1;
132             if (btb_fail) btb_fail_cnt ≤ btb_fail_cnt + 32'h1;
133             else btb_succ_cnt ≤ btb_succ_cnt + 32'h1;
134         end
135     end
136 end
137 end
138 endmodule //btb

```

## 6.13 流水段寄存器

### 6.13.1 IF\_ID.v

```

1 module if_id(
2     input          clk,
3     input          rst,
4     input          en,
5     input          clr,
6     input [31:0] pc,
7     input [31:0] pc_next,
8     input          btb_hit,
9     input [31:0] inst,
10    output reg [31:0] IFID_pc,
11    output reg [31:0] IFID_pc_next,
12    output reg      IFID_btb_hit,
13    output reg [31:0] IFID_inst
14 );
15    always @(posedge clk or posedge rst) begin
16        if (rst | clr) begin
17            IFID_pc      ≤ 32'h0;
18            IFID_pc_next ≤ 32'h0;
19            IFID_btb_hit ≤ 1'h0;
20            IFID_inst    ≤ 32'h0;
21        end
22        else if (en) begin
23            IFID_pc      ≤ pc;
24            IFID_pc_next ≤ pc_next;
25            IFID_btb_hit ≤ btb_hit;
26            IFID_inst    ≤ inst;
27        end
28    end
29 endmodule

```

### 6.13.2 ID\_EX.v

```
1  module id_ex(
2      input          clk,
3      input          rst,
4      input          clr,
5      input          jump,
6      input          branch,
7      input          [3:0] alu_funct,
8      input          [1:0] alu_1_src,
9      input          alu_2_src,
10     input          pc_add_src,
11     input          mem_read,
12     input          mem_write,
13     input          [1:0] reg_src,
14     input          reg_write,
15     input          [31:0] IFID_pc,
16     input          [31:0] IFID_pc_next,
17     input          IFID_btb_hit,
18     input          [2:0] funct3,
19     input          [31:0] reg_1,
20     input          [31:0] reg_2,
21     input          [31:0] imm,
22     input          [4:0] rs_1,
23     input          [4:0] rs_2,
24     input          [4:0] rd,
25     output reg      IDEX_jump,
26     output reg      IDEX_branch,
27     output reg [3:0] IDEX_alu_funct,
28     output reg [1:0] IDEX_alu_1_src,
29     output reg      IDEX_alu_2_src,
30     output reg      IDEX_pc_add_src,
31     output reg      IDEX_mem_read,
32     output reg      IDEX_mem_write,
33     output reg [1:0] IDEX_reg_src,
34     output reg      IDEX_reg_write,
35     output reg [31:0] IDEX_pc,
36     output reg [31:0] IDEX_pc_next,
37     output reg      IDEX_btb_hit,
38     output reg [2:0] IDEX_funct3,
39     output reg [31:0] IDEX_reg_1,
40     output reg [31:0] IDEX_reg_2,
41     output reg [31:0] IDEX_imm,
42     output reg [4:0] IDEX_rs_1,
43     output reg [4:0] IDEX_rs_2,
44     output reg [4:0] IDEX_rd
45 );
46     always @(posedge clk or posedge rst) begin
47         if (rst | clr) begin
```

```

48         IDEX_jump      ≤ 1'h0;
49         IDEX_branch    ≤ 1'h0;
50         IDEX_alu_funct  ≤ 4'h0;
51         IDEX_alu_1_src  ≤ 2'h0;
52         IDEX_alu_2_src  ≤ 1'h0;
53         IDEX_pc_add_src ≤ 1'h0;
54         IDEX_mem_read   ≤ 1'h0;
55         IDEX_mem_write  ≤ 1'h0;
56         IDEX_reg_src    ≤ 2'h0;
57         IDEX_reg_write  ≤ 1'h0;
58         IDEX_pc         ≤ 32'h0;
59         IDEX_pc_next    ≤ 32'h0;
60         IDEX_btb_hit    ≤ 1'h0;
61         IDEX_funct3     ≤ 3'h0;
62         IDEX_reg_1      ≤ 32'h0;
63         IDEX_reg_2      ≤ 32'h0;
64         IDEX_imm        ≤ 32'h0;
65         IDEX_rs_1       ≤ 5'h0;
66         IDEX_rs_2       ≤ 5'h0;
67         IDEX_rd         ≤ 5'h0;
68     end
69     else begin
70         IDEX_jump      ≤ jump;
71         IDEX_branch    ≤ branch;
72         IDEX_alu_funct  ≤ alu_funct;
73         IDEX_alu_1_src  ≤ alu_1_src;
74         IDEX_alu_2_src  ≤ alu_2_src;
75         IDEX_pc_add_src ≤ pc_add_src;
76         IDEX_mem_read   ≤ mem_read;
77         IDEX_mem_write  ≤ mem_write;
78         IDEX_reg_src    ≤ reg_src;
79         IDEX_reg_write  ≤ reg_write;
80         IDEX_pc         ≤ IFID_pc;
81         IDEX_pc_next    ≤ IFID_pc_next;
82         IDEX_btb_hit    ≤ IFID_btb_hit;
83         IDEX_funct3     ≤ funct3;
84         IDEX_reg_1      ≤ reg_1;
85         IDEX_reg_2      ≤ reg_2;
86         IDEX_imm        ≤ imm;
87         IDEX_rs_1       ≤ rs_1;
88         IDEX_rs_2       ≤ rs_2;
89         IDEX_rd         ≤ rd;
90     end
91 end
92 endmodule

```

### 6.15.3 EX\_MEM.v

```
1  module ex_mem(
2      input          clk,
3      input          rst,
4      input          IDEX_mem_read,
5      input          IDEX_mem_write,
6      input [1:0]    IDEX_reg_src,
7      input          IDEX_reg_write,
8      input [2:0]    IDEX_funct3,
9      input [31:0]   alu_out,
10     input [31:0]   IDEX_pc_next,
11     input [31:0]   EX_reg_2,
12     input [4:0]    IDEX_rd,
13     output reg     EXMEM_mem_read,
14     output reg     EXMEM_mem_write,
15     output reg [1:0] EXMEM_reg_src,
16     output reg     EXMEM_reg_write,
17     output reg [2:0] EXMEM_funct3,
18     output reg [31:0] EXMEM_alu_out,
19     output reg [31:0] EXMEM_pc_next,
20     output reg [31:0] EXMEM_reg_2,
21     output reg [4:0] EXMEM_rd
22 );
23     always @(posedge clk or posedge rst) begin
24         if (rst) begin
25             EXMEM_mem_read  ≤ 1'h0;
26             EXMEM_mem_write ≤ 1'h0;
27             EXMEM_reg_src   ≤ 2'h0;
28             EXMEM_reg_write ≤ 1'h0;
29             EXMEM_funct3    ≤ 3'h0;
30             EXMEM_alu_out   ≤ 32'h0;
31             EXMEM_pc_next   ≤ 32'h0;
32             EXMEM_reg_2     ≤ 32'h0;
33             EXMEM_rd        ≤ 5'h0;
34         end
35         else begin
36             EXMEM_mem_read  ≤ IDEX_mem_read;
37             EXMEM_mem_write ≤ IDEX_mem_write;
38             EXMEM_reg_src   ≤ IDEX_reg_src;
39             EXMEM_reg_write ≤ IDEX_reg_write;
40             EXMEM_funct3    ≤ IDEX_funct3;
41             EXMEM_alu_out   ≤ alu_out;
42             EXMEM_pc_next   ≤ IDEX_pc_next;
43             EXMEM_reg_2     ≤ EX_reg_2;
44             EXMEM_rd        ≤ IDEX_rd;
45         end
46     end
47 endmodule
```

#### 6.13.4 MEM\_WB.v

```
1  module mem_wb(
2      input          clk,
3      input          rst,
4      input          [1:0] EXMEM_reg_src,
5      input          EXMEM_reg_write,
6      input          [31:0] mem_out,
7      input          [31:0] EXMEM_pc_next,
8      input          [31:0] EXMEM_alu_out,
9      input          [4:0] EXMEM_rd,
10     output reg [1:0] MEMWB_reg_src,
11     output reg      MEMWB_reg_write,
12     output reg [31:0] MEMWB_mem_out,
13     output reg [31:0] MEMWB_pc_next,
14     output reg [31:0] MEMWB_alu_out,
15     output reg [4:0] MEMWB_rd
16 );
17     always @(posedge clk or posedge rst) begin
18         if (rst) begin
19             MEMWB_reg_src    ≤ 2'h0;
20             MEMWB_reg_write ≤ 1'h0;
21             MEMWB_mem_out    ≤ 32'h0;
22             MEMWB_pc_next    ≤ 32'h0;
23             MEMWB_alu_out    ≤ 32'h0;
24             MEMWB_rd         ≤ 5'h0;
25         end
26         else begin
27             MEMWB_reg_src    ≤ EXMEM_reg_src;
28             MEMWB_reg_write ≤ EXMEM_reg_write;
29             MEMWB_mem_out    ≤ mem_out;
30             MEMWB_pc_next    ≤ EXMEM_pc_next;
31             MEMWB_alu_out    ≤ EXMEM_alu_out;
32             MEMWB_rd         ≤ EXMEM_rd;
33         end
34     end
35 endmodule
```