

任务

任务：检查图1的Verilog代码中包含的处理器元素，并解释其工作原理。

- 译码阶段显示的元素（寄存器文件、指令寄存器和控制单元）位于模块**dec**、**dec_decode_ctl**和**dec_gpr_ctl**中。
- EX1阶段显示的元素位于模块**exu**和**exu_alu_ctl**中。
- FC1阶段显示的元素位于模块**ifu**和**ifu_ifc_ctl**中。

FC1阶段：

- 2:1多路开关：模块**ifu_ifc_ctl**

```
278     assign ifc_fetch_addr_f1[31:1] = ( ({31{exu_flush_final}} & exu_flush_path_final[31:1]) |  
279     ({31{~exu_flush_final}} & ifc_fetch_addr_f1_raw[31:1]));
```

- 5:1多路开关：模块**ifu_ifc_ctl**

```
150     assign fetch_addr_bf[31:1] = ( ({31{miss_sel_flush}} & exu_flush_path_final[31:1]) | // FLUSH path  
151     ({31{sel_miss_addr_bf}} & miss_addr[31:1]) | // MISS path  
152     ({31{sel_btb_addr_bf}} & ifu_bp_btb_target_f2[31:1]) | // BTB target  
153     ({31{sel_last_addr_bf}} & ifc_fetch_addr_f1[31:1]) | // Last cycle  
154     ({31{sel_next_addr_bf}} & fetch_addr_next[31:1])); // SEQ path  
155
```

- 序列地址加法器：模块**ifu_ifc_ctl**

```
185     assign {overflow_nc, fetch_addr_next[31:1]} = ({1'b0, ifc_fetch_addr_f1[31:4]} + 29'b1), 3'b0);
```

EX1阶段：

- 比较器：模块**exu_alu_ctl**

```
145     assign eq = a_ff[31:0] == b_ff[31:0];
```

比较器会比较两个操作数：

- 如果两个操作数相等：eq = 1。
- 如果两个操作数不相等：eq = 0。

- 分支目标地址加法器：模块**exu_alu_ctl**

```
211     rvbradder ibradder (  
212     .pc(pc_ff[31:1]),  
213     .offset(brimm_ff[12:1]),  
214     .dout(pcout[31:1])  
215     );
```

加法器会计算PC与偏移量之和。

- 逻辑：模块**exu_alu_ctl**

```

202      assign actual_taken = (ap.beq & eq) |
203                          (ap.bne & ne) |
204                          (ap.blr & lt) |
205                          (ap.bge & ge) |
206                          (any_jal);
207

```

actual_taken中包含分支方向的结果：如果其值为1，则分支必须跳转；如果其值为0，则分支不得跳转。例如：

- 如果指令为beq指令 (ap.beq==1) 且两个操作数相等 (eq==1) → actual_taken = 1
- 如果指令为bne指令 (ap.bne==1) 且两个操作数不相等 (ne==1) → actual_taken = 1
- 如果指令为jal指令 (any_jal==1)，则分支必须跳转 → actual_taken = 1

```

230      assign cond_mispredict = (ap.predict_t & ~actual_taken) |
231                          (ap.predict_nt & actual_taken);
232

```

如果预测分支跳转 (ap.predict_t = 1) 但实际不跳转 (actual_taken = 0)，或者预测分支不跳转 (ap.predict_nt = 1) 但实际跳转 (actual_taken = 1)，则分支预测错误 (cond_mispredict = 1)

```

237      assign flush_upper = ( ap.jal | cond_mispredict | target_mispredict) & valid_ff & ~flush & ~freeze;
238

```

如果分支预测错误 (cond_mispredict = 1)、指令有效 (valid_ff = 1) 并且流水线尚未清除或冻结，则必须清除流水线。

任务：解释如何在模块exu_alu_ctl中通过信号eq、控制信号ap.beq、ap.predict_t和ap.predict_nt以及部分其他信号生成信号flush_upper。

- 逻辑：模块exu_alu_ctl

```

202      assign actual_taken = (ap.beq & eq) |
203                          (ap.bne & ne) |
204                          (ap.blr & lt) |
205                          (ap.bge & ge) |
206                          (any_jal);
207

```

actual_taken中包含分支方向的结果：如果其值为1，则分支必须跳转；如果其值为0，则分支不得跳转。例如：

- 如果指令为beq指令且两个操作数相等 → actual_taken = 1
- 如果指令为bne指令且两个操作数不相等 → actual_taken = 1
- 如果指令为jal指令，则分支必须跳转 → actual_taken = 1

```
230    assign cond_mispredict = (ap.predict_t & ~actual_taken) |
231    (ap.predict_nt & actual_taken);
232
```

如果预测分支跳转 ($ap.predict_t = 1$) 但实际未跳转 ($actual_taken = 0$)，或者预测分支不跳转 ($ap.predict_nt = 1$) 但实际跳转 ($actual_taken = 1$)，则分支预测错误 ($cond_mispredict = 1$)

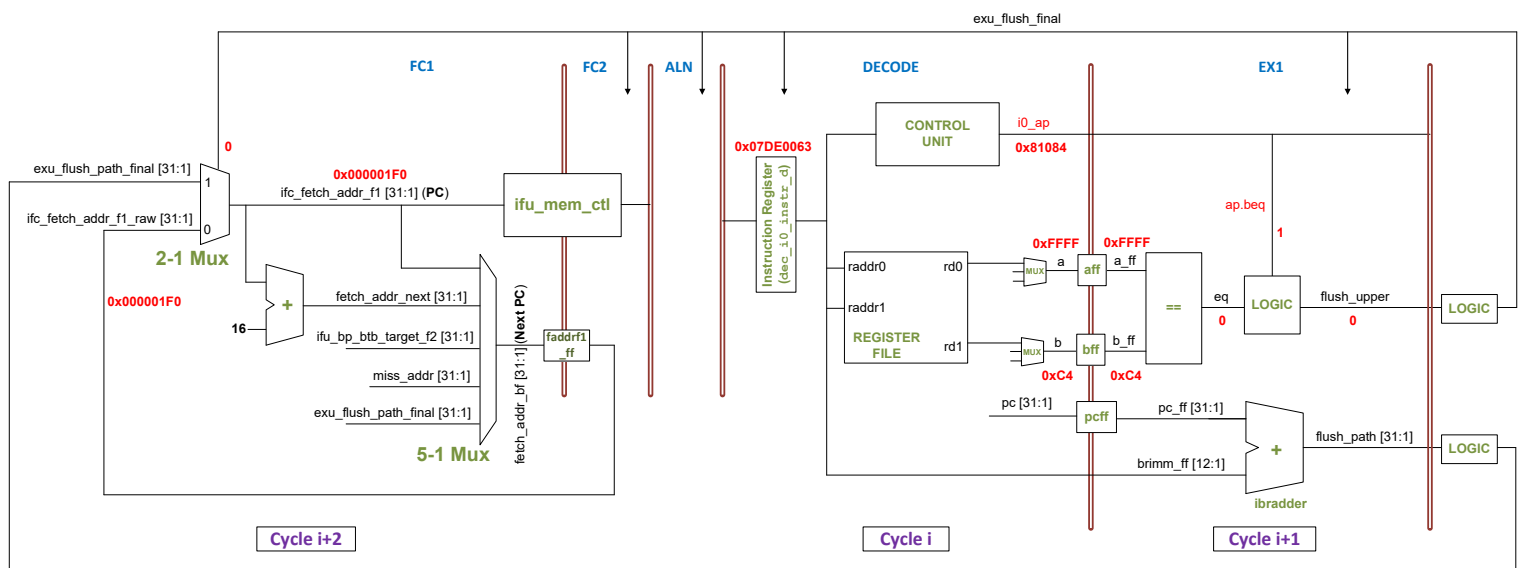
```
237    assign flush_upper = ( ap.jal | cond_mispredict | target_mispredict) & valid_ff & ~flush & ~freeze;
```

如果分支预测错误 ($cond_mispredict = 1$)、指令有效 ($valid_ff = 1$) 并且流水线尚未清除或冻结，则必须清除流水线。

任务： 在Verilog代码中分析信号`exu_flush_final`、`exu_flush_upper_e2`、`exu_i0_flush_final`和`exu_i1_flush_final`对EX1及其之前各阶段（FC1、FC2、对齐和译码）的影响。对于该分析，第2.B部分的仿真非常有用，您可以在其中加入所需的信号。

不提供解答。

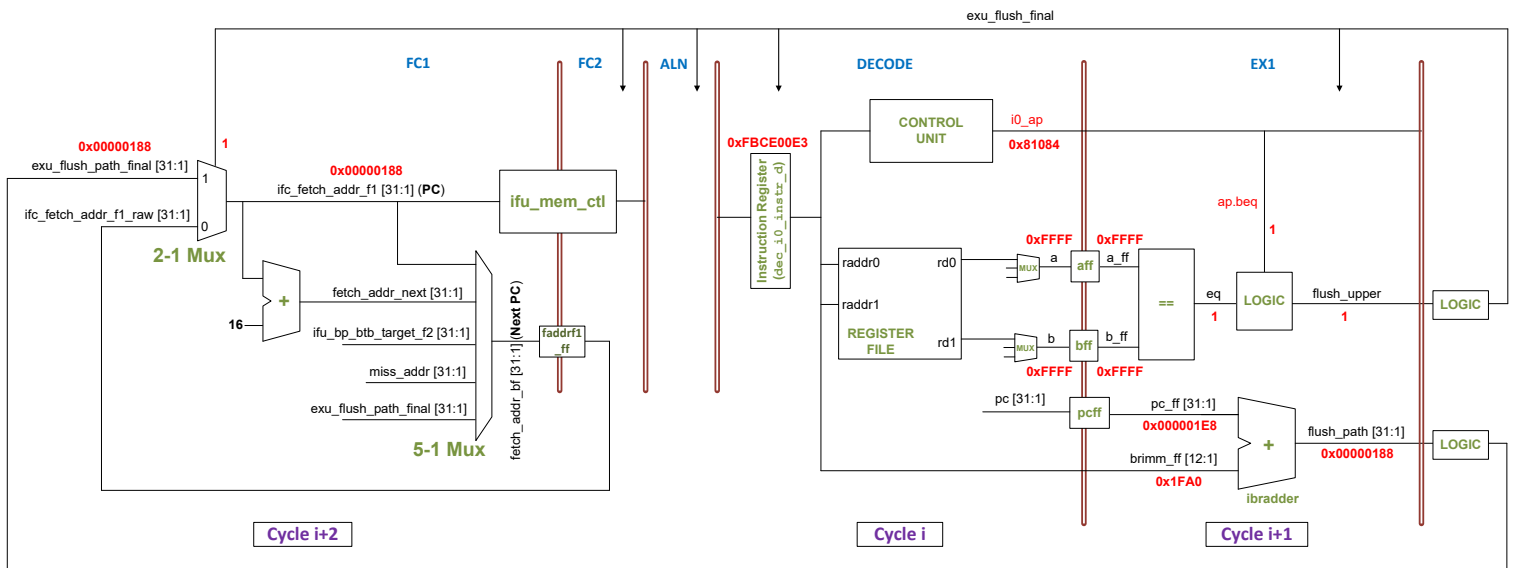
任务： 修改图1，将图3的周期*i*、*i+1*和*i+2*中所示的每个信号的值包含在内。



任务： 修改图2中的程序，让第一条分支指令通过转发的方式获取其输入操作数。

不提供解答。

任务： 修改图1，将图4的周期*i*、*i+1*和*i+2*中所示的每个信号的值包含在内。



任务：根据图2中的示例，检查不同情况下的信号，分析FC1中两个多路开关的操作。

例如，分析在按顺序执行指令（即一组没有分支的指令）的情况下如何完成取指。在这种情况下，您将看到SweRV EH1处理器进行如下操作：

- 在偶数周期中，使用5:1多路开关选择`fetch_addr_next`，该多路开关包含的值为当前取指地址（`ifc_fetch_addr_f1`）+ 16，因此会读取下一个连续的128位指令束（请记住，**I\$读操作提供128位**）。
- 在奇数周期中，使用5:1多路开关选择`ifc_fetch_addr_f1`，因此不会取出新的指令。这样，每2个周期会取出4条32位指令，这与译码阶段所需的取指速率相同（每个周期2条指令）。

注意，在DDCARV所述的处理器中，每个周期只需将PC的值加4（适用于按顺序执行指令的情况），从而在每个周期取一条指令。

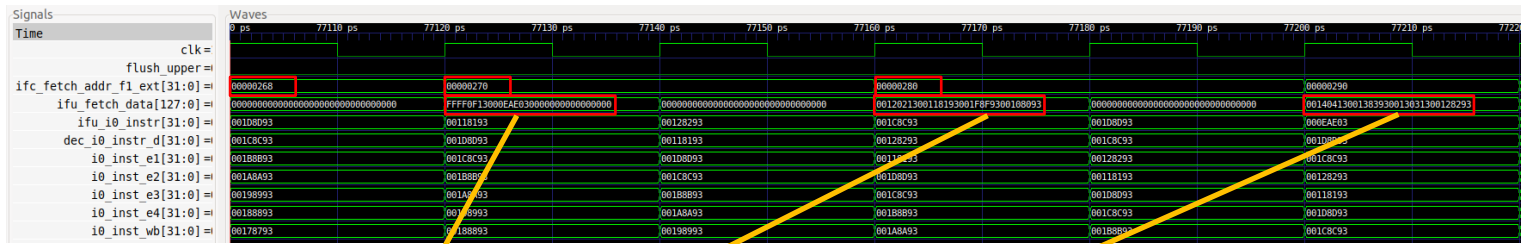
还可以修改图2中的程序以创建新的场景。例如，可以在跳转的分支后添加一些A-L指令，查看如何在重定向后清除这些指令。

按顺序执行:

使用以下来源:

- 程序路径: *[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory*
- Tcl脚本路径:
[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab16/test_SequentialExecution.tcl

在Verilator中可得到如下仿真结果:



268:	000eae03	lw	t3,0(t4)
26c:	ffff0f13	addi	t5,t5,-1
270:	00108093	addi	ra,ra,1
274:	001f8f93	addi	t6,t6,1
278:	00118193	addi	gp,gp,1
27c:	00120213	addi	tp,tp,1
280:	00128293	addi	t0,t0,1
284:	00130013	addi	t1,t1,1
288:	00138393	addi	t2,t2,1
28c:	00140413	addi	s0,s0,1

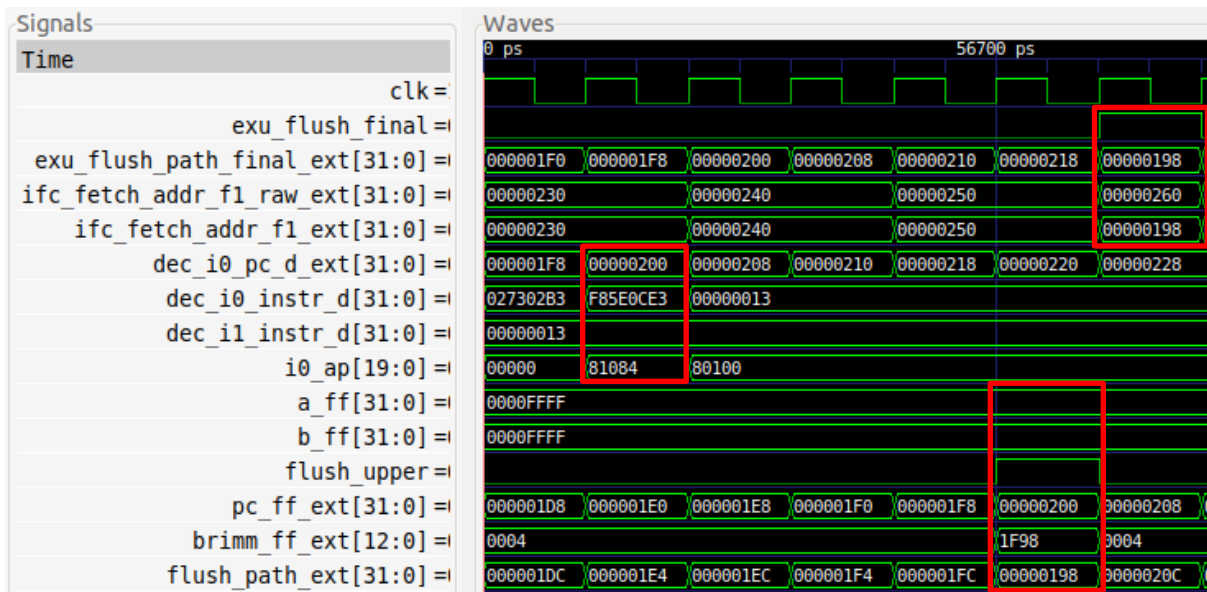
可以看出，每两个周期将取出一个新的128位指令束。

任务：在实验15中，我们已分析过如何在提交阶段通过辅助ALU消除写后读RAW数据冒险。与该实验探讨的A-L指令类似，如果先前执行过多周期操作，则条件分支指令可能产生写后读RAW数据冒险，必须在提交阶段消除冒险。如果确定分支预测错误，则必须在提交阶段清除流水线并重定向。请使用 `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction_HazardCommit` 文件夹中的程序（对图2中的程序进行了少许修改）和.tcl文件分析该情况。

生成的代码：

```
00000198 <LOOP>:
198: 001f0f13      addi  t5,t5,1
19c: 00000013      nop
1a0: 00000013      nop
1a4: 00000013      nop
1a8: 00000013      nop
1ac: 00000013      nop
1b0: 00000013      nop
1b4: 00000013      nop
1b8: 07de0463     beq  t3,t4,220 <OUT>
1bc: 00000013      nop
1c0: 00000013      nop
1c4: 00000013      nop
1c8: 00000013      nop
1cc: 00000013      nop
1d0: 00000013      nop
1d4: 00000013      nop
1d8: 001e8e93     addi  t4,t4,1
1dc: 00000013      nop
1e0: 00000013      nop
1e4: 00000013      nop
1e8: 00000013      nop
1ec: 00000013      nop
1f0: 00000013      nop
1f4: 00000013      nop
1f8: 027302b3     mul  t0,t1,t2
1fc: 00000013      nop
200: f85e0ce3     beq  t3,t0,198 <LOOP>
204: 00000013      nop
208: 00000013      nop
20c: 00000013      nop
210: 00000013      nop
214: 00000013      nop
218: 00000013      nop
21c: 00000013      nop
```

Verilator仿真:



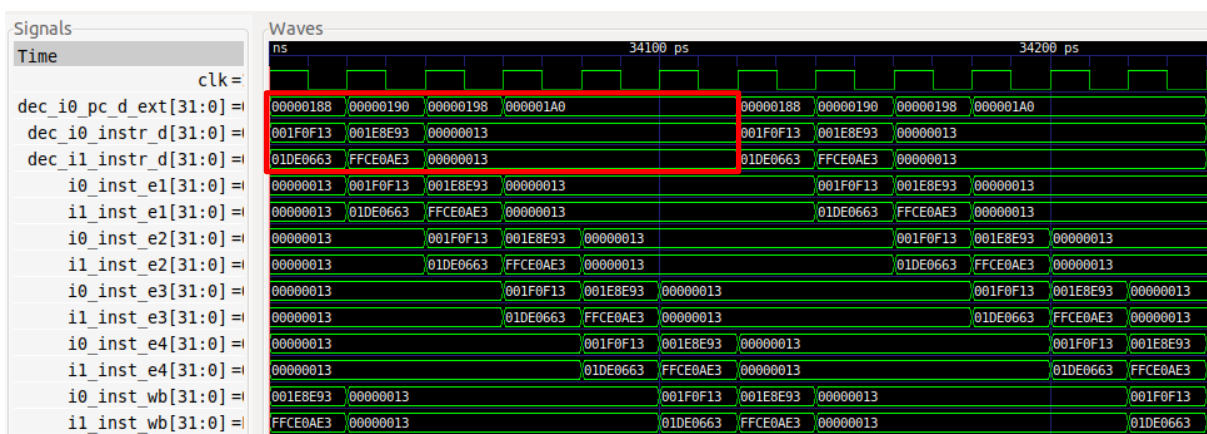
对beq指令（0xf85e0ce3）进行译码，经过EX1（以错误的操作数执行指令）、EX2和EX3阶段，最后进入提交阶段，该阶段将触发清除和重定向（flush_upper = exu_flush_final = 1），从而以正确的操作数再次执行指令。

任务：在图2的示例中，删除所有nop指令并分析仿真。然后通过开发板上执行程序，用性能计数器计算IPC。

使能SweRV EH1中使用的分支预测器（方法为注释掉图2中的两条初始化指令），并分析开发板上的仿真和执行情况。

比较两个实验并解释结果。

简单分支预测器:



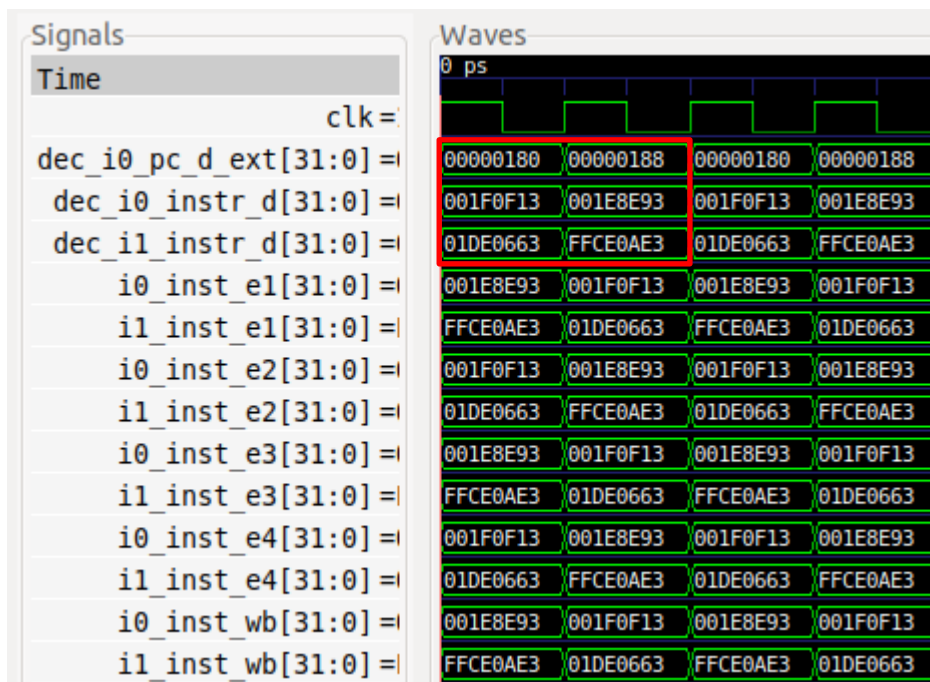
```

PIO Home  C Test.c  Test_Assembly.S x  startup.S
src > Test_Assembly.S
17 Test_Assembly:
18
19 li t2, 0x008           # Disable Branch Predictor
20 csrrs t1, 0x7F9, t2
21 //INSERT_NOPS_2
22
23 li t3, 0xFFFF
24 li t4, 0x1
25 li t5, 0x0
26 li t6, 0x0
27
28 LOOP:
29 add t5, t5, 1
30 beq t3, t4, OUT
31 add t4, t4, 1
32 beq t3, t3, LOOP
33 OUT:
34 INSERT_NOPS_8
35
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
> Executing task: platformio device monitor <
--- Available filters and text transformations: colorize, debug, c
--- More details at http://bit.ly/pio-monitor-filters
--- Monitor on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H
Cycles = 393468
Instructions = 262190

```

$$\text{IPC} = 262 / 393 = 0.67$$

Gshare分支预测器:




```

PIO Home  C Test.c  Test_Assembly.S x  startup.S  File
src > Test_Assembly.S
17 Test_Assembly:
18
19 //li t2, 0x008           # Disable Branch Predictor
20 //csrrs t1, 0x7F9, t2
21
22 li t3, 0xFFFF
23 li t4, 0x1
24 li t5, 0x0
25 li t6, 0x0
26
27 LOOP:
28     add t5, t5, 1
29     beq t3, t4, OUT
30     add t4, t4, 1
31     beq t3, t3, LOOP
32 OUT:
33     INSERT_NOPS_8
34
35 .end

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
> Executing task: platformio device monitor <
--- Available filters and text transformations: colorize, debug, def
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H --
Cycles = 131322
Instructions = 262188

```

$$IPC = 262 / 131 = 2$$

使用Gshare BP时，可以得到理想的IPC，但在使用简单BP时，由于第二个分支指令引发的清除和重定向，IPC远远达不到理想水平。

任务：分析上述所有哈希模块，尝试了解其工作原理及其在Gshare BP结构中的使用方式。

不提供解答。

任务：分析如何对这两个结构进行访问。

不提供解答。

任务：分析如何计算5:1多路开关的选择信号。

不提供解答。

任务：分析如何通过BTB中读取的值（btb_rd_tgt_f2[11:0]）和FC2中的取指地址（ifc_fetch_addr_f2[31:4]）获得预测目标地址（ifu_bp_btb_target_f2）。

模块ifu_bp_ctl:


```

1115 // compute target
1116 // Form the fetch group offset based on the btb hit location and the location of the branch within the 4 byte chunk
1117 assign btb_fg_crossing_f2 = btb_sel_f2[0] & btb_rd_pc4_f2;
1118
1119 wire [2:0] btb_sel_f2_enc, btb_sel_f2_enc_shift;
1120 assign btb_sel_f2_enc[2:0] = encode8_3(btb_sel_f2[7:0]);
1121 assign btb_sel_f2_enc_shift[2:0] = encode8_3(1'b0, btb_sel_f2[7:1]);
1122
1123 assign bp_total_branch_offset_f2[3:1] = (((3{ btb_rd_pc4_f2 } & btb_sel_f2_enc_shift[2:0]) |
1124                                         ((3{ btb_rd_pc4_f2 } & btb_sel_f2_enc[2:0]) |
1125                                         (3{ btb_fg_crossing_f2 })));
1126
1127
1128 logic [31:4] adder_pc_in_f2, ifc_fetch_addr_prior;
1129 rvdffe # (28) faddrf2_ff (., .en(ifc_fetch_req_f2 & ~ifu_bp_kill_next_f2 & ic_hit_f2), .din(ifc_fetch_addr_f2[31:4]), .dout(ifc_fetch_addr_prior[31:4]));
1130
1131 assign ifu_bp_poffset_f2[11:0] = btb_rd_tgt_f2[11:0];
1132
1133 assign adder_pc_in_f2[31:4] = ((28{ btb_fg_crossing_f2 } & ifc_fetch_addr_prior[31:4]) |
1134                               ((28{ ~btb_fg_crossing_f2 } & ifc_fetch_addr_f2[31:4]));
1135
1136 logic [31:0] pc_ext = {adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1], 1'b0};
1137 logic [12:0] offset_ext = {btb_rd_tgt_f2[11:0], 1'b0};
1138
1139
1140 rvbradder predtgt_addr (., p({adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1]},
1141                               .offset(btb_rd_tgt_f2[11:0]),
1142                               .dout(bp_btb_target_addr_f2[31:1])
1143                               );
1144
1145 // mux the btb target address here for a predicted return
1146 assign ifu_bp_btb_target_f2[31:1] = btb_rd_ret_f2 & ~btb_rd_call_f2 ? rets_out[0][31:1] : bp_btb_target_addr_f2[31:1];

```

任务：分析SweRV EH1处理器中实现的RAS。可通过搜索互联网获得有关该结构的更多操作信息（如访问http://www-classes.usc.edu/engr/ee-s/457/EE457_Classnotes/ee457_Branch_Prediction/EE560_05_Ras_Just_FYI.pdf）。

不提供解答。

任务：分析如何更新全局历史记录寄存器。

不提供解答。

练习

1) 实现一个双模分支预测器，并将其性能与Gshare BP的性能进行比较。

不提供解答。

2) （以下练习基于《计算机组织结构和设计》（RISC-V版本，Patterson & Hennessy ([HePa])）中的练习4.25。）

请看下面的循环：

```

LOOP: lw x10, 0(x13)
      lw x11, 4(x13)
      add x12, x10, x11
      add x13, x13, -8
      bnez x12, LOOP

```

假设采用了完美分支预测（在SweRV EH1中，只需避免使用第一次迭代即可模拟该行为），流水线具有完备的转发支持（同样是在SweRV EH1中），并且可在EX1阶段确定分支执行情况。

a. 展示该循环的第二次和第三次迭代的仿真结果。解释出现的行为。可使用[RVfpgaPath]/RVfpga/Labs/Lab16/HePa_Exercise-4-25中提供的程序。

