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(
    input clk,
    input rst,
    // IF
    input [7:0]PC_index_IF,
    // ID
    input [7:0]PC_index_ID,
    // input [31:0]jump_PC_ID,
    input [7:0]jump_PC_ID,
    input is_branch,                // 是否是跳转指令
    input branch,                   // 比较器的结果+是否是 branch 指令

    output [7:0]PC_predict_IF,
    output predict_branch_IF,
    output refetch
    );
    localparam SIZE = 256;
    reg [1:0] BHT[0:SIZE-1];
    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。

  ```
  // 处理 BHT
  always @(posedge clk or posedge rst) begin
      if (rst) begin
          for (i = 0; i < SIZE; i = i + 1) begin
              BHT[i] <= 2'b00;
          end
      end
      else if (is_branch) begin
          case(BHT[PC_index_ID])
              2'b00 : begin
                  if (branch) BHT[PC_index_ID] <= 2'b01;
  ```

```
                else BHT[PC_index_ID] <= 2'b00;
            end
            2'b01 : begin
                if (branch) BHT[PC_index_ID] <= 2'b11;
                else BHT[PC_index_ID] <= 2'b00;
            end
            2'b11 : begin
                if (branch) BHT[PC_index_ID] <= 2'b11;
                else BHT[PC_index_ID] <= 2'b10;
            end
            2'b10 : begin
                if (branch) BHT[PC_index_ID] <= 2'b11;
                else BHT[PC_index_ID] <= 2'b00;
            end
        endcase
    end
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。

```
// 处理 BTB
always @(posedge clk or posedge rst) begin
    if (rst) begin
        for (i = 0; i < SIZE; i = i + 1) begin
            BTB[i] <= 8'd0;
        end
    end
    else begin

        // 修改 BTB
        if (branch) begin
            BTB[PC_index_ID] <= jump_PC_ID;
        end
    end
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。

```
    BranchPrediction branch_prediction(
        .clk(debug_clk),
        .rst(rst),

        .PC_index_IF(PC_IF[9:2]),
        .PC_index_ID(PC_ID[9:2]),
        .predict_branch_IF(taken),
        .PC_predict_IF(pc_to_take),

        .is_branch(j),
        .branch(Branch_ctrl),
        .jump_PC_ID(jump_PC_ID[9:2]),
        .refetch(refetch)
       );
```

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：

```
    MUX2T1_32 mux_IF(.I0(pc_IF),.I1(pc_ID),.s(refetch),.o(next_PC_IF));
    MUX2T1_32 mux_IF_normal(.I0(PC_ID +
4),.I1(jump_PC_ID),.s(Branch_ctrl),.o(pc_ID));
    MUX2T1_32
mux_IF_predict(.I0(PC_4_IF),.I1({22'b0,pc_to_take,2'b0}),.s(taken),.o(pc_IF));
```

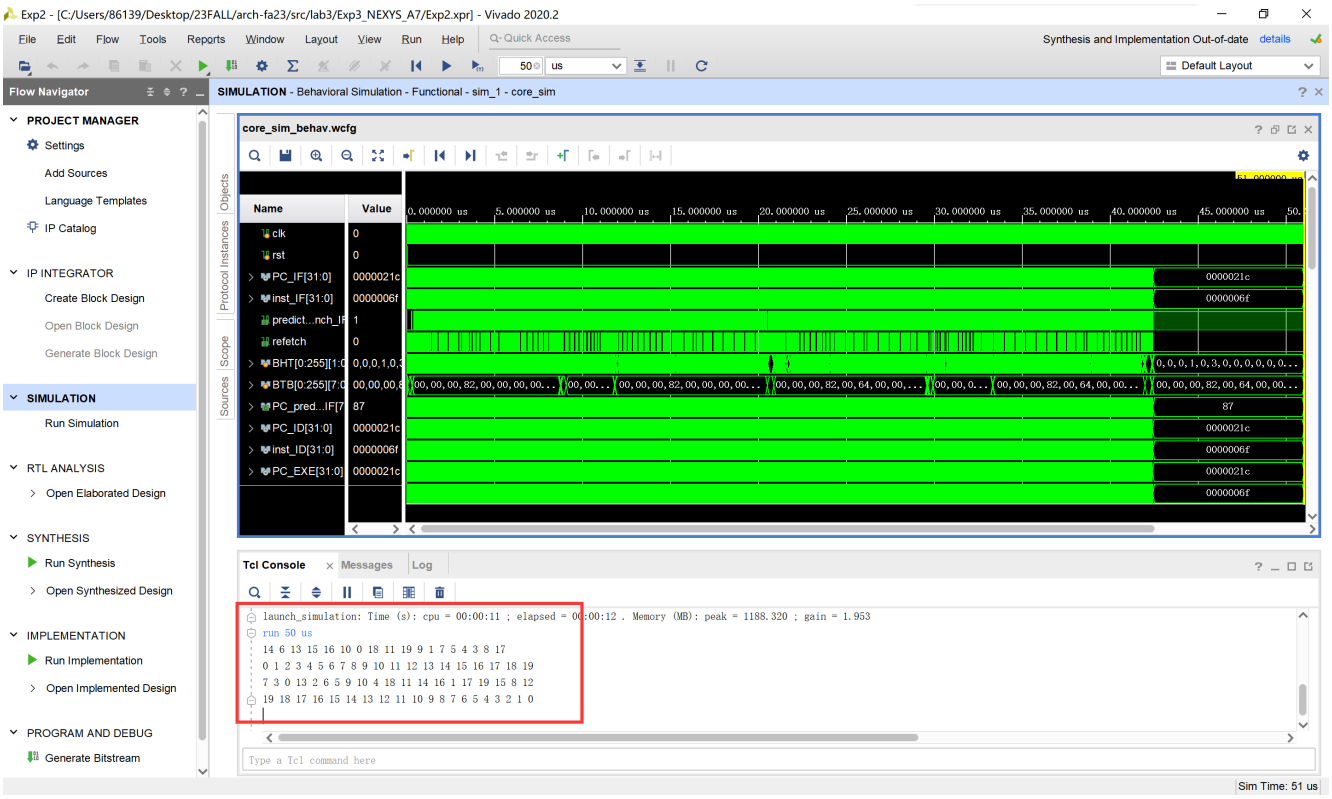
- 如果 `refetch=0`，则我们选择 `pc_IF`，具体为 IF 的下一条指令（不跳转）或者预测的跳转地址（跳转），根据预测的结果 `taken` 决定。
- 如果 `refetch=1`，说明我们需要重新取指，则我们选择 `pc_ID`，具体为 ID 的下一条指令（不跳转）或者跳转指令的目标地址（跳转），根据实际跳转结果 `Branch_ctrl` 决定。

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

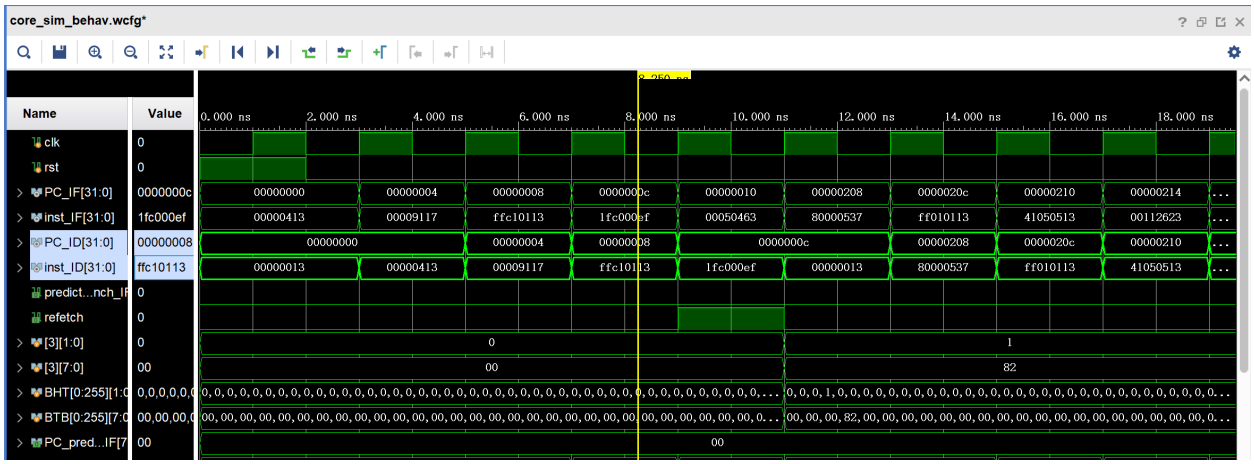
2.1 仿真

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：



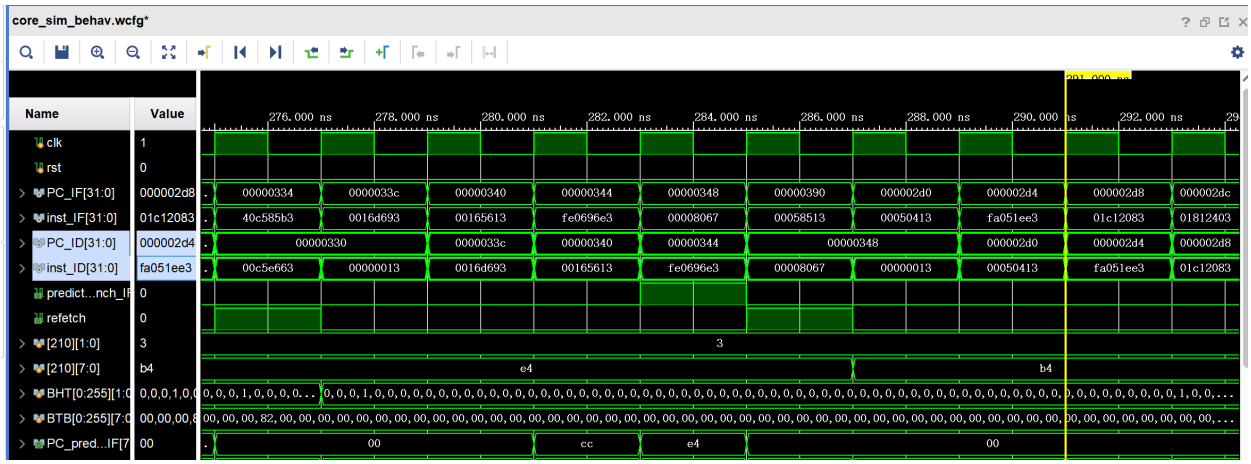
1. 预测不跳转失败

   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。



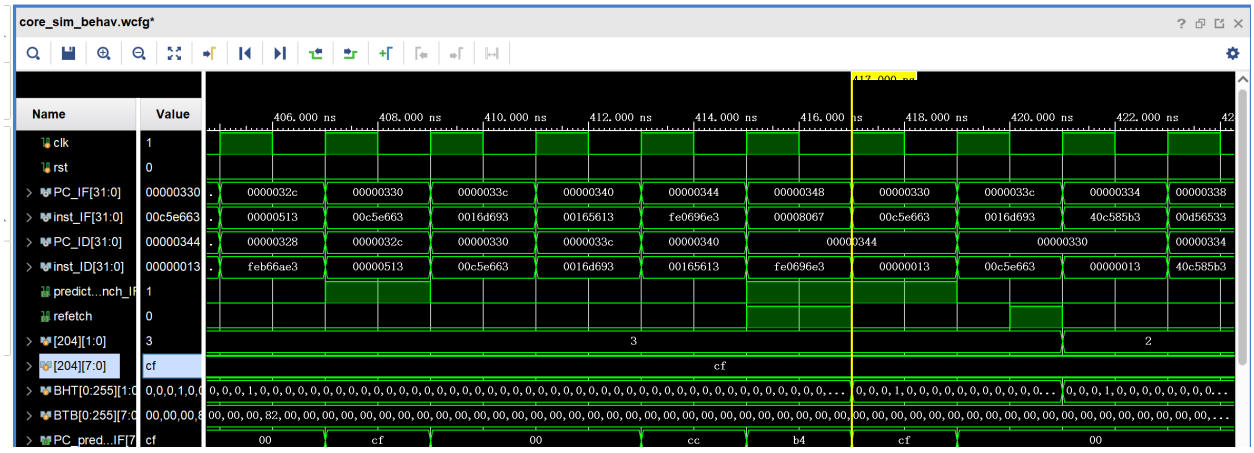
2. 预测跳转成功，但跳转地址不同

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。



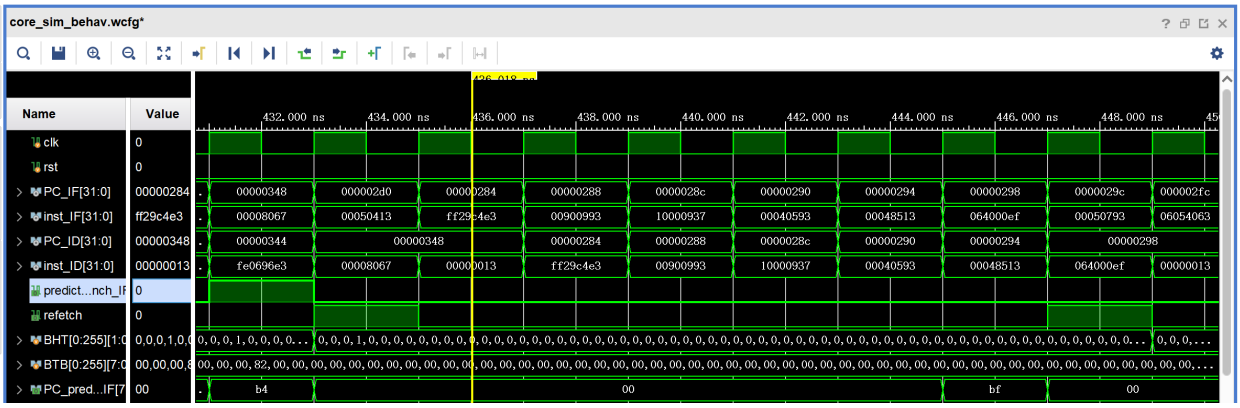
4. 预测不跳转失败之后预测跳转失败

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。

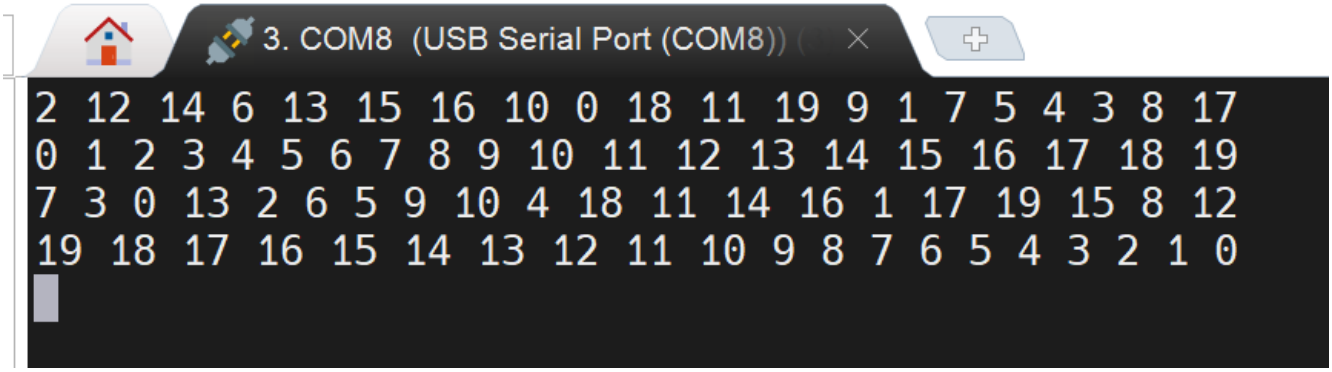
5. 预测不跳转成功

   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 中查不到目标跳转地址，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。