

## TAREFAS

**TAREFA:** Examine os elementos do processador incluídos na Figura 1 no código Verilog e explique o seu funcionamento.

- Os elementos apresentados no andar Decode (Register File, Registo de Instruções e Unidade de Controlo) podem ser encontrados nos módulos **dec**, **dec\_decode\_ctl** e **dec\_gpr\_ctl**.
- Os elementos apresentados no andar EX1 podem ser encontrados nos módulos **exu** e **exu\_alu\_ctl**.
- Os elementos apresentados no andar FC1 podem ser encontrados nos módulos **ifu** e **ifu\_ifc\_ctl**.

### Andar FC1:

- Multiplexer 2:1: Módulo **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]));
```

- Multiplexer 5:1: Módulo **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
```

- Somador para endereços sequenciais: Módulo **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 Stage:

- Comparador: Módulo **exu\_alu\_ctl**

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

Compara os dois operandos:

- Se são iguais:  $eq=1$ .
- Se são diferentes:  $eq=0$ .

- Somador para o endereço de destino do salto: Módulo **exu\_alu\_ctl**

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

Calcula a soma do PC com o offset.

- LOGICA: Módulo **exu\_alu\_ctl**

```

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

```

`actual_taken` contém a resolução da direção do salto: 1 se o salto tiver de ser efectuado e 0 se não tiver de ser efectuado. Por exemplo:

- o Se a instrução for um `beq` (`ap.beq==1`) e os dois operandos forem iguais (`eq==1`) → `actual_taken = 1`
- o Se a instrução for um `bne` (`ap.bne==1`) e os dois operandos forem diferentes (`ne==1`) → `actual_taken = 1`
- o Se a instrução for um `jal` (`any_jal==1`) o salto tem de ser efetuado → `actual_taken = 1`

```

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

```

O salto foi mal previsto (`cond_mispredict=1`) se foi previsto que fosse tomado (`ap.predict_t = 1`) e não deve ser tomado (`actual_taken = 0`), ou se foi previsto que não deve ser tomado (`ap.predict_nt = 1`) e deve ser tomado (`actual_taken = 1`)

```

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

```

O pipeline deve ser descarregado se tiver sido previsto incorretamente (`cond_mispredict=1`), a instrução é válida (`valid_ff=1`), e o pipeline não está a ser descarregado ou bloqueado.

**TAREFA:** Explique como o sinal `flush_upper` é gerado no módulo `exu_alu_ctl` a partir do sinal `eq`, sinais de controlo `ap.beq`, `ap.predict_t` e `ap.predict_nt`, e outros sinais.

#### - LOGICA: Módulo `exu_alu_ctl`

```

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

```

`actual_taken` contém a resolução da direção do salto: 1 se o salto tiver de ser efectuado e 0 se não tiver de ser efectuado. Por exemplo:

- o Se a instrução for um `beq` e os dois operandos forem iguais → `actual_taken = 1`
- o Se a instrução for um `bne` e os dois operandos forem diferentes → `actual_taken = 1`
- o Se a instrução for um `jal` o salto tem de ser efetuado → `actual_taken = 1`

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

O salto foi mal previsto (`cond_mispredict=1`) se foi previsto que fosse tomado (`ap.predict_t = 1`) e não deve ser tomado (`actual_taken = 0`), ou se foi previsto que não deve ser tomado (`ap.predict_nt = 1`) e deve ser tomado (`actual_taken = 1`)

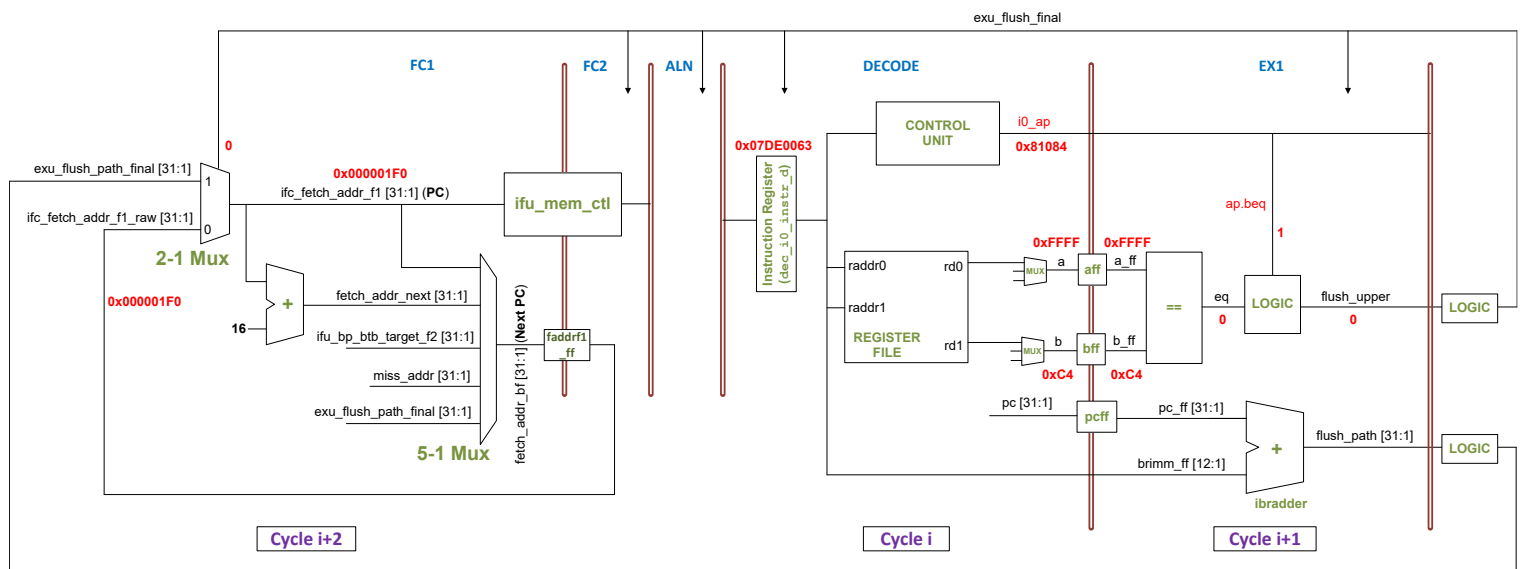
```
237    assign flush_upper = ( ap.jal | cond_mispredict | target_mispredict) & valid_ff & ~flush & ~freeze;
```

O pipeline deve ser descarregado se tiver sido previsto incorretamente (`cond_mispredict=1`), a instrução é válida (`valid_ff=1`), e o pipeline não está a ser descarregado ou bloqueado.

**TAREFA:** Analise no código Verilog o efeito dos sinais `exu_flush_final`, `exu_flush_upper_e2`, `exu_i0_flush_final` e `exu_il_flush_final` em EX1 e nos andares que o precedem: FC1, FC2, Align e Decode. Para esta análise, pode ser útil utilizar as simulações da Secção 2.B, onde pode incluir os sinais de que necessita.

Solução não fornecida.

**TAREFA:** Modifique a figura 1 para incluir os valores de cada sinal indicado na figura 3 nos ciclos *i*, *i+1*, e *i+2*.



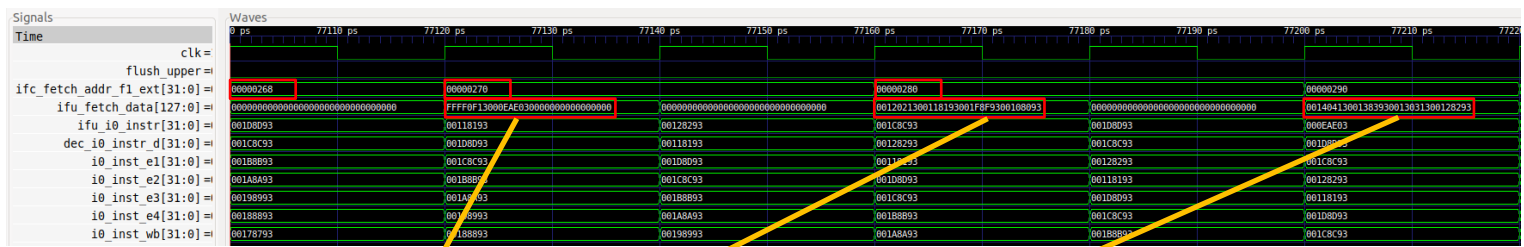
**TAREFA:** Modifique o programa da Figura 2 para fazer com que a primeira instrução de salto obtenha os seus operandos de entrada através de forwarding.

Solução não fornecida.

**TAREFA:** Modifique a figura 1 para incluir os valores de cada sinal indicado na figura 4 nos ciclos *i*, *i+1*, e *i+2*.

Modifique também o programa da Figura 2 para criar novos cenários. Por exemplo, pode adicionar algumas instruções A-L após o salto tomado e ver como são descarregadas após o redirecionamento.

Pode obter a seguinte simulação no Verilator:



|      |          |      |          |
|------|----------|------|----------|
| 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: | 00130313 | addi | t1,t1,1  |
| 288: | 00138393 | addi | t2,t2,1  |
| 28c: | 00140413 | addi | s0,s0,1  |

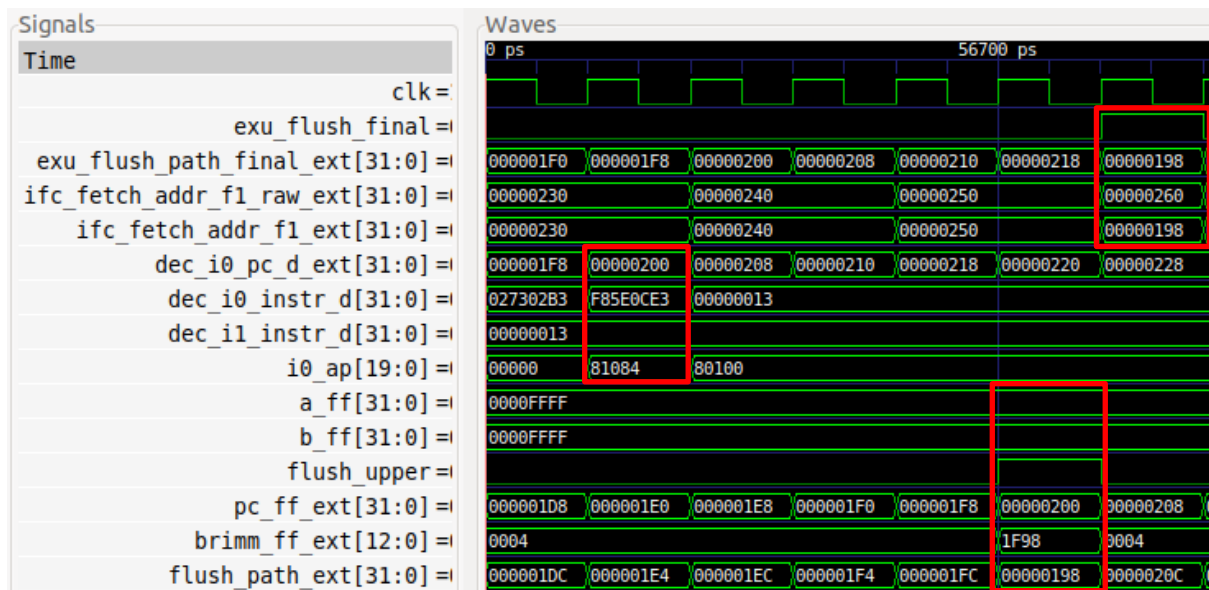
Podemos ver que a cada dois ciclos é carregado um novo pacote de 128 bits.

**TAREFA:** No Lab 15, analisámos como os conflitos de dados RAW são resolvidos no andar Commit através das ALUs secundárias. À semelhança das instruções A-L que estudámos nesse laboratório, uma instrução de salto condicional pode ter um conflito de dados RAW com uma operação multi-ciclo anterior que tem de ser resolvido no momento do Commit. Se for determinado que o salto foi mal previsto, o pipeline deve ser descarregado e redirecionado a partir do andar Commit. Analise esta situação utilizando uma versão ligeiramente modificada do programa da Figura 2, fornecida em `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction_HazardCommit`, e o script `.tcl` fornecido nessa mesma pasta.

### Código gerado:

```
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
```

## Simulação no Verilator:



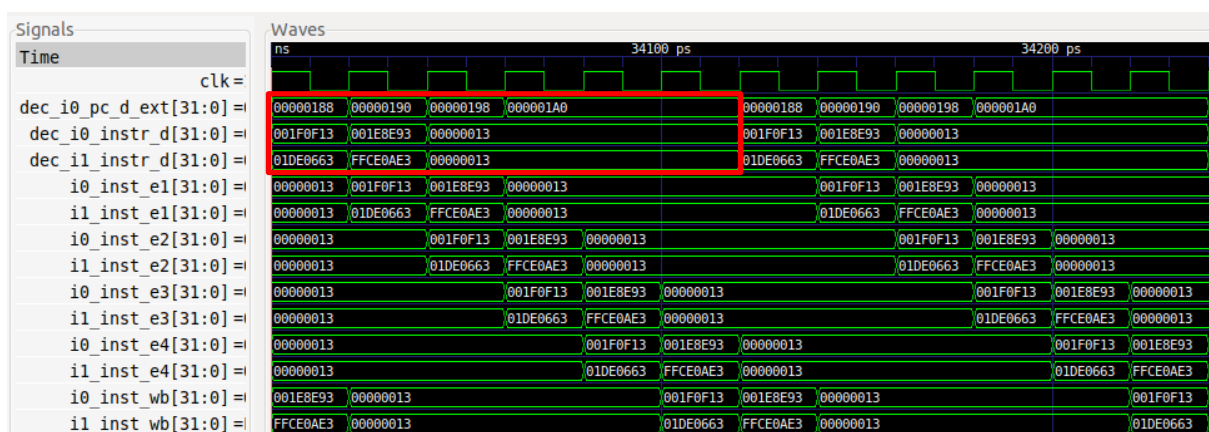
A instrução `beq` (0xf85e0ce3) é decodificado, passa por EX1 (onde executa nos operandos errados), passa por EX2 e EX3, e depois passa por Commit onde executa novamente nos operandos correctos, desencadeando uma descarga e um redireccionamento (`flush_upper = exu_flush_final = 1`).

**TAREFA:** No exemplo da Figura 2, remova todas as instruções `nop` e analise a simulação. Em seguida, calcule o IPC com os Performance Counters, executando o programa na placa.

Active o preditor de saltos utilizado no SweRV EH1 (comentando as duas instruções iniciais da Figura 2) e analise a simulação e a execução na placa.

Compare as duas experiências e explique os resultados.

## Preditor de saltos Naïve:



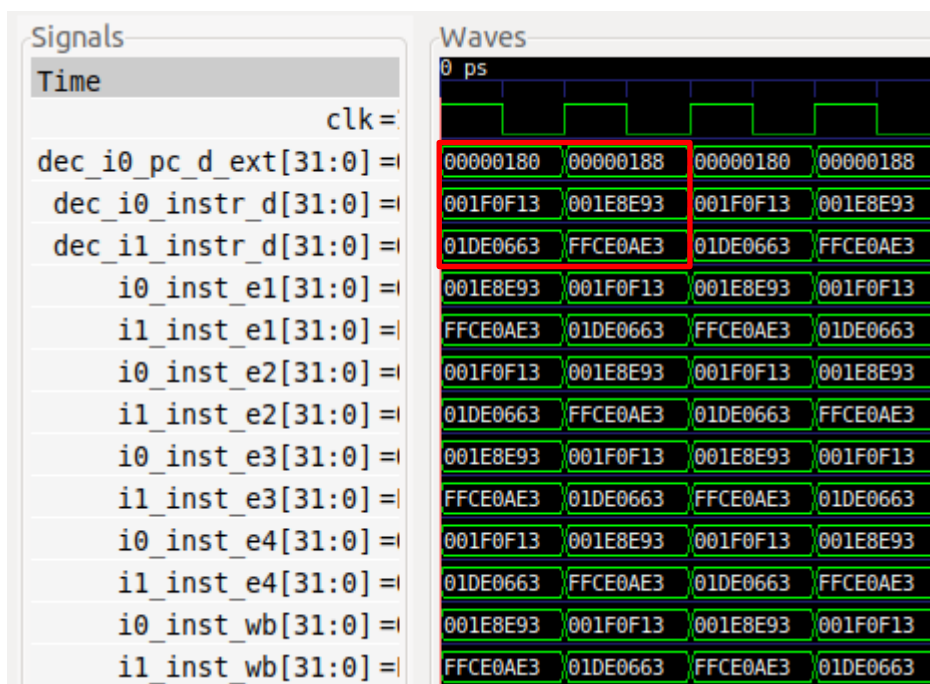
```

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

```

$$IPC = 262 / 393 = 0.67$$

Preditor de saltos Gshare:



```

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

```

$$\text{IPC} = 262 / 131 = 2$$

O IPC é ideal quando utiliza o BP Gshare, mas está longe de ser ideal quando utiliza o BP Naïve devido à descarga e ao redirecionamento causados pela segunda instrução de salto.

**TAREFA:** Analise todos estes módulos de hashing e tente ter uma ideia de como funcionam e como são utilizados nas estruturas BP do Gshare.

Solução não fornecida.

**TAREFA:** Analise como é efetuado o acesso a estas duas estruturas.

Solução não fornecida.

**TAREFA:** Analise como é calculado o sinal de seleção do multiplexer 5:1.

Solução não fornecida.

**TAREFA:** Analise a forma como o endereço-destino previsto (`ifu_bp_btb_target_f2`) é obtido a partir do valor lido na BTB (`btb_rd_tgt_f2[11:0]`) e o endereço de Fetch em FC2 (`ifc_fetch_addr_f2[31:4]`).

Módulo `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 logic [31:4] adder_pc_in_f2, ifc_fetch_addr_prior;
1128 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]));
1129
1130 assign ifu_bp_poffset_f2[11:0] = btb_rd_tgt_f2[11:0];
1131
1132 assign adder_pc_in_f2[31:4] = ( ((28{ btb_fg_crossing_f2 } & ifc_fetch_addr_prior[31:4]) |
1133      ((28{~btb_fg_crossing_f2} & ifc_fetch_addr_f2[31:4]) );
1134
1135 logic [31:0] pc_ext = {adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1], 1'b0};
1136 logic [12:0] offset_ext = {btb_rd_tgt_f2[11:0], 1'b0};
1137
1138 rvbradder predtgt_addr (., p({adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1]}),
1139      .offset(btb_rd_tgt_f2[11:0]),
1140      .dout(bp_btb_target_addr_f2[31:1]));
1141
1142 // mux the return address here for a predicted return
1143 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];
1144
1145

```

**TAREFA:** Analise o RAS implementado no processador SweRV EH1. Uma pesquisa na Internet fornecerá também informações adicionais sobre o funcionamento desta estrutura (por exemplo [http://www-classes.usc.edu/engr/ee-s/457/EE457\\_Classnotes/ee457\\_Branch\\_Prediction/EE560\\_05\\_Ras\\_Just\\_FYI.pdf](http://www-classes.usc.edu/engr/ee-s/457/EE457_Classnotes/ee457_Branch_Prediction/EE560_05_Ras_Just_FYI.pdf)).

Solução não fornecida.

**TAREFA:** Analisar a forma como o Global History Register é atualizado.

Solução não fornecida.

## EXERCÍCIOS

- 1) 1) Implemente um preditor de saltos bimodal e compare o seu desempenho em relação ao BP Gshare.

Solução não fornecida.

- 2) (O exercício seguinte baseia-se no exercício 4.25 do livro “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([HePa]).)

Considere o seguinte ciclo:

```

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

```

Assuma que é utilizada uma previsão perfeita de saltos (no caso do SweRV EH1, podemos emular este comportamento evitando simplesmente a primeira iteração), que o pipeline tem suporte de forwarding completo (mais uma vez, é o caso do SweRV EH1) e que os saltos são resolvidos no andar EX1.

- a. Apresente uma simulação para a segunda e terceira iterações deste ciclo. Explique o comportamento obtido. Pode utilizar o programa fornecido em [\[RVfpgaPath\]/RVfpga/Labs/Lab16/HePa\\_Exercise-4-25](#).

