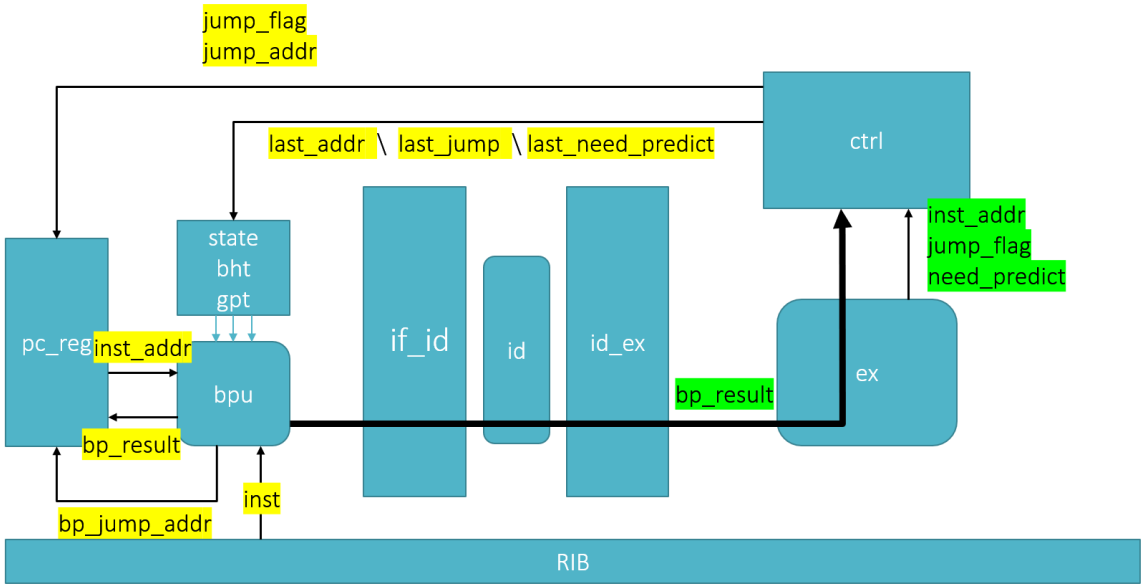
实验报告：Lab2(分支预测器)

思路&实现



上图展示了我在原有tinyriscv处理器核结构的基础上做的优化工作，与分支逻辑预测相关的工作可以概括为三部分：预测、修正、状态更新；

预测

发生在bpu模块与pc_reg模块直接，逻辑是pc寄存器向bpu提供取指的地址和指令（指令由RIB总线直接提供），bpu判断当前指令是否为B型指令或JAL指令，如果是，则根据相应的机器码解码规则获取分支跳转的地址（注意，这一步是PC相对寻址，且两类指令的结构不同），并连接到输出的连线bp_jump_addr上，在下一时钟上升沿参与到pc_reg的更新中；同时，bpu通过读取state\bht\gpt等寄存器的状态或者跳转地址来确定该次分支预测是否生效（及是否影响pc_reg下一周期的值），结果连接到输出连线bp_result。

代码主要由模块间连线和模块内更新两部分构成，这里以静态分支预测的代码为例：

pc_reg&bpu连线：

```verilog
pc_reg u_pc_reg(
    ...
    .bp_result_i(bpu_bp_result_o),
    .bp_jump_addr_i(bpu_bp_jump_addr_o),
    .pc_o(pc_pc_o),
    ...
);

bpu u_bpu(
    ...
    .inst_i(rib_pc_data_i),
    .inst_addr_i(pc_pc_o),
    .bp_result_o(bpu_bp_result_o),
    .bp_jump_addr_o(bpu_bp_jump_addr_o),
    ...
)
```

bpu内部逻辑

```verilog
// bpu.v
wire[6:0] opcode;
wire[`InstAddrBus] jal_addr;
wire[`InstAddrBus] b_addr;

assign opcode = inst_i[6:0];
/* 两类指令的计算跳转地址的方式不同！！！*/
assign jal_addr =  {{12{inst_i[31]}}, inst_i[19:12], inst_i[20], inst_i[30:21], 1'b0} + inst_addr_i;
assign b_addr = {{20{inst_i[31]}}, inst_i[7], inst_i[30:25], inst_i[11:8], 1'b0} + inst_addr_i;

always @ (*) begin
    bp_result_o = `JumpDisable;
    bp_jump_addr_o = `ZeroWord;

    case (opcode)
      `INST_JAL: begin
        bp_jump_addr_o = jal_addr;
        bp_result_o = 1'b1;                                 // static predictor
      end
      `INST_TYPE_B:begin
        bp_jump_addr_o = b_addr;
        bp_result_o = $signed(b_addr) < $signed(inst_addr_i);   // static predictor
      end
      default: begin
      end

    endcase
end
```

修正

针对每次造成跳转的指令（JAL \ B类指令），需要在ex阶段对分支预测的结果进行检验，如果分支预测的结果是错误的，那么需要通过ctrl模块修改pc寄存器完成正确的跳转，并冲洗流水线（消除错误预测的影响），这一步的实现需要将bpu的预测结果bp_result沿流水线的节拍一级一级地向后传，传入ctrl模块，同时ex模块（组合逻辑）根据id_ex寄存器中地数据对当前跳转指令是否跳转进行判断，将当前指令是否发生分支预测need_predict和当前指令是否跳转jump_flag以及当前指令的地址传入ctrl模块。 在ctrl模块中，由jump_flag和bp_result的异或值确定是否需要修正跳转并冲洗流水线，并设置正确跳转的地址（如果不该跳却跳了，设为inst_addr+4，否则设置为jump_addr).

代码修改主要包括ctrl模块的内部修改和对pc_reg\ctrl模块连线情况的修改

ctrl.v内部逻辑

```verilog
wire [`InstAddrBus] addr;
wire jump;

assign jump = jump_flag_i ^ bp_result_i;
assign addr = (bp_result_i) ? (inst_addr_i + 32'b0100) : (jump_addr_i);

always @ (*) begin
    need_predict_o = need_predict_i;
```

```
    inst_addr_o = inst_addr_i;
    jump_act_o = jump_flag_i;
    jump_flag_o = jump;
    jump_addr_o = {32{jump}} & addr;
    ... // 流水线暂停的分类讨论
end
```

ctrl&pc_reg连线

```verilog
pc_reg u_pc_reg(
    ...
    .jump_flag_i(ctrl_jump_flag_o),
    .jump_addr_i(ctrl_jump_addr_o)
);

ctrl u_ctrl(
    ...
    .need_predict_i(ex_need_predict_o),
    .need_predict_o(ctrl_need_predict_o),
    .inst_addr_i(ex_inst_addr_o),
    .bp_result_i(ie_bp_result_o),
    .jump_flag_i(ex_jump_flag_o),
    .jump_addr_i(ex_jump_addr_o),
    .jump_addr_o(ctrl_jump_addr_o),
    ...
);
```

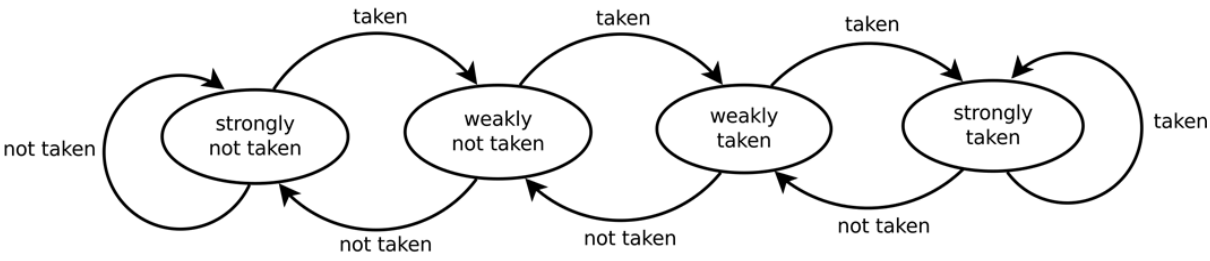
状态更新

通过ctrl模块向bpu模块传递参数last_need_predict(指示一次分支预测的结果知晓)、last_addr(上一次分支预测的指令地址)、last_jump(上一次分支预测的正确结果)，bpu模块中对不同分治策略需要维护的状态进行更新，具体的情况如下：

主要针对2bit饱和计数器、一级预测器、两级预测器这三个需要维护“状态”的动态分支预测方法，在bpu.v中用时序逻辑实现：

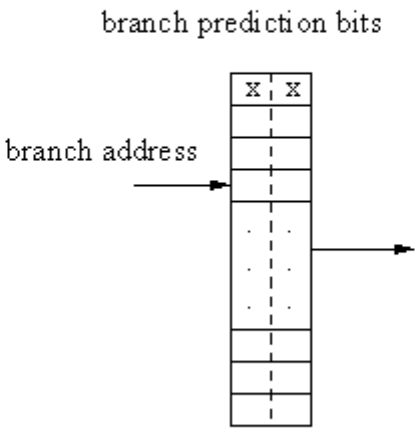
```
reg[2:0] state;          //  2bit饱和计数器
reg[1:0] dht[31:0];      //  一级预测器
reg[4:0] c_reg[31:0];    //  两级预测器的correlation registers
reg[1:0] gpt[31:0];      //  两级预测器的global pattern table
```

三种动态预测的策略：



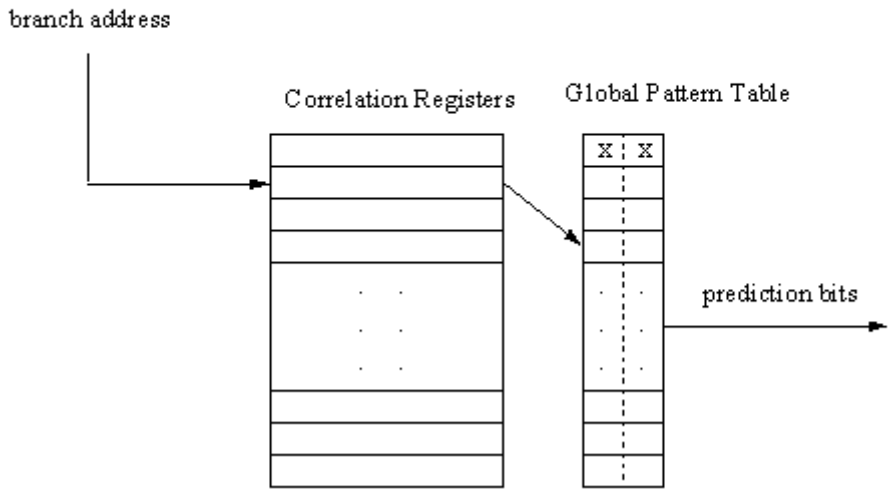
2bit饱和计数器的代码我是用case语句来实现的：

```verilog
 always @(posedge clk) begin
     if (rst == `RstEnable) begin
         state <= 2'b10;
     end else
     if (last_need_predict_i == 1'b1) begin
         case (state)
             2'b00: begin
                 if (last_jump_i == 1'b1) begin
                     state <= 2'b01;
                 end else begin
                     state <= 2'b00;
                 end
             end
             2'b01: begin
                 if (last_jump_i == 1'b1) begin
                     state <= 2'b10;
                 end else begin
                     state <= 2'b00;
                 end
             end
             2'b10: begin
                 if (last_jump_i == 1'b1) begin
                     state <= 2'b11;
                 end else begin
                     state <= 2'b01;
                 end
             end
             2'b11: begin
                 if (last_jump_i == 1'b1) begin
                     state <= 2'b11;
                 end else begin
                     state <= 2'b10;
                 end
             end
             default: begin end
         endcase
     end
   end
```



我的一级预测器的实现是开一个长度为32的数组，每个元素是一个2bit饱和计数器，通过for循环构建分类讨论的硬件结构，这里我利用了指令地址的对齐规则，用指令地址的[6:2]位确定其对应的2bit饱和计数器元素，代码：

```verilog
 always @(posedge clk) begin
     if (rst == `RstEnable) begin
         for (last_5 = 0; last_5 <= 31; last_5 = last_5 + 1)
             dht[last_5] = 2'b10;
     end else begin
     if (last_need_predict_i == 1'b1) begin
         for (last_5 = 0; last_5 <32; last_5 += 1) begin
             if (last_5 == last_addr_i[6:2]) begin
                 case (dht[last_5])
                     ...
                     default: begin end
                 endcase
             end
         end
         end
     end
   end
```



二级自适应预测器通过指令地址的[6:2]位确定一个近5次跳转情况对应的状态，并用这个状态在gpt中查找对应的2bit饱和计数器，代码：

```verilog
      if (rst == `RstEnable) begin
              for (i_5 = 0; i_5 < 32; i_5 = i_5 + 1) begin
                  c_reg[i_5] <= 5'b10101;
                  gpt[i_5] <= 2'b10;
              end
          end else begin
              if (last_need_predict_i == 1'b1) begin
                  for (i_5 = 0; i_5 < 32; i_5 = i_5 + 1) begin
                      if (i_5 == last_addr_i[6:2]) begin
                          c_reg[i_5] = {c_reg[i_5][3:0], last_jump_i};

                          for (j_5 = 0; j_5 < 32; j_5 += 1) begin
                              if (j_5 == c_reg[i_5]) begin
                                  case (gpt[j_5])
                                      ...
                                      default: begin end
                                  endcase
                              end
                          end
                      end
                  end
              end
          end
      end
```

细节

代码中有几个通过实验得出的小技巧：

- 用指令地址的[6:2]位来索引，这是利用了体系结构确定的地址对齐规则；
- 对计数器预热：不要将其初始状态设为2'b00，否则需要一些额外的预热费用；
- 使用循环代替case语句，这里的循环实际上是硬件的展开，写起来比较方便；

测试结果

	无分支预测	静态预测（后跳前不跳）	2bit饱和计数器	一级预测器	二级自适应预测器
always_jmp:	36192	32180	32196	32184	32188
dummy_branch:	90782	86768	74766	74758	75754
fixed_footprint:	54746	50732	48740	48736	49732

结果中其实一级预测器的结果是最好的，当然always_jmp中我的静态预测有一点cheat(我的jal是设置为总是跳转的)，观察波形图可以发现1级预测器能够很成功的让几乎每一条需要分支预测的语句拥有自己的2bit饱和计数器，加上合适的预热，效果是最好的；

而二级预测器的效果却很差，这其实和BHT中的5-bit状态的初始化有很大的关系，事实上最终我也没调出很好的初始化参数，感觉不同位置的分支语句即使有相同的跳转历史，也不一定就需要共用一个2bit饱和计数器，这是我对二级预测器效果不好的解释；