

# Lab 3 : 动态分支预测

陈祎伟

3230102357

2025年4月9日

## 1 设计思路

### 1.1 BHT模块

BHT 模块用于存储历史跳转信息。

在 BHT 的实现中，我使用了类似哈希表的方式来存储跳转信息。BHT 的大小为 256，使用 8 位索引来访问。

具体实现如下：

#### 1. 模块接口定义以及初始化：

- 模块接口定义如下：

```
1  module BHT(  
2      input clk,  
3      input rst,  
4      input [31:0] PC_IF,  
5      input bht_wen,  
6      input taken,  
7      output wire bht_pridict  
8  );
```

其中，PC\_IF 为当前指令地址，bht\_wen 为写使能信号，taken 为 ID 阶段真实跳转结果，bht\_pridict 为预测结果。

- 定义参数如下：

```
1  parameter Pridict_Strongly_Taken = 2'b11;  
2  parameter Pridict_Weakly_Taken = 2'b10;  
3  parameter Pridict_Weakly_Not_Taken = 2'b01;  
4  parameter Pridict_Strongly_Not_Taken = 2'b00;  
5  
6  parameter BHT_SIZE = 256; // Number of entries in the BHT
```

同时，将 BHT 各项的初始值设置为 Pridict\_Weakly\_Not\_Taken。

## 2. 输出预测结果 bht\_pridict 的实现:

```
1 assign bht_pridict = bht[PC_IF[9:2]] >= Pridict_Weakly_Taken;
```

这里将 PC\_IF / 4，即指令数，取低 8 位作为 BHT 的索引。

当 bht 中的值大于等于 Pridict\_Weakly\_Taken 时，即给出预测值 1，否则为 0。

## 3. BHT 的更新:

由于 BHT 是基于 PC\_IF 的预测，这里使用 reg 将 PC\_IF 进行存储。

```
1 reg [31:0] PC_IF_reg;
2 initial begin
3     PC_IF_reg = 32'b0; // Initialize the PC_IF register
4 end
5 always @(posedge clk or posedge rst) begin
6     if (rst) begin
7         PC_IF_reg <= 32'b0; // Reset the PC_IF register
8     end else begin
9         PC_IF_reg <= PC_IF; // Update the PC_IF register
10    end
11 end
```

在接收到 bht\_wen 信号时，根据 PC\_IF\_reg 和实际跳转情况 taken，更新 BHT 的值:

```
1 always @(posedge clk or posedge rst) begin
2     if (rst) begin
3         for (i = 0; i < BHT_SIZE; i = i + 1) begin
4             bht[i] <= Pridict_Weakly_Not_Taken; // Reset all entries to Weakly Not Taken
5         end
6     end else begin
7         if (bht_wen) begin
8             if (taken) begin
9                 bht[PC_IF_reg[9:2]] <= (bht[PC_IF_reg[9:2]] == Pridict_Strongly_Taken) ?
10                    Pridict_Strongly_Taken : bht[PC_IF_reg[9:2]] + 1; // Increment if
11                    taken
12            end else begin
13                bht[PC_IF_reg[9:2]] <= (bht[PC_IF_reg[9:2]] ==
14                    Pridict_Strongly_Not_Taken) ? Pridict_Strongly_Not_Taken : bht[
15                    PC_IF_reg[9:2]] - 1; // Decrement if not taken
16            end
17        end
18    end
19 end
```

## 1.2 BTB 模块

BTB 模块用于存储跳转指令的目标地址。

在 BTB 的实现中，由于跳转指令数相对较少，我直接使用了一个 256 条目的数组来存储跳转指令的目标地址。

具体实现如下：

#### 1. 模块接口定义以及初始化：

- 模块接口定义如下：

```
1  module BTB(  
2      input  clk,  
3      input  rst,  
4      input  [31:0] PC_IF,  
5      input  [31:0] PC_ID,  
6      input  [31:0] PC_jump,  
7      input  btb_wen,  
8      output reg hit,  
9      output reg [31:0] btb_rdata  
10 );
```

其中，PC\_jump 为跳转指令的目标地址，btb\_wen 为写使能信号；

btb\_rdata 为读取的跳转地址，hit 为命中信号，用于标识 btb\_rdata 是否有效。

- 参数定义如下：

```
1  parameter BTB_SIZE = 256; // Number of entries in the BTB  
2  parameter BTB_INDEX_WIDTH = 8; // Index width for the BTB  
3  
4  reg [31:0] btb_fetch_pc [0:BTB_SIZE-1]; // BTB entries for fetched PC  
5  reg [31:0] btb_predict_pc [0:BTB_SIZE-1]; // BTB entries for predicted PC  
6  reg [BTB_INDEX_WIDTH-1:0] btb_top; // Top index for the BTB
```

这里，btb\_fetch\_pc 用于存储跳转指令的地址，btb\_predict\_pc 用于存储跳转指令的目标地址。

btb\_top 用于存储 BTB 最近的空闲位置。

- BTB 的初始化：

这里，我将 BTB 的所有条目初始化为 FFFFFFFF (-1)，避免与其他指令冲突。

#### 2. 输出 btb\_rdata 以及 BTB 更新的实现：

读写 BTB 的时，均采用遍历的方式对 BTB 表进行查找。

- BTB 的读取：

```
1  always @* begin  
2      hit = 1'b0; // Initialize hit flag to 0  
3      for (i = 0; i < BTB_SIZE; i = i + 1) begin  
4          if (btb_fetch_pc[i] == PC_IF) begin  
5              btb_rdata = btb_predict_pc[i]; // Output the predicted PC if a match is  
              found
```

```

6         hit = 1'b1; // Set hit flag to 1
7     end
8 end
9     if (hit == 1'b0) begin
10         btb_rdata = 32'b0; // If no match, output 0
11     end
12 end

```

这里，hit 信号用于标识 BTB 是否命中。

- BTB 的更新:

在接收到 btb\_wen 信号时，遍历 BTB 表，若存在与 PC\_ID 相同的条目，则更新该条目的预测地址；

否则，利用 btb\_top 将新的跳转指令添加到 BTB 表中。

```

1     integer j;
2     always @(posedge clk or posedge rst) begin
3         if (rst) begin
4             for (i = 0; i < BTB_SIZE; i = i + 1) begin
5                 btb_fetch_pc[i] <= -1; // Reset all entries to -1
6                 btb_predict_pc[i] <= -1; // Reset all entries to -1
7                 btb_top <= 0; // Reset top index to 0
8             end
9         end else begin
10            if (btb_wen) begin
11                j = 0; // Initialize j to 0
12                for (i = 0; i < btb_top; i = i + 1) begin
13                    if (btb_fetch_pc[i] == PC_ID) begin
14                        btb_predict_pc[i] <= PC_jump; // Update the predicted PC if a
15                                                         match is found
16                        j = 1; // Set j to 1 to indicate a match
17                    end
18                end
19                if (j == 0) begin
20                    if (btb_top < BTB_SIZE) begin
21                        btb_fetch_pc[btb_top] <= PC_ID; // Add new entry to the BTB
22                        btb_predict_pc[btb_top] <= PC_jump; // Set the predicted PC for
23                                                         the new entry
24                        btb_top <= btb_top + 1; // Increment the top index
25                    end
26                end
27            end
28        end
29    end

```

### 1.3 流水线的修改

#### 1. jump\_right 的生成:

jump\_right 信号在 ID 阶段生成，用于标识 IF 阶段的预测是否正确。

```
1 assign jump_right = Branch_ctrl ? (PC_IF == jump_PC_ID) : (~last_Branch_predict || ~
    last_btb_hit);
```

当实际跳转(Branch\_ctrl)时，若 PC\_IF 与 jump\_PC\_ID 相同，说明预测正确；

否则，说明预测错误。

当实际不跳转时，若 IF 阶段的预测结果为不跳转(!last\_Branch\_predict)或 BTB 未命中(!last\_btb\_hit)，说明预测正确；

否则，说明预测错误。

## 2. next\_PC\_IF 的生成：

### • final\_PC\_IF 的生成：

```
1 MUX2T1_32 mux_branch_predict(.IO(PC_4_IF),.I1(PC_predict),.s(Branch_predict &&
    btb_hit),.o(final_PC_IF)); // choose the predicted PC or the next PC
```

这里，PC\_predict 为 BTB 的预测地址，Branch\_predict 为 BHT 的预测结果，btb\_hit 为 BTB 的命中信号，控制 PC\_predict 是否有效。

当 Branch\_predict 和 btb\_hit 均为 1 时，选择 BTB 的预测地址；否则，选择 PC\_4\_IF。

### • next\_PC\_IF 的生成：

```
1 assign next_PC_IF = Branch_ctrl ? jump_right ? final_PC_IF : // jump right
2                               jump_PC_ID : // not jump when branch
3                               jump_right ? final_PC_IF : // jump right
4                               reg_FD_flush_of_predict_reg ?
5                               final_PC_IF : // flush
                                                                PC_ID +
                                                                4;
```

当实际跳转(Branch\_ctrl)时，若之前预测并跳转正确(jump\_right)，则跳转到 final\_PC\_IF；否则，仍需要进行跳转操作，跳转至 jump\_PC\_ID。

当实际不跳转时，若之前预测正确并不进行跳转，或者前一个周期 PC\_IF 被冲刷，则跳转到 final\_PC\_IF；

否则，需要将错误的 PC\_IF 进行修正，跳转至 PC\_ID + 4。

## 3. BHT 与 BTB 模块的接入：

### • BHT 模块的接入：

```

1      BHT bht(.clk(debug_clk),.rst(rst),.PC_IF(PC_IF),.bht_wen(is_jump && PC_EN_IF),
2      .taken(Branch_ctrl && PC_EN_IF),.bht_predict(Branch_predict));

```

其中，目前为任意跳转指令时(is\_jump)均进行 BHT 的更新，同时考虑 PC\_EN\_IF 信号为 0 时，当前的预测对 PC 并没有影响，是无效的。

因而更新使能信号 bht\_wen 为 is\_jump && PC\_EN\_IF。

- BTB 模块的接入：

```

1      BTB btb(.clk(debug_clk),.rst(rst),.PC_IF(PC_IF),.PC_ID(PC_ID),.PC_jump(
2      jump_PC_ID),
3      .btb_wen(btb_wen),.hit(btb_hit),.btb_rdata(PC_predict));
4      assign btb_wen = Branch_ctrl && ~last_btb_hit;

```

其中，更新使能信号 btb\_wen 为 Branch\_ctrl && !last\_btb\_hit，即发生跳转且 BTB 未命中时，才进行 BTB 的更新。

#### 4. IF ID 寄存器 flush 控制信号的修改：

```

1      assign reg_FD_flush_of_predict = ~jump_right_reg ? Branch_ctrl ? ~jump_right :
2      PC_IF == jump_PC_ID : // jump right
3      ~jump_right; // jump when no branch

```

具体分析：

##### (a) jump\_right\_reg 分支判断：

这里，jump\_right\_reg 为上一个周期的 jump\_right 信号。

若不考虑 jump\_right\_reg，直接使用 jump\_right 信号进行判断：

```

1      assign reg_FD_flush_of_predict = ~jump_right;

```

Name	Value	Waveform
clk	1	218.000 ns 220.000 ns 222.000 ns 224.000 ns 226.000 ns 228.000 ns 230.000 ns 232.000 ns
rst	0	
PC_IF[31:0]	00000330	0000... 00000340 00000344 00000348 00000330 00000334 00000338 0000033c
Inst_IF[31:0]	00c5e663	0016... 00165613 fe0696e3 00008067 00c5e663 40c585b3 00d56533 0016d6b3
PC_predict[31:0]	0000033c	00000000 00000330 0000029c 0000033c 00000000
jump_PC_ID[31:0]	00000344	0000... 0000033d 00000341 00000330 00000344 00000334 00000330
final_PC_IF[31:0]	00000334	0000... 00000344 00000348 0000029c 00000334 00000338 0000033c 00000330
next_PC_IF[31:0]	00000334	0000... 00000344 00000348 00000330 00000334 00000338 0000033c 00000330
PC_ID[31:0]	00000344	0000... 0000033c 00000340 00000344 00000334 00000330
Inst_ID[31:0]	00000013	00d5... 0016d693 00165613 fe0696e3 00000013 40c585b3 00d56533
Branch_ctrl	0	
JALR	0	
Branch_predict	0	
last_Branch_predict	1	
btb_hit	1	
last_btb_hit	1	
bht_wen	0	
btb_wen	0	
predict_right	0	
jump_right	0	
jump_right_reg	0	
reg_FD_flush_of_predict	1	
btb_fetch_pc[0:255][31:0]	0000000c, 00000218, 00000158, 0000010c, 00000260, 000002f8, 00000298, 00000344, 00000330, 00000208, 0000013c, 000000e4, 00000240, 000002f4, 00000288, 000002fc, 00000330, 00000334, 00000338, 0000033c, 00000340, 00000344, 00000348, 0000034c, 00000350, 00000354, 00000358, 0000035c, 00000360, 00000364, 00000368, 0000036c, 00000370, 00000374, 00000378, 0000037c, 00000380, 00000384, 00000388, 0000038c, 00000390, 00000394, 00000398, 0000039c, 000003a0, 000003a4, 000003a8, 000003ac, 000003b0, 000003b4, 000003b8, 000003bc, 000003c0, 000003c4, 000003c8, 000003cc, 000003d0, 000003d4, 000003d8, 000003dc, 000003e0, 000003e4, 000003e8, 000003ec, 000003f0, 000003f4, 000003f8, 000003fc, 00000400, 00000404, 00000408, 0000040c, 00000410, 00000414, 00000418, 0000041c, 00000420, 00000424, 00000428, 0000042c, 00000430, 00000434, 00000438, 0000043c, 00000440, 00000444, 00000448, 0000044c, 00000450, 00000454, 00000458, 0000045c, 00000460, 00000464, 00000468, 0000046c, 00000470, 00000474, 00000478, 0000047c, 00000480, 00000484, 00000488, 0000048c, 00000490, 00000494, 00000498, 0000049c, 000004a0, 000004a4, 000004a8, 000004ac, 000004b0, 000004b4, 000004b8, 000004bc, 000004c0, 000004c4, 000004c8, 000004cc, 000004d0, 000004d4, 000004d8, 000004dc, 000004e0, 000004e4, 000004e8, 000004ec, 000004f0, 000004f4, 000004f8, 000004fc, 00000500, 00000504, 00000508, 0000050c, 00000510, 00000514, 00000518, 0000051c, 00000520, 00000524, 00000528, 0000052c, 00000530, 00000534, 00000538, 0000053c, 00000540, 00000544, 00000548, 0000054c, 00000550, 00000554, 00000558, 0000055c, 00000560, 00000564, 00000568, 0000056c, 00000570, 00000574, 00000578, 0000057c, 00000580, 00000584, 00000588, 0000058c, 00000590, 00000594, 00000598, 0000059c, 000005a0, 000005a4, 000005a8, 000005ac, 000005b0, 000005b4, 000005b8, 000005bc, 000005c0, 000005c4, 000005c8, 000005cc, 000005d0, 000005d4, 000005d8, 000005dc, 000005e0, 000005e4, 000005e8, 000005ec, 000005f0, 000005f4, 000005f8, 000005fc, 00000600, 00000604, 00000608, 0000060c, 00000610, 00000614, 00000618, 0000061c, 00000620, 00000624, 00000628, 0000062c, 00000630, 00000634, 00000638, 0000063c, 00000640, 00000644, 00000648, 0000064c, 00000650, 00000654, 00000658, 0000065c, 00000660, 00000664, 00000668, 0000066c, 00000670, 00000674, 00000678, 0000067c, 00000680, 00000684, 00000688, 0000068c, 00000690, 00000694, 00000698, 0000069c, 000006a0, 000006a4, 000006a8, 000006ac, 000006b0, 000006b4, 000006b8, 000006bc, 000006c0, 000006c4, 000006c8, 000006cc, 000006d0, 000006d4, 000006d8, 000006dc, 000006e0, 000006e4, 000006e8, 000006ec, 000006f0, 000006f4, 000006f8, 000006fc, 00000700, 00000704, 00000708, 0000070c, 00000710, 00000714, 00000718, 0000071c, 00000720, 00000724, 00000728, 0000072c, 00000730, 00000734, 00000738, 0000073c, 00000740, 00000744, 00000748, 0000074c, 00000750, 00000754, 00000758, 0000075c, 00000760, 00000764, 00000768, 0000076c,	

图 1: 不考虑 jump\_right\_reg 的 flush 控制

此时，由于前一个周期预测跳转错误(!jump\_right\_reg)，next\_PC\_IF 仍然需要进行跳转为 jump\_PC\_ID；

尽管本周期的预测跳转同样错误(!jump\_right)，但错误的预测值 final\_PC\_IF 并没有被写入到 PC\_IF 中；(见1.3 流水线的修改：2 next\_PC\_IF的生成)

因而此时不需要对 IF\_ID 寄存器进行 flush 操作。而实际进行了 flush 操作，导致了本应该执行的指令(PC = 0x330)被丢弃。

更正后，结果如下：

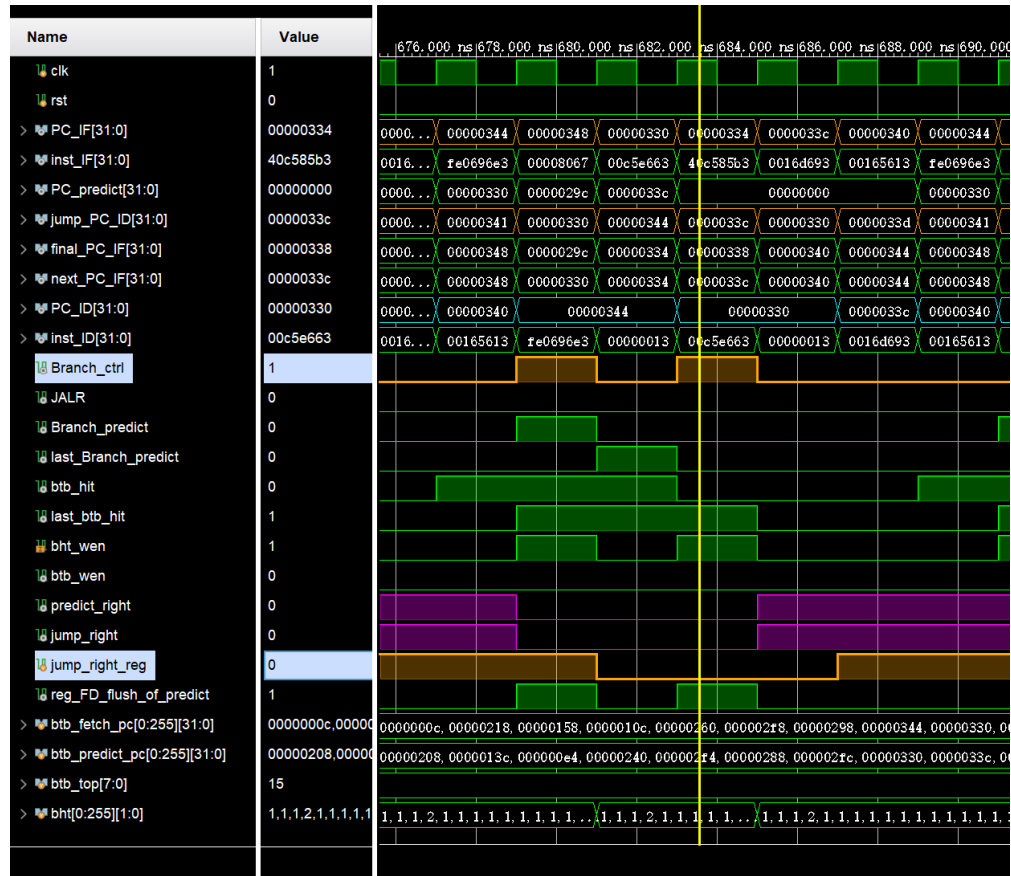


图 2: jump\_right\_reg 问题更正

可见问题已经解决。

(b) !jump\_right\_reg 分支下的 Branch\_ctrl 判断：

在考虑 jump\_right\_reg 为 0 的情况下，还需要考虑 Branch\_ctrl 的值。

若不考虑 Branch\_ctrl 的值：

```
1 assign reg_FD_flush_of_predict = ~jump_right_reg ? 1'b0 : // jump_right_reg fix
2                               ~jump_right;
```



则会出现新的问题:

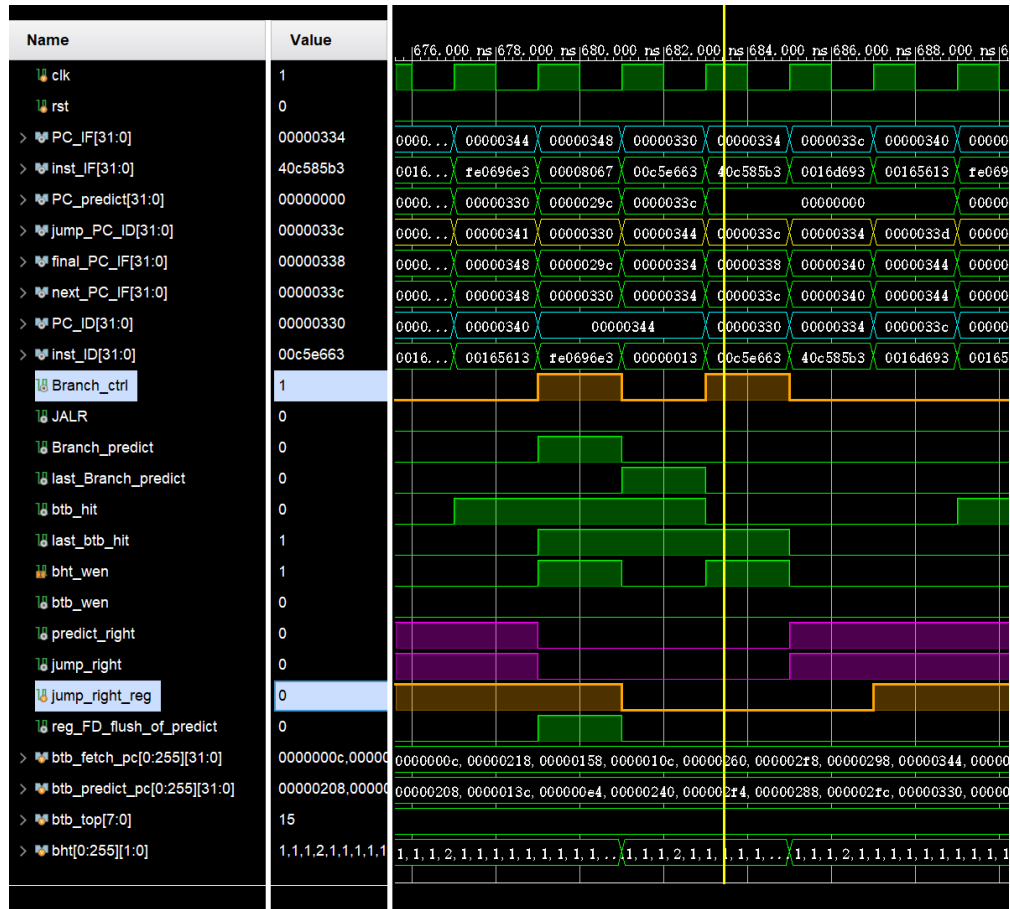


图 3: jump\_right\_reg 基础上不考虑 Branch\_ctrl 的 flush 控制

如图 3 所示, 当 PC\_IF 为 0x334 时, jump\_right\_reg 信号为 0, 然而此时 Branch\_ctrl 信号为 1, 且 jump\_right 信号为 0, 意味着仍需暂停一拍, 将错误的 PC\_IF 冲刷掉, 这与图中不相符。

因而，当 `jump_right_reg` 信号为 0，且 `Branch_ctrl` 信号为 1 时，还需要对 `flush` 策略进行修正，即：

```
1 assign reg_FD_flush_of_predict = ~jump_right_reg ? Branch_ctrl ? ~jump_right :
2                                     1'b0;
3                                     ~jump_right;
```

[illegible]

可见问题已经解决。

上述对于 `jump_right_reg` 的问题的处理，是对 `reg_FD_flush_of_predict` 信号的修正，然而该问题实际上还会影响 `jump_right` 等信号的值，进而传递到下一个周期再度引发问题：

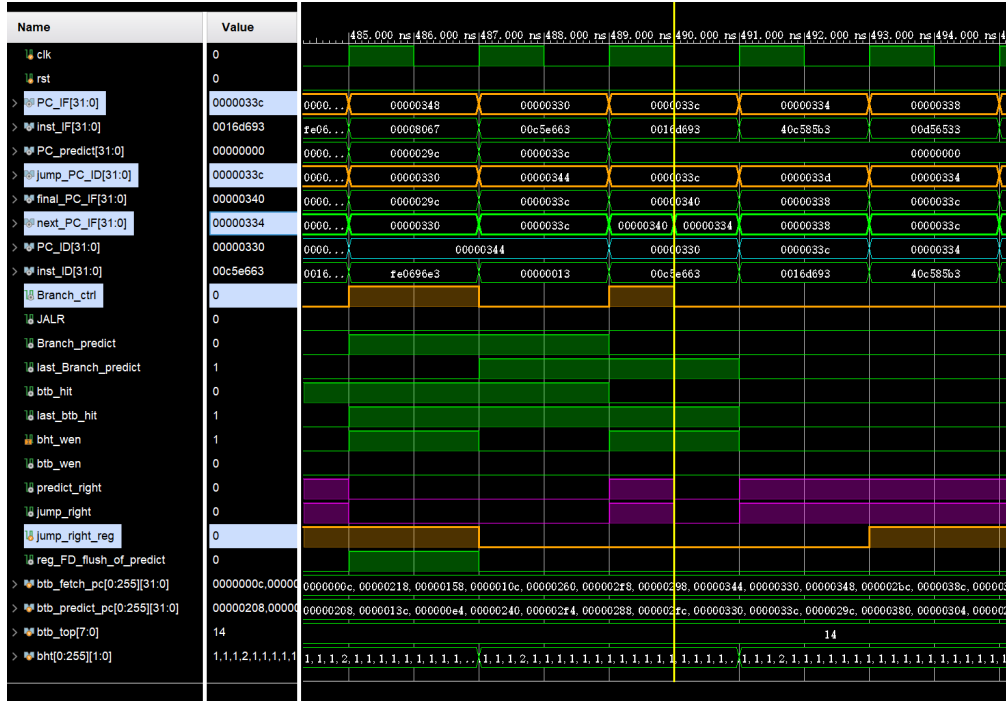


图 5: 问题的延续

(注：由于实现中 Regs 模块采用的是下降沿写入，因而下降沿之后各信号的值才是正确值。)

如图所示，当 PC\_IF 为 0x33c 时，下降沿之后 jump\_right\_reg 信号为 0，且 Branch\_ctrl 信号为 0，根据之前的修正，reg\_FD\_flush\_of\_predict 信号应当为 0；

然而实际上，PC\_IF = 0x33c 是前一个周期给出的错误预测值，应当被冲刷掉。

这个问题的产生正是因为，前一个周期的 jump\_right\_reg 信号也为 0，虽然该周期对 reg\_FD\_flush\_of\_predict 信号进行了正确修正，但该周期的 jump\_right 信号其实是错误的，且被传递到了下一个周期。

因此，仍需要添加补丁，如下：

```
1 assign reg_FD_flush_of_predict = ~jump_right_reg ? Branch_ctrl ? ~jump_right :
2                               PC_IF == jump_PC_ID : // jump right
3                               ~jump_right; // jump when no branch
```

修正后，结果如下：

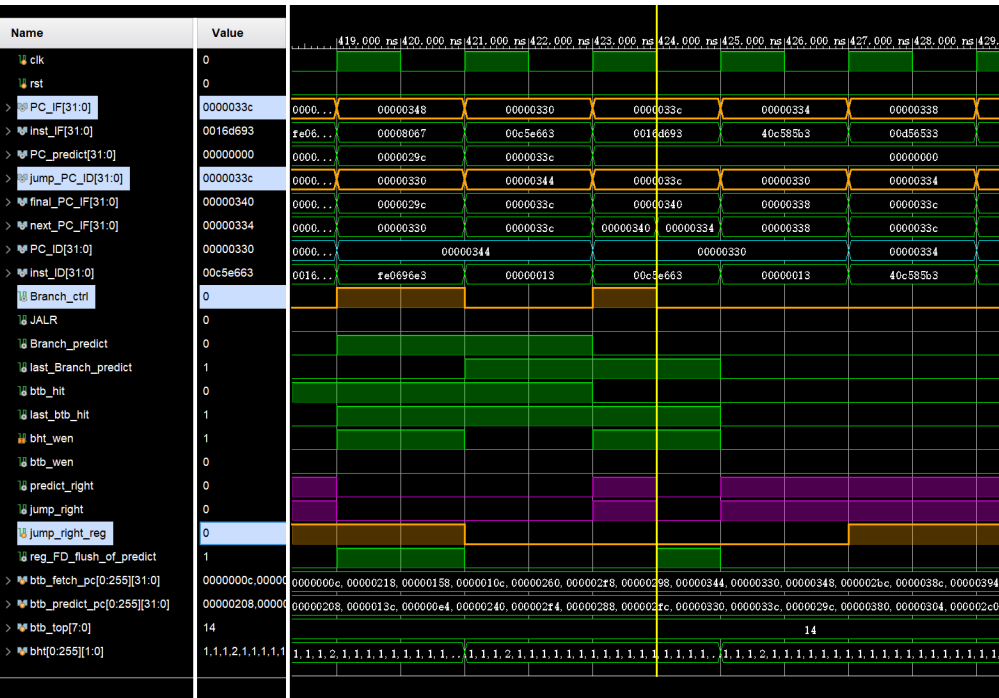


图 6: 问题的延续修正

可见问题已经解决。

## 2 仿真结果与上板验证

1. 仿真结果如下：

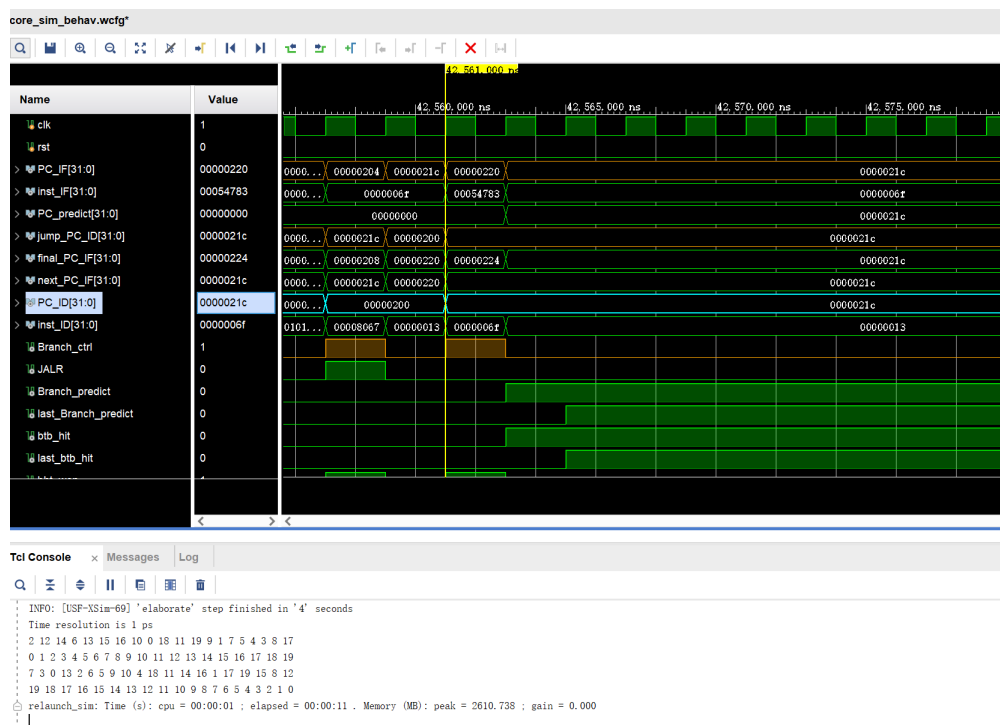


图 7: 仿真结果

可见，仿真正确输入了预期值，且仿真进入循环的时间为 42561 ns，达到了实验要求的阈值。实验设计符合预期。

2. 上板验证结果如下：

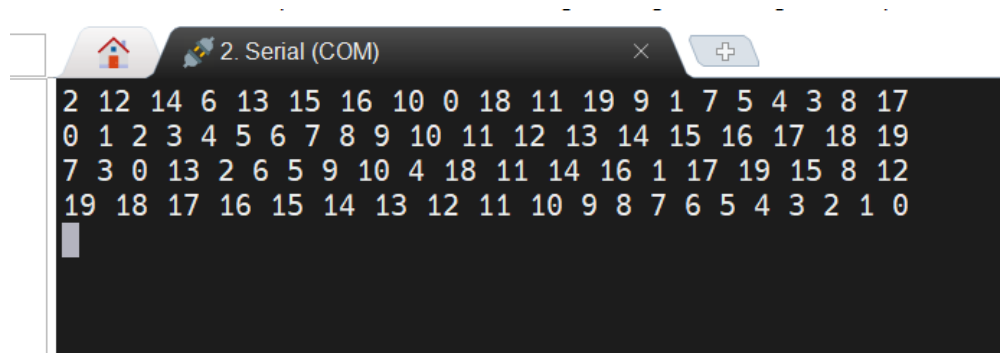


图 8: 上板验证结果

可见，上板验证时串口输出值正确，实验设计符合预期。

### 3 思考题 1

1. 加了分支预测后，仿真跑测试程序，较没加的时候快了多少？以仿真中进入 loop 的时间作为程序运行时间，计算加速比。

未添加分支预测时，仿真结果如下：

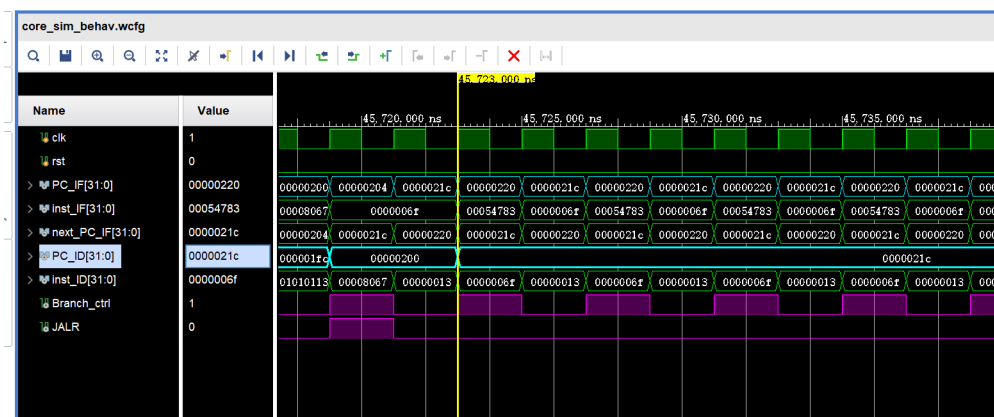


图 9: 未添加分支预测时的仿真结果

可见此时进入循环的仿真时间为 45732 ns。

与我添加预测后的仿真时间 42561 ns 相比，速度提升了 6.93%。

2. 请在报告里展示以下五种仿真波形：

- 分支预测跳转，实际不跳转，分支预测不跳转，实际跳转

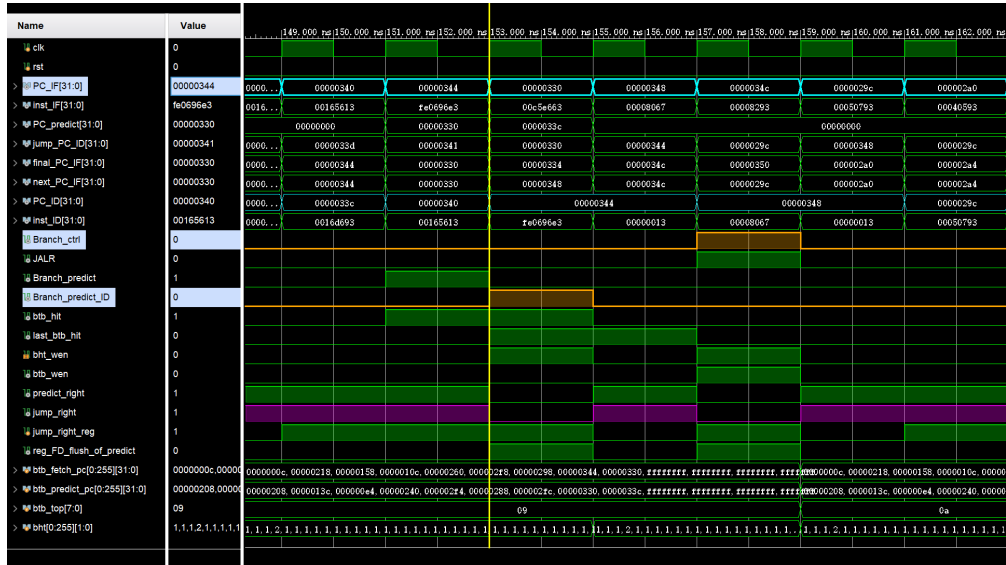


图 10: 分支预测跳转，实际不跳转以及分支预测不跳转，实际跳转

如图，PC\_IF = 0x34c 时，即为分支预测跳转，实际不跳转的情形。

而 PC\_IF = 0x330 时，即为分支预测不跳转，实际跳转的情形。







