# 实验 3 - 动态分支预测

## 1 实验步骤

### 1.1 分支预测模块

模块定义如下:

```
module BranchPrediction(
2
        input clk,
3
        input rst,
4
       // IF
5
        input [7:0]PC_index_IF,
       // ID
6
7
        input [7:0]PC_index_ID,
8
       // input [31:0]jump_PC_ID,
        input [7:0]jump_PC_ID,
9
        input is_branch,
                                      // 是否是跳转指令
10
                                       // 比较器的结果+是否是 branch 指令
11
        input branch,
12
13
        output [7:0]PC_predict_IF,
14
        output predict_branch_IF,
        output refetch
15
16
        );
17
        localparam SIZE = 256;
        reg [1:0] BHT[0:SIZE-1];
18
19
        reg [7:0] BTB[0:SIZE-1];
```

分支预测模块中,主要实现的是 BHT 和 BTB 的逻辑,为了方便书写我们分开实现。

BHT

BHT 主要的逻辑就是一个有限状态机,根据当前状态和输入的 branch 信号,决定下一个状态。

```
1
   // 处理 BHT
 2
    always @(posedge clk or posedge rst) begin
 3
        if (rst) begin
 4
            for (i = 0; i < SIZE; i = i + 1) begin
 5
                 BHT[i] <= 2'b00;
 6
            end
 7
        end
 8
        else if (is_branch) begin
 9
            case(BHT[PC_index_ID])
10
                 2'b00 : begin
                     if (branch) BHT[PC_index_ID] <= 2'b01;</pre>
11
```

```
12
                       else BHT[PC_index_ID] <= 2'b00;</pre>
13
                  end
14
                  2'b01 : begin
                       if (branch) BHT[PC_index_ID] <= 2'b11;</pre>
15
                       else BHT[PC_index_ID] <= 2'b00;</pre>
16
17
                  end
                  2'b11 : begin
18
19
                       if (branch) BHT[PC_index_ID] <= 2'b11;</pre>
                       else BHT[PC_index_ID] <= 2'b10;</pre>
20
21
                  end
22
                  2'b10 : begin
23
                       if (branch) BHT[PC_index_ID] <= 2'b11;</pre>
24
                       else BHT[PC_index_ID] <= 2'b00;</pre>
25
                  end
26
             endcase
27
         end
28 end
```

#### BTB

BTB 的逻辑是在跳转发生,同时 BTB 里没有存储目标地址或者目标地址过时的时候,将目标地址存入 BTB。但是为了方便起见,只要跳转发生,我们就将目标地址写入 BTB,实时更新。这样不会影响程序的正确性,相反还能减少判断,进而降低门延迟。

```
1
   // 处理 BTB
 2
    always @(posedge clk or posedge rst) begin
 3
        if (rst) begin
 4
            for (i = 0; i < SIZE; i = i + 1) begin
 5
                 BTB[i] <= 8'd0;
 6
            end
 7
        end
 8
        else begin
 9
10
            // 修改 BTB
11
            if (branch) begin
12
                 BTB[PC_index_ID] <= jump_PC_ID;</pre>
13
            end
14
        end
15
    end
```

处理完 BTB 和 BHT 的更新后,我们要根据这两个表来预测跳转。 predict\_branch\_IF 表示我们预测是否跳转, PC\_predict\_IF 表示预测的跳转地址, refetch 表示是否需要重新取指令。

```
assign refetch = is_branch && ((BHT[PC_index_ID][1] != branch) || (branch &&
BTB[PC_index_ID] != jump_PC_ID));
assign predict_branch_IF = BHT[PC_index_IF][1];
assign PC_predict_IF = BTB[PC_index_IF];
```

这里 predict\_branch\_IF 直接由 BHT 的第二位决定(即状态为 11 或者 00 的时候表示预测跳转),PC\_predict\_IF 则由 BTB 决定。需要注意的是,我们更新 BHT 和 BTB 要根据跳转结果,因此是在 ID 阶段更新的,使用的也是 PC\_index\_ID 。但是预测跳转的时候,我们是根据最新的 PC 来预测,因此是在 IF 阶段预测的,使用的是 PC\_index\_IF 。

refetch 发生在以下几种情况:

- 预测跳转,但是实际上不发生跳转。
- 预测不跳转,但是实际上发生跳转。
- 预测跳转,实际上也跳转,但是跳转的目标地址和预测的不一样。d

这几种情况 refetch 为 1,此时跳转指令在 ID 阶段, IF 阶段对应的指令是错误的,需要 flush 掉并重新取指令。

### 1.2 RV32Core 模块修改

实例化 BranchPrediction 模块,并且将其输出接入到 RV32Core 模块中。

```
1
        BranchPrediction branch_prediction(
2
            .clk(debug_clk),
 3
            .rst(rst),
4
5
            .PC_index_IF(PC_IF[9:2]),
            .PC_index_ID(PC_ID[9:2]),
6
7
            .predict_branch_IF(taken),
8
            .PC_predict_IF(pc_to_take),
9
10
            .is_branch(j),
11
            .branch(Branch_ctrl),
12
            .jump_PC_ID(jump_PC_ID[9:2]),
            .refetch(refetch)
13
14
           );
```

这里的 j 是从 CtrlUnit 接出的信号,用来表示当前 ID 阶段是否是跳转指令。在 CtrlUnit 中我们添加 assign branch\_type = JAL | JALR | B\_valid; 来判断是否是跳转指令。

对于 refetch 的情况, 我们修改 IF 阶段 PC 取指的逻辑如下:

- 如果 refetch=0 ,则我们选择 pc\_IF ,具体为 IF 的下一条指令(不跳转)或者预测的跳转地址(跳转),根据预测的结果 taken 决定。
- 如果 refetch=1, 说明我们需要重新取指,则我们选择 pc\_ID, 具体为 ID 的下一条指令(不跳转)或者 跳转指令的目标地址(跳转),根据实际跳转结果 Branch\_ctrl 决定。

除此之外, REG\_IF\_ID 中的 flush 信号也要改为 refetch, 即重新取指就要把当前 IF 的指令 flush 掉。

# 2 实验评估

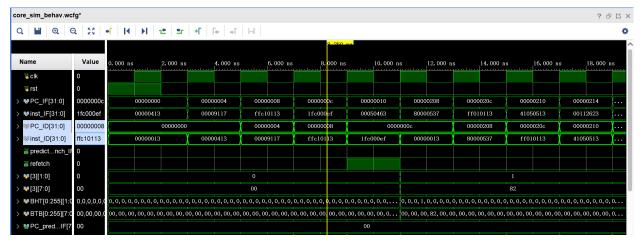
### 2.1 仿真

仿真可以正确运行所给的程序,完整的波形包括串口输出如下: Exp2 - [C:/Users/86139/Desktop/23FALL/arch-fa23/src/lab3/Exp3\_NEXYS\_A7/Exp2.xpr] - Vivado 2020.2 File Edit Flow Tools Reports Window Layout View Run Help Q-Quick Access Synthesis and Implementation Out-of-date details  $\blacksquare$   $\blacksquare$   $\times$   $\blacktriangleright$   $\blacksquare$  Φ Σ  $\varnothing$   $\varnothing$  ⊗ ⊗  $\blacksquare$   $\blacksquare$   $\blacksquare$  50⊗ us ∨ <u>₹</u> || C ■ Default Layout Flow Navigator 😤 🗧 SIMULATION - Behavioral Simulation - Functional - sim\_1 - core\_sim ∨ PROJECT MANAGER core\_sim\_behav.wcfg ? @ U × Settings Q | @ | Q | 23 | # | H | M | 12 | 21 | # | F | # | H Add Sources Language Templates Name Value P IP Catalog ✓ IP INTEGRATOR ₩PC\_IF[31:0] 00000210 **⊌** inst\_IF[31:0] predict...nch
 \_ Open Block Design ₩ refetch WBHT[0:255][1:0 0,0,0,1,0, 0, 0, 0, 1, 0, 3, 0, 0, 0, 0, 0, 0 ₩ BTB[0:255][7:0 00.00.00. 00, 00, 00, 82, 00, 64, 00, 00, SIMULATION PC\_pred...IF[7 87 Run Simulation ₩PC ID[31:0] 00000210 0000021 **⊌** inst\_ID[31:0] 00000066 00000061 ✓ RTL ANALYSIS ₩PC\_EXE[31:0] 00000210 0000021 0000006f > Open Elaborated Design ✓ SYNTHESIS Run Synthesis Tcl Console × Messages Log ? \_ 0 6 > Open Synthesized Design Q 🛨 🖨 II 🖫 🎟 🗰 launch\_simulation: Time (s): cpu = 00:00:11 ; elapsed = 00:00:12 . Memory (MB): peak = 1188.320 ; gain = 1.953 Run Implementation 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 7 3 0 13 2 6 5 9 10 4 18 11 14 16 1 17 19 15 8 12 > Open Implemented Design 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 ▼ PROGRAM AND DEBUG ♣ Generate Bitstream

#### 1. 预测不跳转失败

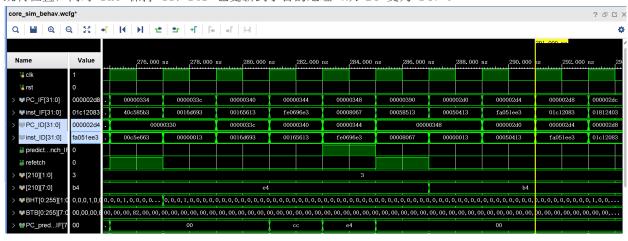
在 7ns, 0xC 对应指令 jal ra,80000208 ,由于是第一次运行,BHT 为 00,所以预测不跳转,但实际上进行了跳转,所以在 0xC 进入 ID 阶段时 refetch 信号变为了 1,因此下一周期 PC\_IF 重新取指得到了正确的跳转位置,同时 BHT 更新为了 01,BTB 也更新到了目的地址。整个波形和前面预先设计的时序图是一致的。

Sim Time: 51 us



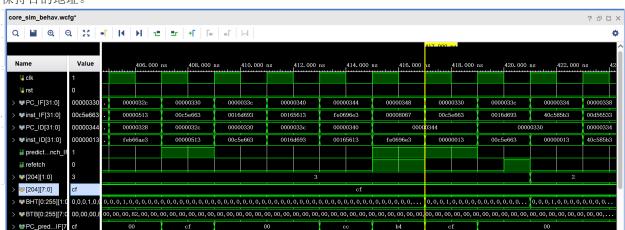
#### 2. 预测跳转成功,但跳转地址不同

在 283ns, 0x348 对应指令 **jalr zero,0(ra)**, 这里 BHT 为 11, 所以预测跳转。但是因为 **ra** 值不同, 所以实际跳转的地址也不同, 因此 **refetch** 信号变为了 1, 下一周期 **PC\_IF** 重新取指得到了正确的跳转位置, 同时 BHT 保持 11, BTB 也更新到了目的地址(从 E4 变为 B4)。



#### 3. 预测跳转成功

在 407ns, 0x330 对应指令 bltu a1,a2,8000033c , 这里 BHT 为 11, 所以预测跳转。实际上也跳转,而且跳转地址相同, 因此 refetch 信号为 0, 下一周期 PC\_IF 无缝取指, 同时 BHT 保持 11, BTB 也保持目的地址。



4. 预测不跳转失败之后预测跳转失败

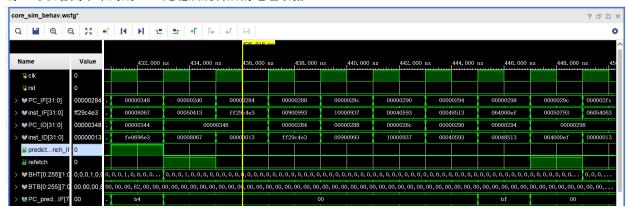
图同上, 在 413ns 时, 0x344 对应指令 bne a3,zero,80000330 , 这里预测不跳转

( predict\_branch\_IF 为 0),但实际上发生了跳转,因此在 0x344 进入 ID 阶段时 refetch 信号变为了 1,下一周期 PC\_IF 重新取指得到了正确的跳转位置。

417 ns 时,0x330 对应跳转指令,这里预测跳转(  $predict\_branch\_IF$  为 1),实际上不跳转,因此预测失败。在 0x330 进入 ID 阶段时 refetch 信号变为了 1,下一周期  $PC\_IF$  重新取指得到了正确的跳转位置。

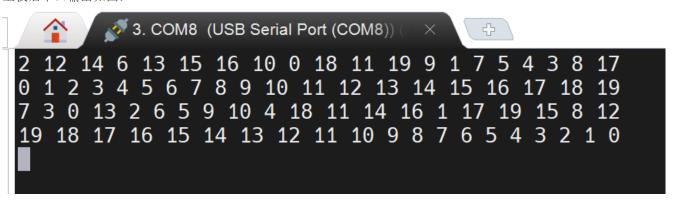
#### 5. 预测不跳转成功

435 ns 时,0x284 对应指令 **blt s3,s2,8000026c**,这里预测不跳转,实际上也不跳转,因此预测成功。可以看到下个周期 PC 无缝从跳转后的地址取指。



### 2.2 上板结果

上板后串口输出如图:



# 3 思考题

- 1. 加了分支预测后,跑测试程序,较没加的时候快了多少? 仿真下,不加分支预测(默认 predict not taken)需要跑满 45 微秒,而加了分支预测后只需要 42 微妙 左右就能结束,因此快了大概 7%。
- 2. 在正确实现 BTB 和 BHT 的情况下,有没有可能会出现 BHT 预测分支发生跳转,也就是 branch taken,但是 BTB 中杳不到目标跳转地址,为什么?

不可能,当 BHT 预测分支发生跳转时,说明 BTB 中的状态为 10 或者 11,而我们的初始状态为 00。只有当发生了跳转之后,BHT 才会预测跳转,这个时候 BTB 已经更新了,因此不可能出现 BHT 预测分支发生跳转,但是 BTB 中查不到目标跳转地址的情况。

3. 前面介绍的 BHT 和 BTB 都是基于内容检索,即通过将当前 PC 和表中存储的 PC 比较来确定分支信息存储于哪一表项。这种设计很像一个全相联的 cache,硬件逻辑实际上会比较复杂,那么能否参考直接映射或组相联的 cache 来简化 BHT/BTB 的存储和检索逻辑?请简述你的思路。

实际上我认为本次实验设计的 BTB 和 BHT 也可以认为是一个直接映射的 cache (即直接根据 PC 地址找到 对应的项)

组相联的方式我认为可以,我们将每次的 PC 地址分成 index 和 tag bits,随后直接参考 cache 的设计即可。访问 index,就从对应的组里比较 tag,如果有 tag 相同而且有效的项就找到了对应的值,也可以根据实际跳转结果更新跳转地址。但是我认为这样硬件的逻辑会更加复杂。