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

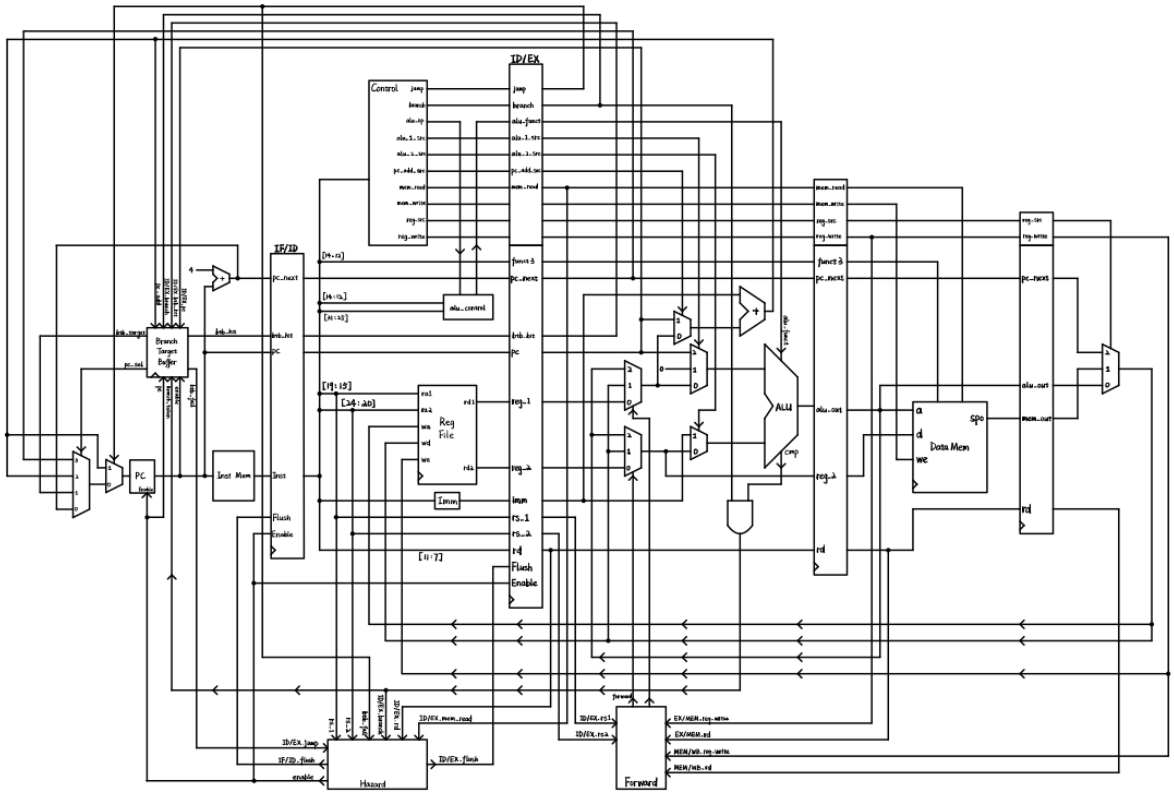
- Xilinx Vivado v2019.1
- Microsoft Visual Studio Code
- FPGAOL

4 实验过程

整个项目结构如下

```
main (Main.v)
├── PDU:  pdu  (PDU.v)
└── CPU:  cpu  (CPU.v)
      ├── Inst_Mem:       rom            (rom.xci)
      ├── Data_Mem:       data_mem       (Data_Mem.v)
      │     └── Data_Mem: dual_port_ram  (dual_port_ram.xci)
      ├── Control:        control        (Ctrl.v)
      ├── ALU_control:    alu_control    (ALU_Ctrl.v)
      ├── Reg_File:       regfile        (Reg_File.v)
      ├── Imm_Gen:        imm_gen        (Imm_Gen.v)
      ├── ALU:            alu            (ALU.v)
      ├── Forward:        forward        (forward.v)
      ├── Hazard:         hazard         (hazard.v)
      ├── BTB:            btb            (BTB.v)
      ├── IF_ID:          if_id          (IF_ID.v)
      ├── ID_EX:          id_ex          (ID_EX.v)
      ├── EX_MEM:         ex_mem         (EX_MEM.v)
      └── 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_funct` 信号, 控制 ALU 的行为. 其中 `alu_funct` 信号的低 3 位为 `funct3`, 第四位视情况而定, 具体如下

alu_funct	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_funct` 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

```
always @(*) begin
    case (IDEX_alu_1_src)
        2'b00:   alu_in_1 = EX_reg_1;
        2'b01:   alu_in_1 = 32'h0;
        2'b10:   alu_in_1 = IDEX_pc;
        default: alu_in_1 = 32'h0;
    endcase
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

```
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

```
always @(*) begin
    if (mem_read) begin
        case (funct3)
            3'b000:
            case (addr[1:0])
                2'b00: mem_data = {{24{data_out[7]}}, data_out[7:0]};
                2'b01: mem_data = {{24{data_out[15]}}, data_out[15:8]};
                2'b10: mem_data = {{24{data_out[23]}}, data_out[23:16]};
                2'b11: mem_data = {{24{data_out[31]}}, data_out[31:24]};
            endcase
            3'b001:
            case (addr[1])
                1'b0: mem_data = {{16{data_out[15]}}, data_out[15:0]};
                1'b1: mem_data = {{16{data_out[31]}}, data_out[31:16]};
            endcase
            3'b010: mem_data = data_out;
            3'b100:
            case (addr[1:0])
                2'b00: mem_data = {24'h0, data_out[7:0]};
                2'b01: mem_data = {24'h0, data_out[15:8]};
                2'b10: mem_data = {24'h0, data_out[23:16]};
```

```
                2'b11: mem_data = {24'h0, data_out[31:24]};
            endcase
            3'b101:
            case (addr[1])
                1'b0: mem_data = {16'h0, data_out[15:0]};
                1'b1: mem_data = {16'h0, data_out[31:16]};
            endcase
            default: mem_data = data_out;
        endcase
    end
    else begin
        mem_data = data_out;
    end
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

```
always @(*) begin
    if (mem_we) begin
        case(funct3)
            3'b000:
            case(addr[1:0])
                2'b00: data_in = {data_out[31:8], data[7:0]};
                2'b01: data_in = {data_out[31:16], data[7:0], data_out[7:0]};
                2'b10: data_in = {data_out[31:24], data[7:0], data_out[15:0]};
                2'b11: data_in = {data[7:0], data_out[23:0]};
            endcase
            3'b001:
            case (addr[1])
                1'b0: data_in = {data_out[31:16], data[15:0]};
                1'b1: data_in = {data[15:0], data_out[15:0]};
            endcase
            3'b010: data_in = data;
            default: data_in = data_out;
        endcase
    end
    else begin
        data_in = data_out;
    end
end
```

4.2 分支预测

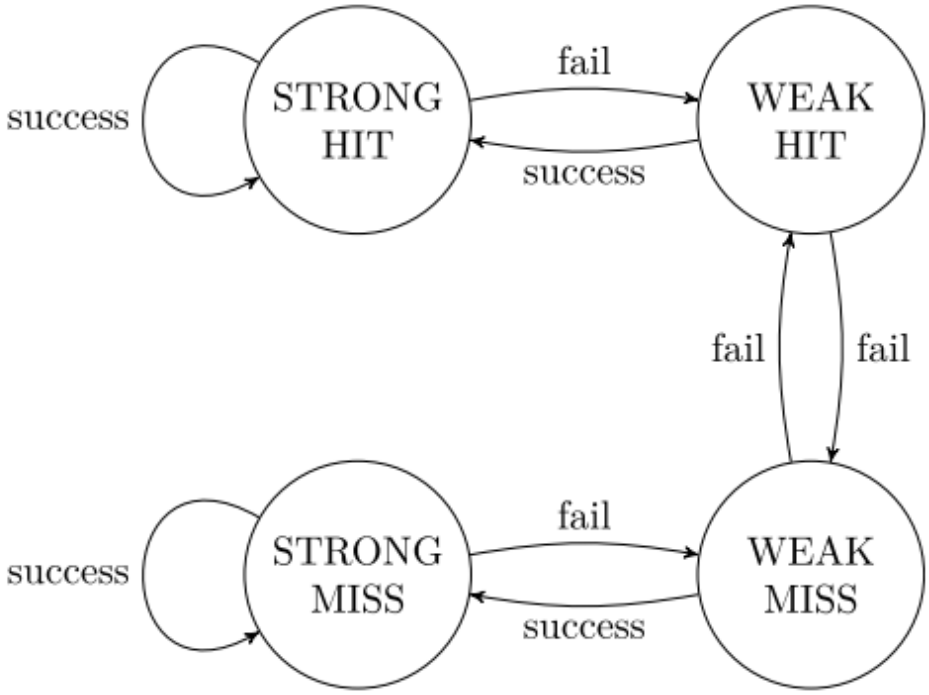
本次实验在 IF 段增加了一个分支预测模块, 其中有一个 cache 存储分支指令的相关信息, 每个 cache 数据块有四个部分, 分别为:

- 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:

```
// Buffer Register
reg [TAG_LEN - 1 : 0] tag    [BUF_SIZE - 1 : 0];
reg [31:0]            target [BUF_SIZE - 1 : 0];
reg                   valid  [BUF_SIZE - 1 : 0];
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

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

```
        btb_target = 32'h0;
    end
    else if (valid[pc_idx] &&
             (pc_tag == tag[pc_idx]) &&
             state[pc_idx][1]) begin
        btb_hit    = 1'b1;
        btb_target = target[pc_idx];
    end
    else begin
        btb_hit    = 1'b0;
        btb_target = 32'h0;
    end
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

```
integer i = 0;
always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < BUF_SIZE; i = i + 1) begin
            tag[i]    <= 0;
            target[i] <= 32'h0;
            valid[i]  <= 1'b0;
            state[i]  <= WEAK_MISS;
        end
    end
    else if (enable && IDEX_branch) begin
        // tag matched, update state
        if ((tag[IDEX_pc_idx] == IDEX_pc_tag) && valid[IDEX_pc_idx])
begin
            case (state[IDEX_pc_idx])
            STRONG_HIT:
                state[IDEX_pc_idx] <= btb_fail ? WEAK_HIT  :
STRONG_HIT;
            WEAK_HIT:
                state[IDEX_pc_idx] <= btb_fail ? WEAK_MISS :
STRONG_HIT;
            WEAK_MISS:
                state[IDEX_pc_idx] <= btb_fail ? WEAK_HIT  :
STRONG_MISS;
            STRONG_MISS:
```

```
                state[IDEX_pc_idx] <= btb_fail ? WEAK_MISS :
STRONG_MISS;
            endcase
        end
        // tag not matched, change buffer
        else begin
            tag[IDEX_pc_idx]    <= IDEX_pc_tag;
            target[IDEX_pc_idx] <= IDEX_branch_target;
            valid[IDEX_pc_idx]  <= 1'b1;
            state[IDEX_pc_idx]  <= WEAK_MISS;
        end
    end
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

```
reg btb_p_nt;  // branch predicted but not taken
reg btb_np_t;  // not branch predicted but taken

assign btb_fail = btb_p_nt | btb_np_t;

always @(*) begin
    if (rst) begin
        btb_p_nt  = 1'b0;
        btb_np_t  = 1'b0;
    end
    else begin
        btb_p_nt = (IDEX_btb_hit) & (~branch_taken);
        btb_np_t = (~IDEX_btb_hit) & (branch_taken);
    end
end
// PC Controller
always @(*) begin
    if (btb_p_nt)      pc_sel <= 2'b11;  // IDEX_pc_next
    else if (btb_np_t) pc_sel <= 2'b10;  // pc_add
    else if (btb_hit)  pc_sel <= 2'b01;  // btb_target
    else               pc_sel <= 2'b00;  // pc_next
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

```
// Counter Part
always @(posedge clk or posedge rst) begin
    if (rst) begin
        branch_cnt    ≤ 32'h0;
        btb_succ_cnt  ≤ 32'h0;
        btb_fail_cnt  ≤ 32'h0;
    end
    else begin
        if (IDEX_branch) begin
            branch_cnt    ≤ branch_cnt    + 32'h1;
            if (btb_fail) btb_fail_cnt ≤ btb_fail_cnt + 32'h1;
            else          btb_succ_cnt ≤ btb_succ_cnt + 32'h1;
        end

    end
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

```
always @(*) begin
    if (IDEX_jump) pc_in = pc_add;
    else begin
        case (pc_sel)
            2'b11: pc_in = IDEX_pc_next;
            2'b10: pc_in = pc_add;
            2'b01: pc_in = btb_target;
            2'b00: pc_in = pc_next;
        endcase
    end
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

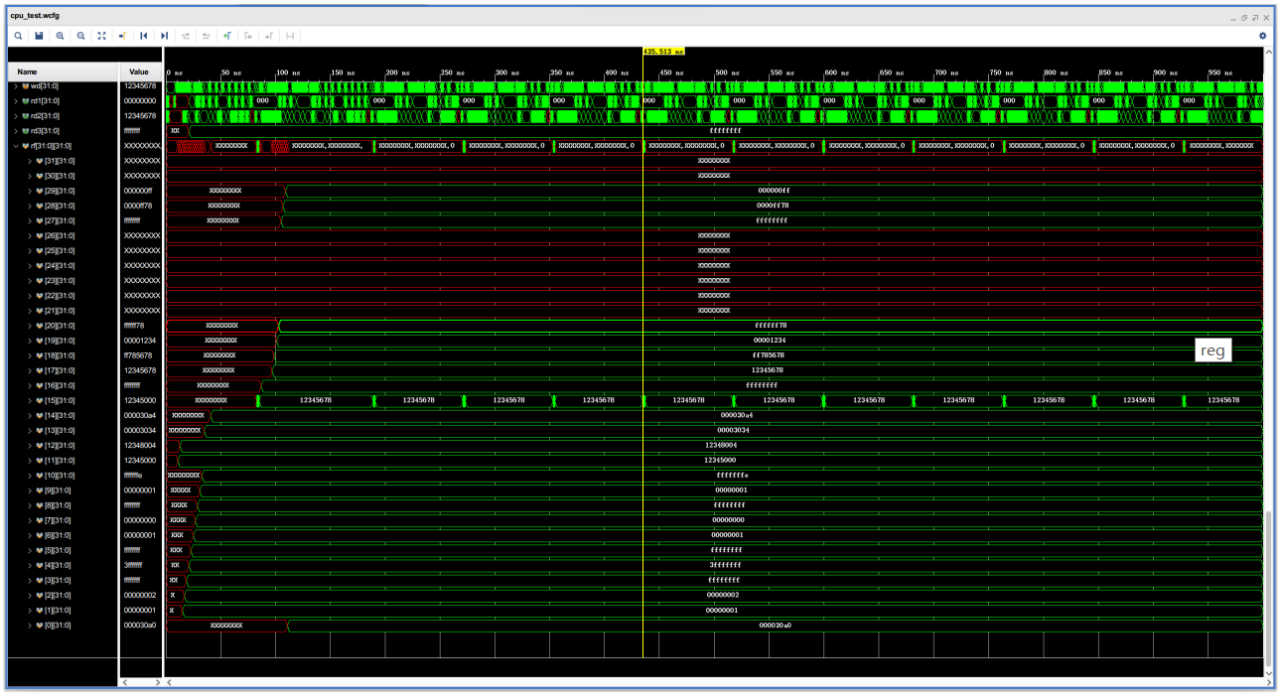
5.1 测试指令集

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

```
# begins at 0x3000
start:
  # test for lui & auipc
  lui   x11, 0x12345 # x11 = 0x12345000 305,418,240
  auipc x12, 0x12345 # x12 = 0x12348004 305,430,532

  # test for arithmetic & logical operations
  addi  x1,  x0, 1  # x1 =  0x00000001
  slli  x2,  x1, 1  # x2 =  0x00000002
  sub   x3,  x1, x2 # x3 =  0xFFFFFFFF
  srl   x4,  x3, x2 # x4 =  0x3FFFFFFF
  srai  x5,  x3, 2  # x5 =  0xFFFFFFFF
  slt   x6,  x5, x4 # x6 =  0x00000001
  sltu  x7,  x5, x4 # x7 =  0x00000000
  or    x8,  x5, x1 # x8 =  0xFFFFFFFF
  and   x9,  x5, x1 # x9 =  0x00000001
  xor   x10, x5, x1 # x10 = 0xFFFFFFFE

  # test for jal & jalr
  jal x13, jal_test # x13 = 0x00003034

  # test for branch
  beq  x1, x1, bne_test
  addi x21, x0, 1       # if beq failed, x21 = 1
bne_test:
  bne  x1, x0, blt_test
  addi x22, x0, 1       # if bne failed, x22 = 1
blt_test:
  blt  x0, x1, bge_test
  addi x23, x0, 1       # if blt failed, x23 = 1
bge_test:
  bge  x1, x0, bltu_test
  addi x24, x0, 1       # if bge failed, x24 = 1
bltu_test:
  bltu x4, x3, bgeu_test
  addi x25, x0, 1       # if bltu failed, x25 = 1
bgeu_test:
  bgeu x3, x4, ls_test
  addi x26, x0, 1       # if bgeu failed, x26 = 1

```

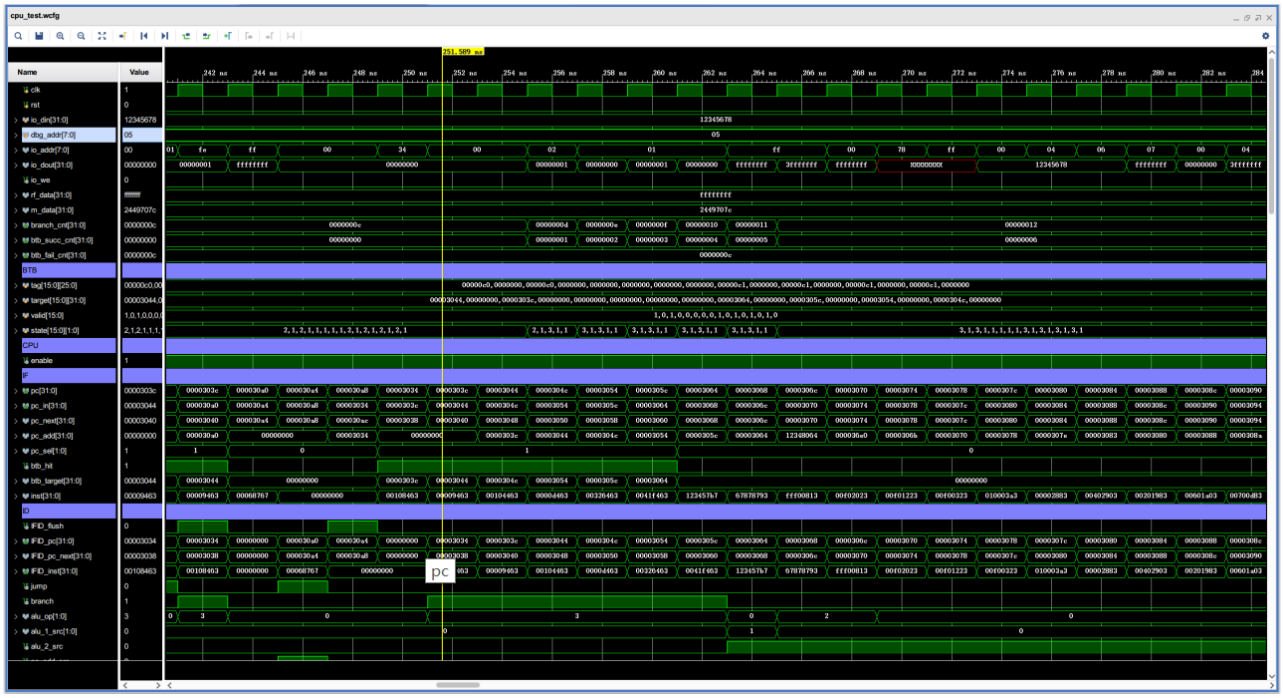
```
41  # test for load & store
42  ls_test:
43    li  x15, 0x12345678
44    li  x16, 0xFFFFFFFF
45    sw  x15, 0x000(x0)  # mem[0] = 0x12345678
46    sh  x15, 0x004(x0)
47    sb  x15, 0x006(x0)
48    sb  x16, 0x007(x0)  # mem[1] = 0xFF785678
49    lw  x17, 0x000(x0)  # x17 = 0x12345678
50    lw  x18, 0x004(x0)  # x18 = 0xFF785678
51    lh  x19, 0x002(x0)  # x19 = 0x00001234
52    lh  x20, 0x006(x0)  # x20 = 0xFFFFFF78
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.

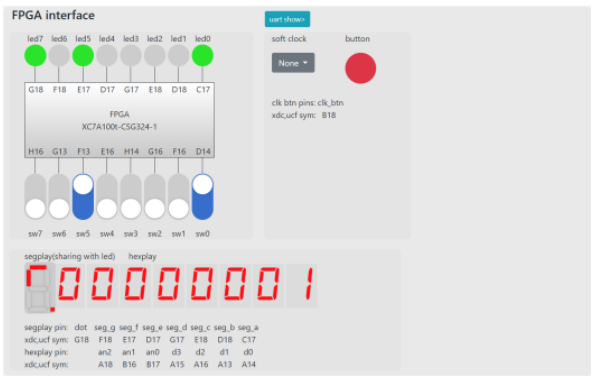


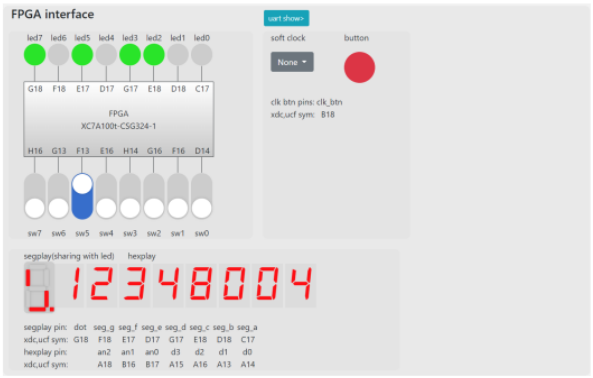
而观察 pc 部分如下, 可以看到进行分支预测 (`btb_hit` 有效) 时, pc 并不是每次都加 4, 而且没有冲刷流水线的痕迹, 观察计数器, 预测失败的次数在最后稳定到了 `0x0000000c` (12), 与预期也相同: 六条分支跳转指令, 第一轮添加到 cache, 默认状态为 WEAK_MISS, 所以第二轮预测不跳转, 共 12

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

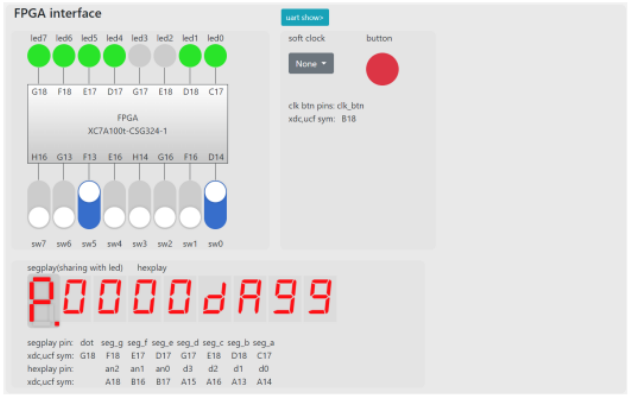
```
# memory starts at 0x000, ends at 0x3FF
    li   t0, 1024    # t0 represents for size
    add  x1, x0, x0  # x1 = 0, represents for i

loop1:
    mv   x2, x1      # x2 = i, represents for j
    lw   t1, 0(x1)   # t1 = mem[i]

loop2:
    lw   t2, 0(x2)   # t2 = mem[j]
    bge  t2, t1, loop2_end  # if mem[i] ≥ mem[j], jump to loop2_end
    sw   t1, 0(x2)   # mem[j] = mem[i]
    sw   t2, 0(x1)   # mem[i] = mem[j]
    add  t1, x0, t2  # t1 = new mem[i]
loop2_end:
    addi x2, x2, 4   # j++
    blt  x2, t0, loop2

    addi x1, x1, 4      # i++
    blt  x1, t0, loop1

    # Get the counters
```

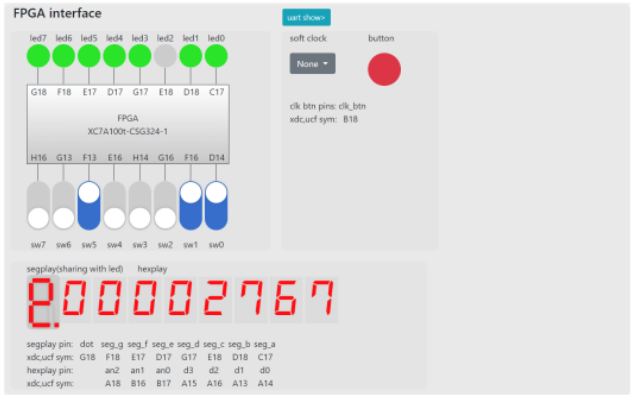
```
    li   s4, 0x800
    lw   s0, 0(s4)  # s0 = branch_count
    lw   s1, 4(s4)  # s1 = success_count
    lw   s2, 8(s4)  # s2 = fail_count

    # Signal the end of the program
    sw   x0, 0x408(x0)  # segplay echoes 0

end:
    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.

```
from random import randrange
import sys
with open('.\\CPU\\CPU.srcs\\sources_1\\coes\\random_mem.coe', 'w') as f:
    print('memory_initialization_radix=16;', file=f)
    print('memory_initialization_vector=', file=f)

    for i in range(256):
        print('%x' % randrange(2147483647), file=f)

    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

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

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

```
module main(
    input         clk,    // clk_100MHz
    input         step,   // btn
    input         rst,    // sw[7]
    input         run,    // sw[6]
    input         valid,  // sw[5]
    input  [4:0]  in,     // sw[4:0]
    output        ready,  // led[7]
    output [1:0]  check,  // led[6:5]
    output [4:0]  out_0,  // led[4:0]
    output [2:0]  an,     // segplay_an
    output [3:0]  seg     // segplay_data
);
    wire clk_cpu;

    // IO_BUS
    wire        io_we;
    wire [7:0]  io_addr;
    wire [31:0] io_din;
    wire [31:0] io_dout;

    // Debug_BUS
    wire [31:0] rf_data;
    wire [31:0] m_data;
    wire [7:0]  dbg_addr;

    // BTB Counter
    wire [31:0] branch_cnt;
    wire [31:0] btb_succ_cnt;
    wire [31:0] btb_fail_cnt;

    // Pipeline Register
    wire [31:0] pc_in;
    wire [31:0] pc;
    wire [31:0] IFID_pc;
    wire [31:0] IFID_inst;
    wire [31:0] IDEX_pc;
    wire [31:0] IDEX_reg_1;
    wire [31:0] IDEX_reg_2;
    wire [31:0] IDEX_imm;
```

```
    wire [4:0]  IDEX_rd;
    wire [31:0] EXMEM_alu_out;
    wire [31:0] EXMEM_reg_2;
    wire [4:0]  EXMEM_rd;
    wire [31:0] MEMWB_alu_out;
    wire [31:0] MEMWB_mem_out;
    wire [4:0]  MEMWB_rd;

    // Wire Mapping
    cpu CPU(
        .clk           (clk_cpu),
        .rst           (rst),
        .io_we         (io_we),
        .io_addr       (io_addr),
        .io_din        (io_din),
        .io_dout       (io_dout),
        .rf_data       (rf_data),
        .m_data        (m_data),
        .dbg_addr      (dbg_addr),
        .branch_cnt    (branch_cnt),
        .btb_succ_cnt  (btb_succ_cnt),
        .btb_fail_cnt  (btb_fail_cnt),
        .pc_in         (pc_in),
        .pc            (pc),
        .IFID_pc       (IFID_pc),
        .IFID_inst     (IFID_inst),
        .IDEX_pc       (IDEX_pc),
        .IDEX_reg_1    (IDEX_reg_1),
        .IDEX_reg_2    (IDEX_reg_2),
        .IDEX_imm      (IDEX_imm),
        .IDEX_rd       (IDEX_rd),
        .EXMEM_alu_out (EXMEM_alu_out),
        .EXMEM_reg_2   (EXMEM_reg_2),
        .EXMEM_rd      (EXMEM_rd),
        .MEMWB_alu_out (MEMWB_alu_out),
        .MEMWB_mem_out (MEMWB_mem_out),
        .MEMWB_rd      (MEMWB_rd)
    );

    pdu PDU(
        .clk       (clk),
        .rst       (rst),
        .run       (run),
        .step      (step),
        .clk_cpu   (clk_cpu),
        .valid     (valid),
        .in        (in),
        .ready     (ready),
```

```
        .check     (check),
        .out0      (out_0),
        .an        (an),
        .seg       (seg),
        .io_we     (io_we),
        .io_addr   (io_addr),
        .io_din    (io_din),
        .io_dout   (io_dout),
        .rf_data   (rf_data),
        .m_data    (m_data),
        .m_rf_addr (dbg_addr),
        .pcin      (pc_in),
        .pc        (pc),
        .pcd       (IFID_pc),
        .pce       (IDEX_pc),
        .ir        (IFID_inst),
        .imm       (IDEX_imm),
        .mdr       (MEMWB_mem_out),
        .a         (IDEX_reg_1),
        .b         (IDEX_reg_2),
        .y         (EXMEM_alu_out),
        .bm        (EXMEM_reg_2),
        .yw        (MEMWB_alu_out),
        .rd        (IDEX_rd),
        .rdm       (MEMWB_rd),
        .rdw       (EXMEM_rd),
        .br_cnt    (branch_cnt),
        .succ_cnt  (btb_succ_cnt),
        .fail_cnt  (btb_fail_cnt)
    );
endmodule
```

6.2 PDU.v

```
module  pdu(
    input clk,
    input rst,

    //选择CPU工作方式;
    input run,
    input step,
    output clk_cpu,

    //输入switch的端口
    input valid,
    input [4:0] in,

    //输出led和seg的端口
    output [1:0] check,  //led6-5:查看类型
```

```
    output [4:0] out0,    //led4-0
    output [2:0] an,      //8个数码管
    output [3:0] seg,
    output ready,         //led7

    //IO_BUS
    input          io_we,
    input  [7:0]   io_addr,
    input  [31:0]  io_dout,
    output [31:0]  io_din,

    //Debug_BUS
    input      [31:0] rf_data,
    input      [31:0] m_data,
    output reg [7:0]  m_rf_addr,

    //增加流水线寄存器调试接口
    input [31:0] pcin, pc, pcd, pce,
    input [31:0] ir, imm, mdr,
    input [31:0] a, b, y, bm, yw,
    input [4:0]  rd, rdm, rdw,
    input [31:0] br_cnt, succ_cnt, fail_cnt

    // 串口输入输出
);

    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
    reg run_r, step_r, step_2r, valid_r, valid_2r;
    wire step_p, valid_pn;     //取边沿信号
    wire pre_pn,next_pn;       //增加取边沿信号

    reg clk_cpu_r;       //寄存器输出CPU时钟
    reg [4:0] out0_r;    //输出外设端口
    reg [31:0] out1_r;
    reg ready_r;
    reg [19:0] cnt;      //刷新计数器，刷新频率约为95Hz
    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

    reg [7:0] io_din_a; //_a表示为满足组合always描述要求定义的，下同
    reg [4:0] out0_a;
    reg [31:0] out1_a;
    reg [3:0] seg_a;

    //增加pre,next取边沿计数器
    reg [4:0] cnt_m_rf;      //寄存器堆和存储器地址计数器
    reg [1:0] cnt_ah_plr;    //流水线寄存器高两位地址计数器
    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
```

```
    end

    //写外设端口
    always @(posedge clk, posedge rst) begin
        if (rst) begin
            out0_r ≤ 5'h1f;
            out1_r ≤ 32'h1234_5678;
            ready_r ≤ 1'b1;
        end
        else if (io_we)
        case (io_addr)
            8'h00:
                out0_r  ≤ io_dout[4:0];
            8'h04:
                ready_r ≤ io_dout[0];
            8'h08:
                out1_r  ≤ io_dout;
            default:
                ;
        endcase
    end

    //增加寄存器堆和存储器地址计数：依靠pre,next边沿计数使能
    always @(posedge clk, posedge rst) begin
        if (rst)
            cnt_m_rf ≤ 5'b0_0000;
        else if (step_p)
            cnt_m_rf ≤ 5'b0_0000;
        else if (next_pn)
            cnt_m_rf ≤ cnt_m_rf + 5'b0_0001;
        else if (pre_pn)
            cnt_m_rf ≤ cnt_m_rf - 5'b0_0001;
    end

    //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
    always @(posedge clk, posedge rst) begin
        if (rst)
            cnt_ah_plr ≤ 2'b00;
        else if (step_p)
            cnt_ah_plr ≤ 2'b00;
        else if (pre_pn)
            cnt_ah_plr ≤ cnt_ah_plr + 2'b01;
    end

    always @(posedge clk, posedge rst) begin
        if (rst)
            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;
```

```
                    3'b001:
                        plr_data = a;
                    3'b010:
                        plr_data = b;
                    3'b011:
                        plr_data = imm;
                    3'b100:
                        plr_data = {{27{1'b0}},rd};
                    3'b101:
                        plr_data = br_cnt;
                    default:
                        plr_data = pce;
                endcase
            end
            //EX/MEM
            2'b10:
            case (cnt_al_plr[1:0])
                2'b00:
                    plr_data = y;
                2'b01:
                    plr_data = bm;
                2'b10:
                    plr_data = {{27{1'b0}},rdm};
                2'b11:
                    plr_data = succ_cnt;
            endcase
            //MEM/WB
            2'b11:
            case (cnt_al_plr[1:0])
                2'b00:
                    plr_data = yw;
                2'b01:
                    plr_data = mdr;
                2'b10:
                    plr_data = {{27{1'b0}},rdw};
                2'b11:
                    plr_data = fail_cnt;
            endcase
        endcase
    end

    //LED和数码管查看类型
    always @(posedge clk, posedge rst) begin
        if(rst)
            check_r <= 2'b00;
        else if(run_r)
            check_r <= 2'b00;
        else if (step_p)
```

```
            check_r <= 2'b00;
        else if (valid_pn)
            check_r <= check - 2'b01;
    end

    //LED和数码管显示内容
    always @(*) begin
        case (check_r)
            2'b00: begin
                out0_a = out0_r;
                out1_a = out1_r;
            end
            2'b01: begin
                out0_a = cnt_m_rf;
                out1_a = rf_data;
            end
            2'b10: begin
                out0_a = cnt_m_rf;
                out1_a = m_data;
            end
            2'b11: begin
                out0_a = addr_plr;
                out1_a = plr_data;     //更改为流水线寄存器地址和数据显示
            end
        endcase
    end

    //扫描数码管
    always @(posedge clk, posedge rst) begin
        if (rst)
            cnt <= 20'h0_0000;
        else
            cnt <= cnt + 20'h0_0001;
    end

    always @* begin
        case (an)
            3'd0:
                seg_a = out1_a[3:0];
            3'd1:
                seg_a = out1_a[7:4];
            3'd2:
                seg_a = out1_a[11:8];
            3'd3:
                seg_a = out1_a[15:12];
            3'd4:
                seg_a = out1_a[19:16];
            3'd5:
```

```
                seg_a = out1_a[23:20];
            3'd6:
                seg_a = out1_a[27:24];
            3'd7:
                seg_a = out1_a[31:28];
            default:
                ;
        endcase
    end

endmodule
```

6.3 CPU.v

```
module cpu(
    input clk,
    input rst,

    // IO_BUS
    input      [31:0] io_din,    // 来自 sw 的输入数据
    output     [7:0]  io_addr,   // led 和 seg 的地址
    output     [31:0] io_dout,   // 输出 led 和 seg 的数据
    output            io_we,     // 输出 led 和 seg 数据时的使能信号

    // Debug_BUS
    input      [7:0]  dbg_addr,  // 存储器 (MEM) 或寄存器堆 (RF) 的调试读口地址
    output     [31:0] rf_data,   // 从RF读取的数据
    output     [31:0] m_data,    // 从MEM读取的数据

    // BTB Counter
    output     [31:0] branch_cnt,
    output     [31:0] btb_succ_cnt,
    output     [31:0] btb_fail_cnt,

    // PC/IF/ID 流水段寄存器
    output reg [31:0] pc,
    output reg [31:0] pc_in,
    output     [31:0] IFID_pc,
    output     [31:0] IFID_inst,

    // ID/EX 流水段寄存器
    output     [31:0] IDEX_pc,
    output     [31:0] IDEX_reg_1,
    output     [31:0] IDEX_reg_2,
    output     [31:0] IDEX_imm,
    output     [4:0]  IDEX_rd,

    // EX/MEM 流水段寄存器
```

```
    output      [31:0] EXMEM_alu_out,
    output      [31:0] EXMEM_reg_2,
    output      [4:0]  EXMEM_rd,

    // MEM/WB 流水段寄存器
    output      [31:0] MEMWB_alu_out,
    output      [31:0] MEMWB_mem_out,
    output      [4:0]  MEMWB_rd
);
    // Control Signal
    wire        enable;

    // IF: pc, inst_mem, btb
//    reg  [31:0] pc;
//    reg  [31:0] pc_in;
    wire [31:0] pc_next;
    wire [31:0] pc_add;
    wire [31:0] inst;
    wire [1:0]  pc_sel;
    wire        btb_hit;
    wire [31:0] btb_target;
    wire        btb_fail;
//    wire [31:0] branch_cnt;
//    wire [31:0] btb_succ_cnt;
//    wire [31:0] btb_fail_cnt;


    // IF/ID
    wire        IFID_flush;
//    wire [31:0] IFID_pc;
    wire [31:0] IFID_pc_next;
//    wire [31:0] IFID_inst;
    wire        IFID_btb_hit;

    // ID
    wire        jump;
    wire        branch;
    wire [1:0]  alu_op;
    wire [1:0]  alu_1_src;
    wire        alu_2_src;
    wire        pc_add_src;
    wire        mem_read;
    wire        mem_write;
    wire [1:0]  reg_src;
    wire        reg_write;
    wire [3:0]  alu_funct;
    wire [2:0]  funct3;
    wire [31:0] reg_1;
```

```
    wire [31:0] reg_2;
    wire [4:0]  rs_1;
    wire [4:0]  rs_2;
    wire [4:0]  rd;
    wire [31:0] imm;

    // ID/EX
    wire        IDEX_flush;
    wire        IDEX_jump;
    wire        IDEX_branch;
    wire [3:0]  IDEX_alu_funct;
    wire [1:0]  IDEX_alu_1_src;
    wire        IDEX_alu_2_src;
    wire        IDEX_pc_add_src;
    wire        IDEX_mem_read;
    wire        IDEX_mem_write;
    wire [1:0]  IDEX_reg_src;
    wire        IDEX_reg_write;
    wire [2:0]  IDEX_funct3;
//    wire [31:0] IDEX_pc;
    wire [31:0] IDEX_pc_next;
    wire        IDEX_btb_hit;
//    wire [31:0] IDEX_reg_1;
//    wire [31:0] IDEX_reg_2;
//    wire [31:0] IDEX_imm;
//    wire [4:0]  IDEX_rd;
    wire [4:0]  IDEX_rs_1;
    wire [4:0]  IDEX_rs_2;

    // EX
    wire [1:0]  forward_1;
    wire [1:0]  forward_2;
    reg  [31:0] EX_reg_1;
    reg  [31:0] EX_reg_2;
    reg  [31:0] alu_in_1;
    wire [31:0] alu_in_2;
    wire [31:0] alu_out;
    wire        cmp;
    wire        branch_taken;

    // EX/MEM
    wire [31:0] EXMEM_pc_next;
    wire        EXMEM_mem_read;
    wire        EXMEM_mem_write;
    wire [1:0]  EXMEM_reg_src;
    wire        EXMEM_reg_write;
    wire [2:0]  EXMEM_funct3;
//    wire [31:0] EXMEM_alu_out;
```

```
//    wire [31:0] EXMEM_reg_2;
//    wire [4:0]  EXMEM_rd;

    // MEM
    wire        mem_we;
    wire [31:0] mem_out;
    wire [31:0] mem_data;
    // MEM/WB
    wire [1:0]  MEMWB_reg_src;
    wire        MEMWB_reg_write;
    wire [31:0] MEMWB_pc_next;
//    wire [31:0] MEMWB_mem_out;
//    wire [31:0] MEMWB_alu_out;
//    wire [4:0]  MEMWB_rd;
    // WB
    reg  [31:0] WB_data;

    // IF: pc, inst_mem
    assign pc_next = pc + 32'h4;

    always @(*) begin
        if (IDEX_jump) pc_in = pc_add;
        else begin
            case (pc_sel)
                2'b11: pc_in = IDEX_pc_next;
                2'b10: pc_in = pc_add;
                2'b01: pc_in = btb_target;
                2'b00: pc_in = pc_next;
            endcase
        end
    end

    always @(posedge clk or posedge rst) begin
        if (rst)         pc ≤ 32'h3000;
        else if (enable) pc ≤ pc_in;
    end

    btb #(.BUF_LEN(4)) BTB (
        .clk                (clk),
        .rst                (rst),
        .enable             (enable),
        .pc                 (pc[31:2]),
        .IDEX_pc            (IDEX_pc[31:2]),
        .IDEX_branch        (IDEX_branch),
        .branch_taken       (branch_taken),
        .IDEX_branch_target (pc_add),
        .IDEX_btb_hit       (IDEX_btb_hit),
        .pc_sel             (pc_sel),
```

```
        .btb_hit            (btb_hit),
        .btb_target         (btb_target),
        .btb_fail           (btb_fail),
        .branch_cnt         (branch_cnt),
        .btb_succ_cnt       (btb_succ_cnt),
        .btb_fail_cnt       (btb_fail_cnt)
    );

    rom Inst_Mem (
        .a   (pc[9:2]),
        .spo (inst)
    );
    // IF/ID Register
    if_id IF_ID (
        .clk          (clk),
        .rst          (rst),
        .en           (enable),
        .clr          (IFID_flush),
        .pc           (pc),
        .pc_next      (pc_next),
        .btb_hit      (btb_hit),
        .inst         (inst),
        .IFID_pc      (IFID_pc),
        .IFID_pc_next (IFID_pc_next),
        .IFID_btb_hit (IFID_btb_hit),
        .IFID_inst    (IFID_inst)
    );
    // ID: hazard, ctrl, regfile, imm_gen
    assign funct3 = IFID_inst[14:12];
    assign rs_1   = IFID_inst[19:15];
    assign rs_2   = IFID_inst[24:20];
    assign rd     = IFID_inst[11:7];

    hazard Hazard (
        .rs_1          (rs_1),
        .rs_2          (rs_2),
        .btb_fail      (btb_fail),
        .IDEX_jump     (IDEX_jump),
        .IDEX_rd       (IDEX_rd),
        .IDEX_mem_read (IDEX_mem_read),
        .enable        (enable),
        .IFID_flush    (IFID_flush),
        .IDEX_flush    (IDEX_flush)
    );

    control Control (
        .inst       (IFID_inst[6:0]),
        .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),
```

```
        .IDEX_rd         (IDEX_rd),
        .EXMEM_mem_read  (EXMEM_mem_read),
        .EXMEM_mem_write (EXMEM_mem_write),
        .EXMEM_reg_src   (EXMEM_reg_src),
        .EXMEM_reg_write (EXMEM_reg_write),
        .EXMEM_funct3    (EXMEM_funct3),
        .EXMEM_alu_out   (EXMEM_alu_out),
        .EXMEM_pc_next   (EXMEM_pc_next),
        .EXMEM_reg_2     (EXMEM_reg_2),
        .EXMEM_rd        (EXMEM_rd)
    );

    // MEM: data_mem
    assign io_we   = EXMEM_mem_write & EXMEM_alu_out[10];
    assign io_addr = EXMEM_alu_out[7:0];
    assign io_dout = EXMEM_reg_2;
    assign mem_we  = EXMEM_mem_write & (~EXMEM_alu_out[10]);
    assign mem_out = EXMEM_alu_out[10] ? io_din : mem_data;

    data_mem Data_Mem (
        .clk      (clk),
        .funct3   (EXMEM_funct3),
        .addr     (EXMEM_alu_out[9:0]),
        .data     (EXMEM_reg_2),
        .mem_read (EXMEM_mem_read),
        .mem_we   (mem_we),
        .dbg_addr (dbg_addr),
        .mem_data (mem_data),
        .dbg_data (m_data)
    );

    // MEM/WB Register
    mem_wb MEM_WB (
        .clk             (clk),
        .rst             (rst),
        .EXMEM_reg_src   (EXMEM_reg_src),
        .EXMEM_reg_write (EXMEM_reg_write),
        .mem_out         (mem_out),
        .EXMEM_pc_next   (EXMEM_pc_next),
        .EXMEM_alu_out   (EXMEM_alu_out),
        .EXMEM_rd        (EXMEM_rd),
        .MEMWB_reg_src   (MEMWB_reg_src),
        .MEMWB_reg_write (MEMWB_reg_write),
        .MEMWB_mem_out   (MEMWB_mem_out),
        .MEMWB_pc_next   (MEMWB_pc_next),
        .MEMWB_alu_out   (MEMWB_alu_out),
        .MEMWB_rd        (MEMWB_rd)
    );
```

```
    // WB
    always @(*) begin
        case (MEMWB_reg_src)
            2'b00:   WB_data = MEMWB_alu_out;
            2'b01:   WB_data = MEMWB_mem_out;
            2'b10:   WB_data = MEMWB_pc_next;
            default: WB_data = 32'h0;
        endcase
    end
endmodule
```

6.4 Data_Mem.v

```
module data_mem(
    input             clk,
    input      [2:0]  funct3,
    input      [9:0]  addr,
    input      [31:0] data,
    input             mem_read,
    input             mem_we,
    input      [7:0]  dbg_addr,
    output reg [31:0] mem_data,
    output     [31:0] dbg_data
);
    reg  [31:0] data_in;
    wire [31:0] data_out;

    always @(*) begin
        if (mem_we) begin
            case(funct3)
                3'b000:
                case(addr[1:0])
                    2'b00: data_in = {data_out[31:8], data[7:0]};
                    2'b01: data_in = {data_out[31:16], data[7:0],
data_out[7:0]};
                    2'b10: data_in = {data_out[31:24], data[7:0],
data_out[15:0]};
                    2'b11: data_in = {data[7:0], data_out[23:0]};
                endcase
                3'b001:
                case (addr[1])
                    1'b0: data_in = {data_out[31:16], data[15:0]};
                    1'b1: data_in = {data[15:0], data_out[15:0]};
                endcase
                3'b010: data_in = data;
                default: data_in = data_out;
            endcase
        end
        else begin
```

```
            data_in = data_out;
        end
    end

    always @(*) begin
        if (mem_read) begin
            case (funct3)
                3'b000:
                case (addr[1:0])
                    2'b00: mem_data = {{24{data_out[7]}}, data_out[7:0]};
                    2'b01: mem_data = {{24{data_out[15]}}, data_out[15:8]};
                    2'b10: mem_data = {{24{data_out[23]}}, data_out[23:16]};
                    2'b11: mem_data = {{24{data_out[31]}}, data_out[31:24]};
                endcase
                3'b001:
                case (addr[1])
                    1'b0: mem_data = {{16{data_out[15]}}, data_out[15:0]};
                    1'b1: mem_data = {{16{data_out[31]}}, data_out[31:16]};
                endcase
                3'b010: mem_data = data_out;
                3'b100:
                case (addr[1:0])
                    2'b00: mem_data = {24'h0, data_out[7:0]};
                    2'b01: mem_data = {24'h0, data_out[15:8]};
                    2'b10: mem_data = {24'h0, data_out[23:16]};
                    2'b11: mem_data = {24'h0, data_out[31:24]};
                endcase
                3'b101:
                case (addr[1])
                    1'b0: mem_data = {16'h0, data_out[15:0]};
                    1'b1: mem_data = {16'h0, data_out[31:16]};
                endcase
                default: mem_data = data_out;
            endcase
        end
        else begin
            mem_data = data_out;
        end
    end

    dual_port_ram Data_Mem(
        .clk  (clk),
```

```
        .we   (mem_we),
        .a    (addr[9:2]),
        .dpra (dbg_addr),
        .d    (data_in),
        .spo  (data_out),
        .dpo  (dbg_data)
    );
endmodule
```

6.5 Ctrl.v

关于 Ctrl.v 的介绍见 control 模块

```
module control(
    input  [6:0] inst,
    output       jump,
    output       branch,
    output [1:0] alu_op,     // 2'b11 for branch, 2'b10 for imm, 2'b01 for reg
    output [1:0] alu_1_src,  // 2'b10 for auipc, 2'b01 for lui
    output       alu_2_src,
    output       pc_add_src, // 1'b1 for jalr
    output       mem_read,
    output       mem_write,
    output [1:0] reg_src,
    output       reg_write
);
    wire lui;
    wire auipc;
    wire load;
    wire store;
    wire reg_arith;
    wire imm_arith;
    assign lui       = (inst == 7'b0110111);
    assign auipc     = (inst == 7'b0010111);
    assign load      = (inst == 7'b0000011);
    assign store     = (inst == 7'b0100011);
    assign reg_arith = (inst == 7'b0110011);
    assign imm_arith = (inst == 7'b0010011);

    assign jump       = (inst == 7'b1101111) | (inst == 7'b1100111);
    assign branch     = (inst == 7'b1100011);
    assign alu_op     = {(imm_arith | branch), (reg_arith | branch)};
    assign alu_1_src  = {auipc, lui};
    assign alu_2_src  = (imm_arith | load | store | lui | auipc);
    assign pc_add_src = (inst == 7'b1100111);
    assign mem_read   = (load);
    assign mem_write  = (store);
    assign reg_src    = {jump, load};
```

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

6.6 ALU_Ctrl.v

关于 ALU_control.v 的介绍见 alu_control 模块

```
module alu_control(
    input      [1:0] alu_op,
    input      [2:0] funct3,
    input            funct7,
    output reg [3:0] alu_funct
);
    always @(*) begin
        case (alu_op)
            2'b00:   alu_funct = 4'b1111;
            2'b01:   alu_funct = {funct7, funct3};
            2'b10:   alu_funct = {((funct3 == 3'b101) & funct7), funct3};
            2'b11:   alu_funct = {1'b0, funct3};
            default: alu_funct = 4'b1111;
        endcase
    end
endmodule
```

6.7 Reg_File.v

```
module regfile (
    input             clk,
    input             we,
    input      [4:0]  ra1,
    input      [4:0]  ra2,
    input      [4:0]  ra3,
    input      [4:0]  wa,
    input      [31:0] wd,
    output reg [31:0] rd1,
    output reg [31:0] rd2,
    output reg [31:0] rd3
);
    reg [31:0] rf [31:0];

    always @(posedge clk) begin
        if (we) rf[wa] ≤ wd;
    end

    always @(*) begin
        if (ra1 == 5'h0)    rd1 = 32'h0;
        else if (ra1 == wa) rd1 = wd;
```

```
        else                rd1 = rf[ra1];
    end

    always @(*) begin
        if (ra2 == 5'h0)    rd2 = 32'h0;
        else if (ra2 == wa) rd2 = wd;
        else                rd2 = rf[ra2];
    end

    always @(*) begin
        if (ra3 == 5'h0)    rd3 = 32'h0;
        else if (ra3 == wa) rd3 = wd;
        else                rd3 = rf[ra3];
    end
endmodule
```

6.8 Imm_Gen.v

```
module imm_gen(
    input      [31:0] inst,
    output reg [31:0] imm
);
    always @(*) begin
        case (inst[6:0])
            7'b0110111, // lui
            7'b0010111: // auipc
                imm = {inst[31:12], 12'h0000};

            7'b1101111: // jal
                imm = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};

            7'b1100011: // branch
                imm = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};

            7'b1100111, // jalr
            7'b0000011: // load
                imm = {{21{inst[31]}}, inst[30:20]};

            7'b0010011: // imm arithmetic
                imm = (inst[13:12] == 2'b01) ?
                      {27'h0, inst[24:20]} :
                      {{21{inst[31]}}, inst[30:20]};

            7'b0100011: // store
                imm = {{21{inst[31]}}, inst[30:25], inst[11:7]};

            default:
```

```
                imm = 32'h0;
        endcase
    end
endmodule
```

6.9 ALU.v

```
module alu(
    input      [31:0] in_1,
    input      [31:0] in_2,
    input      [3:0]  funct,
    output reg [31:0] arithmetic,
    output reg        comparision
);
    always @(*) begin
        case (funct)
            4'b0000: arithmetic = in_1 + in_2;
            4'b0001: arithmetic = in_1 << in_2;
            4'b0010: arithmetic = $signed(in_1) < $signed(in_2);
            4'b0011: arithmetic = in_1 < in_2;
            4'b0100: arithmetic = in_1 ^ in_2;
            4'b0101: arithmetic = in_1 >> in_2;
            4'b0110: arithmetic = in_1 | in_2;
            4'b0111: arithmetic = in_1 & in_2;
            4'b1000: arithmetic = in_1 - in_2;
            4'b1101: arithmetic = $signed(in_1) >>> in_2;
            4'b1111: arithmetic = in_1 + in_2;
            default: arithmetic = 32'h0;
        endcase
    end

    always @(*) begin
        case (funct)
            4'b0000: comparision = in_1 == in_2;
            4'b0001: comparision = in_1 ≠ in_2;
            4'b0100: comparision = $signed(in_1) < $signed(in_2);
            4'b0101: comparision = $signed(in_1) ≥ $signed(in_2);
            4'b0110: comparision = in_1 < in_2;
            4'b0111: comparision = in_1 ≥ in_2;
            default: comparision = 1'b0;
        endcase
    end
endmodule
```

6.10 Forward.v

```
module forward(
    input      [4:0]  IDEX_rs_1,
    input      [4:0]  IDEX_rs_2,
    input             EXMEM_reg_write,
    input      [4:0]  EXMEM_rd,
    input             MEMWB_reg_write,
    input      [4:0]  MEMWB_rd,
    output reg [1:0]  forward_1,
    output reg [1:0]  forward_2
);
    always @(*) begin
        if (EXMEM_reg_write & (EXMEM_rd ≠ 5'h0) & (EXMEM_rd == IDEX_rs_1))
            forward_1 = 2'b10;
        else if (MEMWB_reg_write & (MEMWB_rd ≠ 5'h0) & (MEMWB_rd == IDEX_rs_1))
            forward_1 = 2'b01;
        else
            forward_1 = 2'b00;
    end
    always @(*) begin
        if (EXMEM_reg_write & (EXMEM_rd ≠ 5'h0) & (EXMEM_rd == IDEX_rs_2))
            forward_2 = 2'b10;
        else if (MEMWB_reg_write & (MEMWB_rd ≠ 5'h0) & (MEMWB_rd == IDEX_rs_2))
            forward_2 = 2'b01;
        else
            forward_2 = 2'b00;
    end
endmodule
```

6.11 Hazard.v

```
module hazard(
    input [4:0] rs_1,
    input [4:0] rs_2,
    input       btb_fail,
    input       IDEX_jump,
    input [4:0] IDEX_rd,
    input       IDEX_mem_read,
    output reg  enable,
    output      IFID_flush,
    output      IDEX_flush
);
    always @(*) begin
        if ((IDEX_mem_read) & (IDEX_rd ≠ 5'h0) &
```

```
            (IDEX_rd == rs_1 | IDEX_rd == rs_2)
        )
            enable = 1'b0;
        else
            enable = 1'b1;
    end
    assign IFID_flush = btb_fail | IDEX_jump;
    assign IDEX_flush = ~enable | btb_fail | IDEX_jump;
endmodule
```

6.12 BTB.v

关于 BTB.v 的介绍见分支预测

```
module btb #(
    parameter BUF_LEN = 4
) (
    input             clk,
    input             rst,
    input             enable,
    input      [29:0] pc,
    input      [29:0] IDEX_pc,
    input             IDEX_branch,
    input             branch_taken,
    input      [31:0] IDEX_branch_target,
    input             IDEX_btb_hit,
    output reg [1:0]  pc_sel,
    output reg        btb_hit,
    output reg [31:0] btb_target,
    output            btb_fail,

    // Counters
    output reg [31:0] branch_cnt,
    output reg [31:0] btb_succ_cnt,
    output reg [31:0] btb_fail_cnt
);
    localparam TAG_LEN  = 30 - BUF_LEN;
    localparam BUF_SIZE = 1 << BUF_LEN;

    localparam STRONG_HIT  = 2'b11;
    localparam WEAK_HIT    = 2'b10;
    localparam WEAK_MISS   = 2'b01;
    localparam STRONG_MISS = 2'b00;

    // Buffer Register
    reg [TAG_LEN - 1 : 0] tag    [BUF_SIZE - 1 : 0];
    reg [31:0]            target [BUF_SIZE - 1 : 0];
    reg                   valid  [BUF_SIZE - 1 : 0];
    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
```

```
            for (i = 0; i < BUF_SIZE; i = i + 1) begin
                tag[i]    <= 0;
                target[i] <= 32'h0;
                valid[i]  <= 1'b0;
                state[i]  <= WEAK_MISS;
            end
        end
        else if (enable && IDEX_branch) begin
            // tag matched, update state
            if ((tag[IDEX_pc_idx] == IDEX_pc_tag) &&
valid[IDEX_pc_idx]) begin
                case (state[IDEX_pc_idx])
                STRONG_HIT:
                    state[IDEX_pc_idx] <= btb_fail ? WEAK_HIT  :
STRONG_HIT;
                WEAK_HIT:
                    state[IDEX_pc_idx] <= btb_fail ? WEAK_MISS :
STRONG_HIT;
                WEAK_MISS:
                    state[IDEX_pc_idx] <= btb_fail ? WEAK_HIT  :
STRONG_MISS;
                STRONG_MISS:
                    state[IDEX_pc_idx] <= btb_fail ? WEAK_MISS :
STRONG_MISS;
                endcase
            end
            // tag not matched, change buffer
            else begin
                tag[IDEX_pc_idx]    <= IDEX_pc_tag;
                target[IDEX_pc_idx] <= IDEX_branch_target;
                valid[IDEX_pc_idx]  <= 1'b1;
                state[IDEX_pc_idx]  <= WEAK_MISS;
            end
        end
    end
    // PC Controller
    always @(*) begin
        if (btb_p_nt)      pc_sel <= 2'b11;  // IDEX_pc_next
        else if (btb_np_t) pc_sel <= 2'b10;  // pc_add
        else if (btb_hit)  pc_sel <= 2'b01;  // btb_target
        else               pc_sel <= 2'b00;  // pc_next
    end

    // Counter Part
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            branch_cnt   <= 32'h0;
            btb_succ_cnt <= 32'h0;
```

```
            btb_fail_cnt <= 32'h0;
        end
        else begin
            if (IDEX_branch) begin
                branch_cnt   <= branch_cnt   + 32'h1;
                if (btb_fail) btb_fail_cnt <= btb_fail_cnt + 32'h1;
                else          btb_succ_cnt <= btb_succ_cnt + 32'h1;
            end

        end
    end
endmodule  //btb
```

6.13 流水段寄存器

6.13.1 IF_ID.v

```
module if_id(
    input             clk,
    input             rst,
    input             en,
    input             clr,
    input      [31:0] pc,
    input      [31:0] pc_next,
    input             btb_hit,
    input      [31:0] inst,
    output reg [31:0] IFID_pc,
    output reg [31:0] IFID_pc_next,
    output reg        IFID_btb_hit,
    output reg [31:0] IFID_inst
);
    always @(posedge clk or posedge rst) begin
        if (rst | clr) begin
            IFID_pc      <= 32'h0;
            IFID_pc_next <= 32'h0;
            IFID_btb_hit <= 1'h0;
            IFID_inst    <= 32'h0;
        end
        else if (en) begin
            IFID_pc      <= pc;
            IFID_pc_next <= pc_next;
            IFID_btb_hit <= btb_hit;
            IFID_inst    <= inst;
        end
    end
endmodule
```

6.13.2 ID_EX.v

```
module id_ex(
    input             clk,
    input             rst,
    input             clr,
    input             jump,
    input             branch,
    input      [3:0]  alu_funct,
    input      [1:0]  alu_1_src,
    input             alu_2_src,
    input             pc_add_src,
    input             mem_read,
    input             mem_write,
    input      [1:0]  reg_src,
    input             reg_write,
    input      [31:0] IFID_pc,
    input      [31:0] IFID_pc_next,
    input             IFID_btb_hit,
    input      [2:0]  funct3,
    input      [31:0] reg_1,
    input      [31:0] reg_2,
    input      [31:0] imm,
    input      [4:0]  rs_1,
    input      [4:0]  rs_2,
    input      [4:0]  rd,
    output reg        IDEX_jump,
    output reg        IDEX_branch,
    output reg [3:0]  IDEX_alu_funct,
    output reg [1:0]  IDEX_alu_1_src,
    output reg        IDEX_alu_2_src,
    output reg        IDEX_pc_add_src,
    output reg        IDEX_mem_read,
    output reg        IDEX_mem_write,
    output reg [1:0]  IDEX_reg_src,
    output reg        IDEX_reg_write,
    output reg [31:0] IDEX_pc,
    output reg [31:0] IDEX_pc_next,
    output reg        IDEX_btb_hit,
    output reg [2:0]  IDEX_funct3,
    output reg [31:0] IDEX_reg_1,
    output reg [31:0] IDEX_reg_2,
    output reg [31:0] IDEX_imm,
    output reg [4:0]  IDEX_rs_1,
    output reg [4:0]  IDEX_rs_2,
    output reg [4:0]  IDEX_rd
);
    always @(posedge clk or posedge rst) begin
        if (rst | clr) begin
```

```
            IDEX_jump       <= 1'h0;
            IDEX_branch     <= 1'h0;
            IDEX_alu_funct  <= 4'h0;
            IDEX_alu_1_src  <= 2'h0;
            IDEX_alu_2_src  <= 1'h0;
            IDEX_pc_add_src <= 1'h0;
            IDEX_mem_read   <= 1'h0;
            IDEX_mem_write  <= 1'h0;
            IDEX_reg_src    <= 2'h0;
            IDEX_reg_write  <= 1'h0;
            IDEX_pc         <= 32'h0;
            IDEX_pc_next    <= 32'h0;
            IDEX_btb_hit    <= 1'h0;
            IDEX_funct3     <= 3'h0;
            IDEX_reg_1      <= 32'h0;
            IDEX_reg_2      <= 32'h0;
            IDEX_imm        <= 32'h0;
            IDEX_rs_1       <= 5'h0;
            IDEX_rs_2       <= 5'h0;
            IDEX_rd         <= 5'h0;
        end
        else begin
            IDEX_jump       <= jump;
            IDEX_branch     <= branch;
            IDEX_alu_funct  <= alu_funct;
            IDEX_alu_1_src  <= alu_1_src;
            IDEX_alu_2_src  <= alu_2_src;
            IDEX_pc_add_src <= pc_add_src;
            IDEX_mem_read   <= mem_read;
            IDEX_mem_write  <= mem_write;
            IDEX_reg_src    <= reg_src;
            IDEX_reg_write  <= reg_write;
            IDEX_pc         <= IFID_pc;
            IDEX_pc_next    <= IFID_pc_next;
            IDEX_btb_hit    <= IFID_btb_hit;
            IDEX_funct3     <= funct3;
            IDEX_reg_1      <= reg_1;
            IDEX_reg_2      <= reg_2;
            IDEX_imm        <= imm;
            IDEX_rs_1       <= rs_1;
            IDEX_rs_2       <= rs_2;
            IDEX_rd         <= rd;
        end
    end
endmodule
```

6.15.3 EX_MEM.v

```
module ex_mem(
    input             clk,
    input             rst,
    input             IDEX_mem_read,
    input             IDEX_mem_write,
    input      [1:0]  IDEX_reg_src,
    input             IDEX_reg_write,
    input      [2:0]  IDEX_funct3,
    input      [31:0] alu_out,
    input      [31:0] IDEX_pc_next,
    input      [31:0] EX_reg_2,
    input      [4:0]  IDEX_rd,
    output reg        EXMEM_mem_read,
    output reg        EXMEM_mem_write,
    output reg [1:0]  EXMEM_reg_src,
    output reg        EXMEM_reg_write,
    output reg [2:0]  EXMEM_funct3,
    output reg [31:0] EXMEM_alu_out,
    output reg [31:0] EXMEM_pc_next,
    output reg [31:0] EXMEM_reg_2,
    output reg [4:0]  EXMEM_rd
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            EXMEM_mem_read  <= 1'h0;
            EXMEM_mem_write <= 1'h0;
            EXMEM_reg_src   <= 2'h0;
            EXMEM_reg_write <= 1'h0;
            EXMEM_funct3    <= 3'h0;
            EXMEM_alu_out   <= 32'h0;
            EXMEM_pc_next   <= 32'h0;
            EXMEM_reg_2     <= 32'h0;
            EXMEM_rd        <= 5'h0;
        end
        else begin
            EXMEM_mem_read  <= IDEX_mem_read;
            EXMEM_mem_write <= IDEX_mem_write;
            EXMEM_reg_src   <= IDEX_reg_src;
            EXMEM_reg_write <= IDEX_reg_write;
            EXMEM_funct3    <= IDEX_funct3;
            EXMEM_alu_out   <= alu_out;
            EXMEM_pc_next   <= IDEX_pc_next;
            EXMEM_reg_2     <= EX_reg_2;
            EXMEM_rd        <= IDEX_rd;
        end
    end
endmodule
```

6.13.4 MEM_WB.v

```
module mem_wb(
    input             clk,
    input             rst,
    input      [1:0]  EXMEM_reg_src,
    input             EXMEM_reg_write,
    input      [31:0] mem_out,
    input      [31:0] EXMEM_pc_next,
    input      [31:0] EXMEM_alu_out,
    input      [4:0]  EXMEM_rd,
    output reg [1:0]  MEMWB_reg_src,
    output reg        MEMWB_reg_write,
    output reg [31:0] MEMWB_mem_out,
    output reg [31:0] MEMWB_pc_next,
    output reg [31:0] MEMWB_alu_out,
    output reg [4:0]  MEMWB_rd
);
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            MEMWB_reg_src   <= 2'h0;
            MEMWB_reg_write <= 1'h0;
            MEMWB_mem_out   <= 32'h0;
            MEMWB_pc_next   <= 32'h0;
            MEMWB_alu_out   <= 32'h0;
            MEMWB_rd        <= 5'h0;
        end
        else begin
            MEMWB_reg_src   <= EXMEM_reg_src;
            MEMWB_reg_write <= EXMEM_reg_write;
            MEMWB_mem_out   <= mem_out;
            MEMWB_pc_next   <= EXMEM_pc_next;
            MEMWB_alu_out   <= EXMEM_alu_out;
            MEMWB_rd        <= EXMEM_rd;
        end
    end
endmodule
```