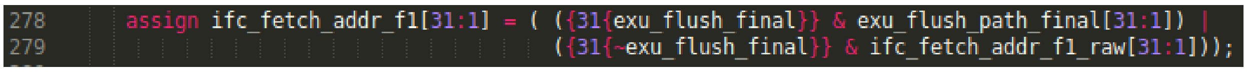
TASKS

TASK: Examine the processor elements included in Figure 1 in the Verilog code and explain how they work.
- The elements shown in the Decode stage (Register File, Instruction Register and Control Unit) can be found in modules **dec**, **dec_decode_ctl** and **dec_gpr_ctl**.
- The elements shown in the EX1 stage can be found in modules **exu** and **exu_alu_ctl**.
- The elements shown in the FC1 stage can be found in modules **ifu** and **ifu_ifc_ctl**.

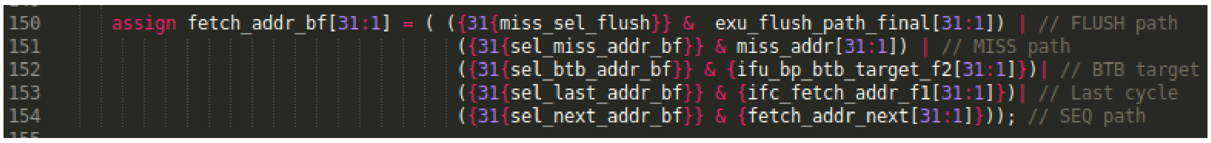
FC1 Stage:

- 2:1 Multiplexer: Module **ifu_ifc_ctl**



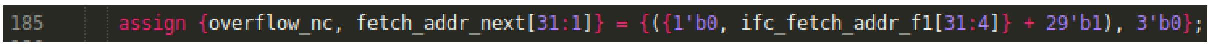
```
assign ifc_fetch_addr_f1[31:1] = ( ({31{exu_flush_final}} & exu_flush_path_final[31:1]) |
                                   ({31{~exu_flush_final}} & ifc_fetch_addr_f1_raw[31:1]));
```

- 5:1 Multiplexer: Module **ifu_ifc_ctl**



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

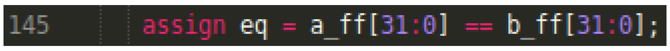
- Adder for sequential address: Module **ifu_ifc_ctl**



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

EX1 Stage:

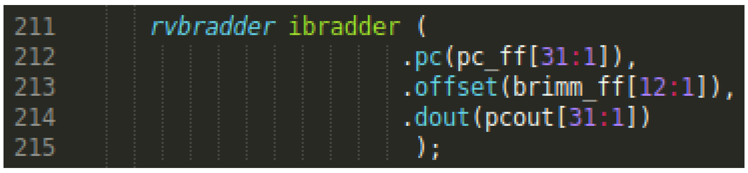
- Comparator: Module **exu_alu_ctl**



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

It compares the two operands:
- If they are equal: `eq`=1.
- If they are different: `eq`=0.

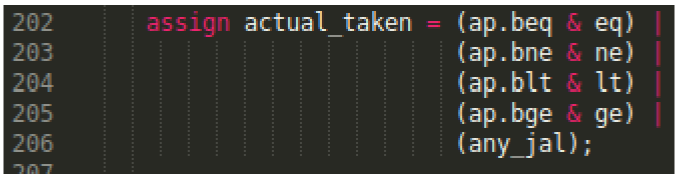
- Adder for the branch target address: Module **exu_alu_ctl**



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

It computes the addition of the PC and the offset.

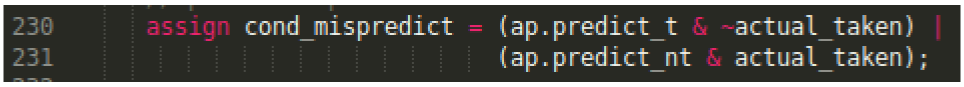
- LOGIC: Module **exu_alu_ctl**



```
assign actual_taken = (ap.beq & eq) |
                      (ap.bne & ne) |
                      (ap.blt & lt) |
                      (ap.bge & ge) |
                      (any_jal);
```

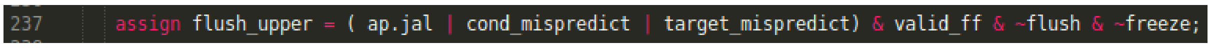
`actual_taken` contains the resolution of the branch direction: 1 if the branch must be taken and 0 if it must be not-taken. For example:

- If the instruction is a `beq` (`ap.beq==1`) and the two operands are equal (`eq==1`) → `actual_taken` = 1
- If the instruction is a `bne` (`ap.bne==1`) and the two operands are different (`ne==1`) → `actual_taken` = 1
- If the instruction is a `jal` (`any_jal==1`) the branch must be taken → `actual_taken` = 1



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

The branch has been mispredicted (`cond_mispredict`=1) if it was predicted taken (`ap.predict_t` = 1) and it must be not-taken (`actual_taken` = 0), or if it was predicted not-taken (`ap.predict_nt` = 1) and it must be taken (`actual_taken` = 1)

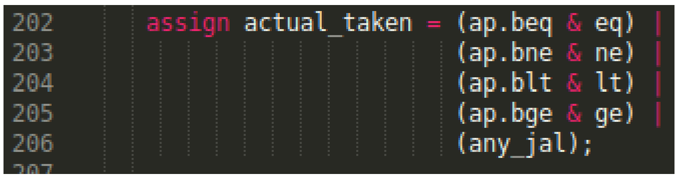


```
assign flush_upper = ( ap.jal | cond_mispredict | target_mispredict) & valid_ff & ~flush & ~freeze;
```

The pipeline must be flushed if it was mispredicted (`cond_mispredict`=1), the instruction is valid (`valid_ff`=1), and the pipeline is not being flushed or frozen.

<u>TASK</u>: Explain how signal `flush_upper` is generated in module **exu_alu_ctl** from signal `eq`, control signals `ap.beq`, `ap.predict_t` and `ap.predict_nt`, and some other signals.

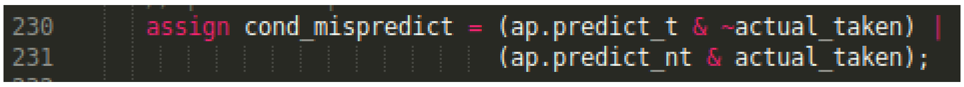
- LOGIC: Module **exu_alu_ctl**



```
assign actual_taken = (ap.beq & eq) |
                      (ap.bne & ne) |
                      (ap.blt & lt) |
                      (ap.bge & ge) |
                      (any_jal);
```

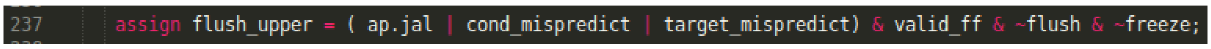
`actual_taken` contains the resolution of the branch direction: 1 if the branch must be taken and 0 if it must be not-taken. For example:

- If the instruction is a `beq` and the two operands are equal → `actual_taken` = 1
- If the instruction is a `bne` and the two operands are different → `actual_taken` = 1
- If the instruction is a `jal` the branch must always be taken → `actual_taken` = 1



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

The branch has been mispredicted (`cond_mispredict`=1) if it was predicted taken (`ap.predict_t` = 1) and it is not actually taken (`actual_taken` = 0), or if it was predicted not taken (`ap.predict_nt` = 1) and it is actually taken (`actual_taken` = 0)



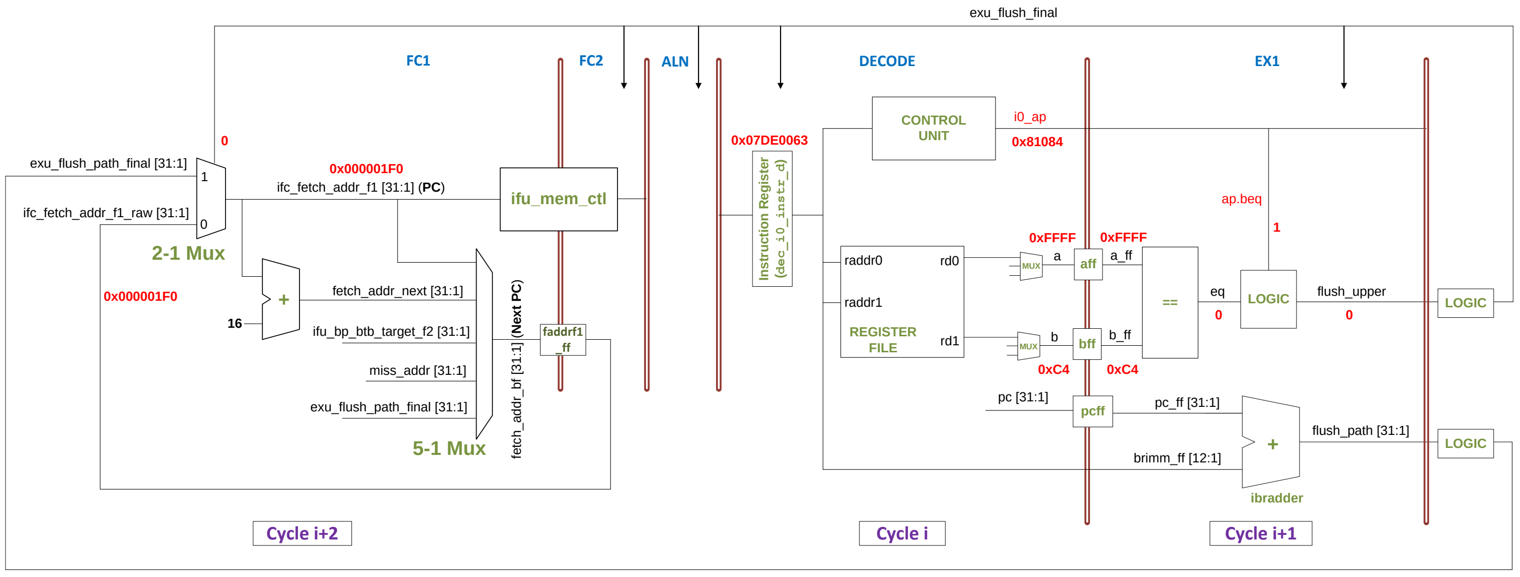
```
   assign flush_upper = ( ap.jal | cond_mispredict | target_mispredict) & valid_ff & ~flush & ~freeze;
```

The pipeline must be flushed if it was mispredicted (`cond_mispredict`=1), the instruction is valid (`valid_ff`=1), and the pipeline is not being flushed or frozen.

TASK: Analyse in the Verilog code the effect of signals `exu_flush_final`, `exu_flush_upper_e2`, `exu_i0_flush_final` and `exu_i1_flush_final` in EX1 and in the stages preceding it: FC1, FC2, Align, and Decode. For this analysis, it can be useful to use the simulations from Section 2.B, where you can include the signals that you need.

Solution not provided.

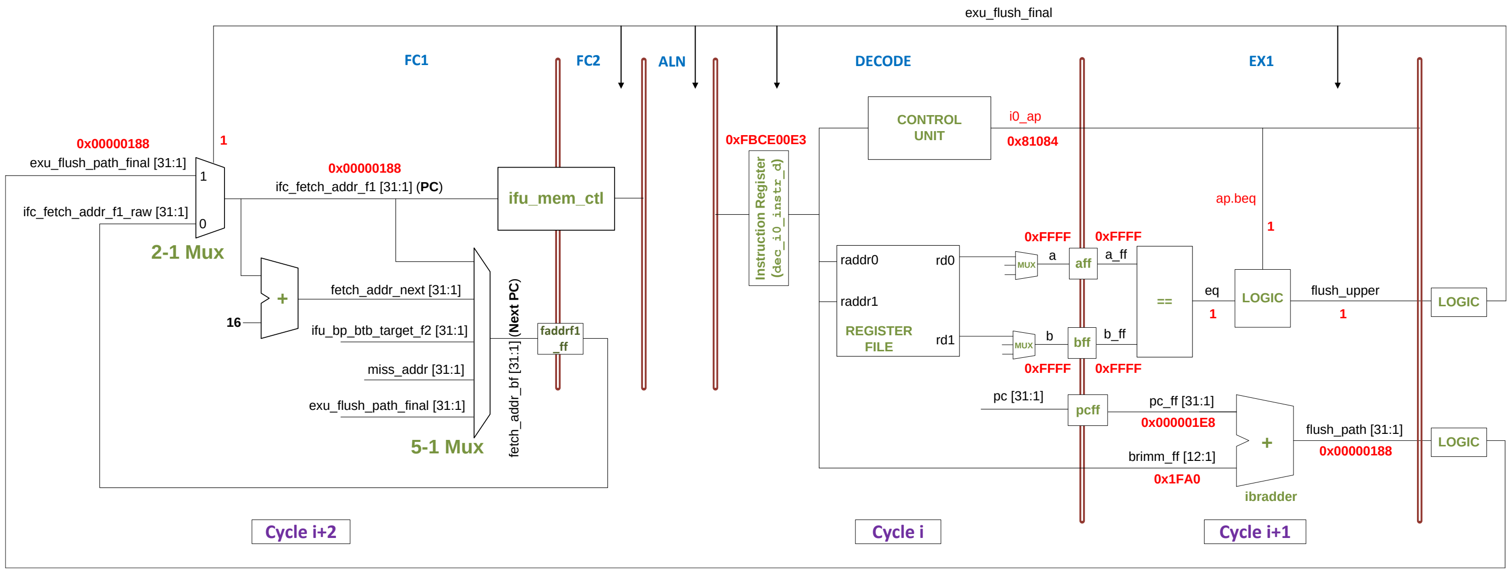
TASK: Modify Figure 1 to include the values of each signal shown in Figure 3 in cycles *i*, *i+1*, and *i+2*.



TASK: Modify the program from Figure 2 to make the first branch instruction retrieve its input operands through forwarding.

Solution not provided.

TASK: Modify Figure 1 to include the values of each signal shown in Figure 4 in cycles *i*, *i+1*, and *i+2*.



TASK: Analyse the operation of the two multiplexers from FC1 with the example from Figure 2, examining the signals under different circumstances.

For example, analyse how fetch is accomplished for sequential execution (i.e. a group of instructions with no branches). You will see that, in the SweRV EH1 processor, the operation in this case is as follows:
- In the even cycles, the `fetch_addr_next` is selected using the 5:1 multiplexer, which contains the current Fetch Address (`ifc_fetch_addr_f1`) plus 16, thus reading the next sequential 128-bit bundle of instructions (remember that an I$ read provides 128 bits).
- In the odd cycles, the `ifc_fetch_addr_f1` is selected using the 5:1 multiplexer, thus no new instructions are fetched.

This way, four 32-bit instructions are fetched every 2 cycles, which is the same rate of instructions needed by the Decode stage (2 instructions per cycle).
Note that in the processors from DDCARV the PC is simply incremented by four in every cycle (for sequential execution) to fetch one instruction per cycle.

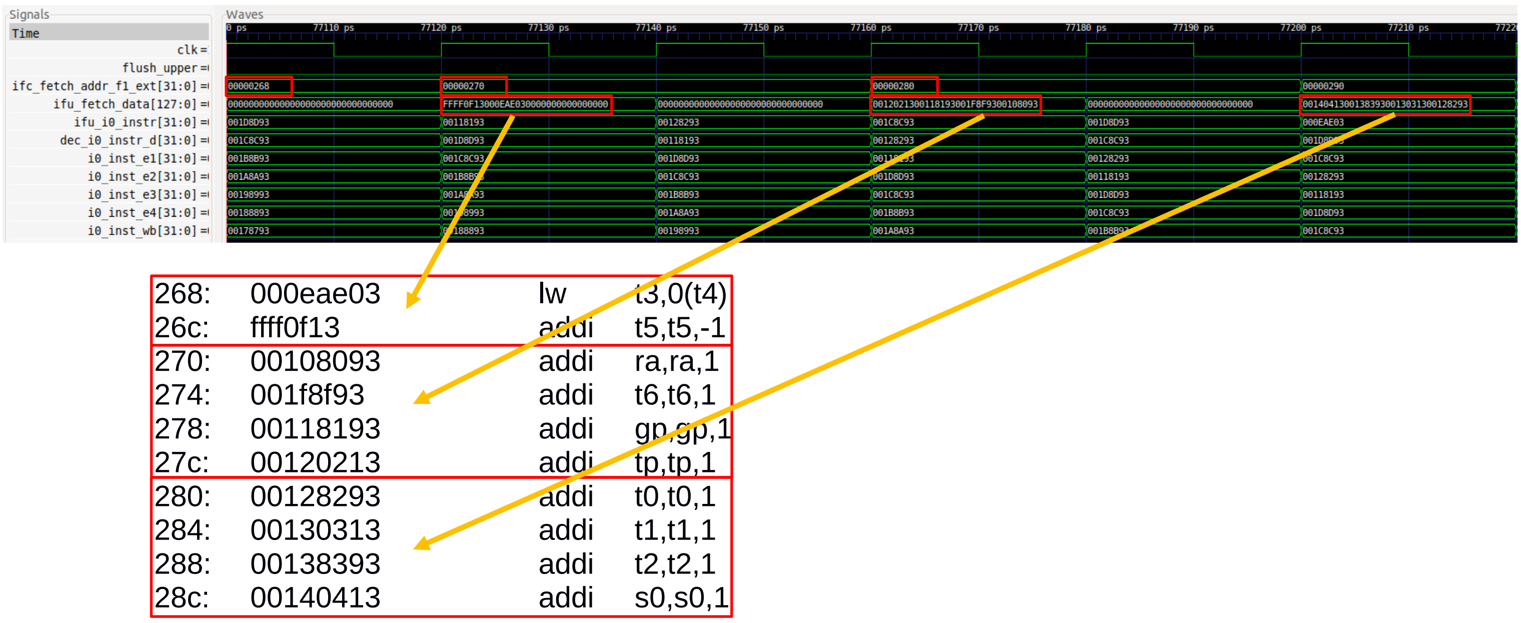
Also modify the program from Figure 2 to create new scenarios. For example, you can add some A-L instructions after the taken branch and see how they are flushed after the redirection.

SEQUENTIAL EXECUTION:

Using the following sources:

- Program from: *[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory*
- Tcl Script from: *[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab16/test_SequentialExecution.tcl*

We can obtain the following simulation in Verilator:

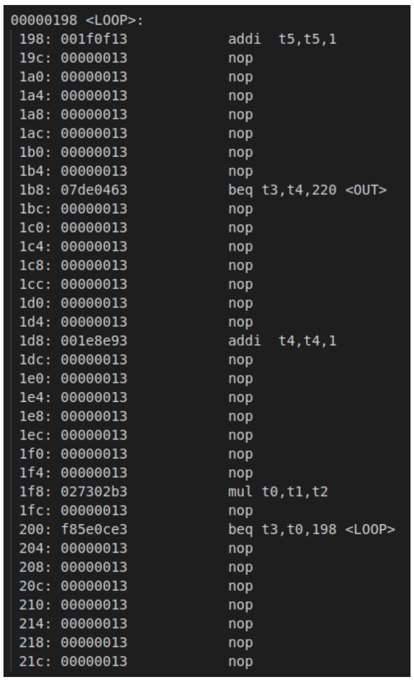


```
268:   000eae03        lw     t3,0(t4)
26c:   ffff0f13        addi   t5,t5,-1
270:   00108093        addi   ra,ra,1
274:   001f8f93        addi   t6,t6,1
278:   00118193        addi   gp,gp,1
27c:   00120213        addi   tp,tp,1
280:   00128293        addi   t0,t0,1
284:   00130313        addi   t1,t1,1
288:   00138393        addi   t2,t2,1
28c:   00140413        addi   s0,s0,1
```

We can see that every two cycles a new 128-bit bundle is fetched.

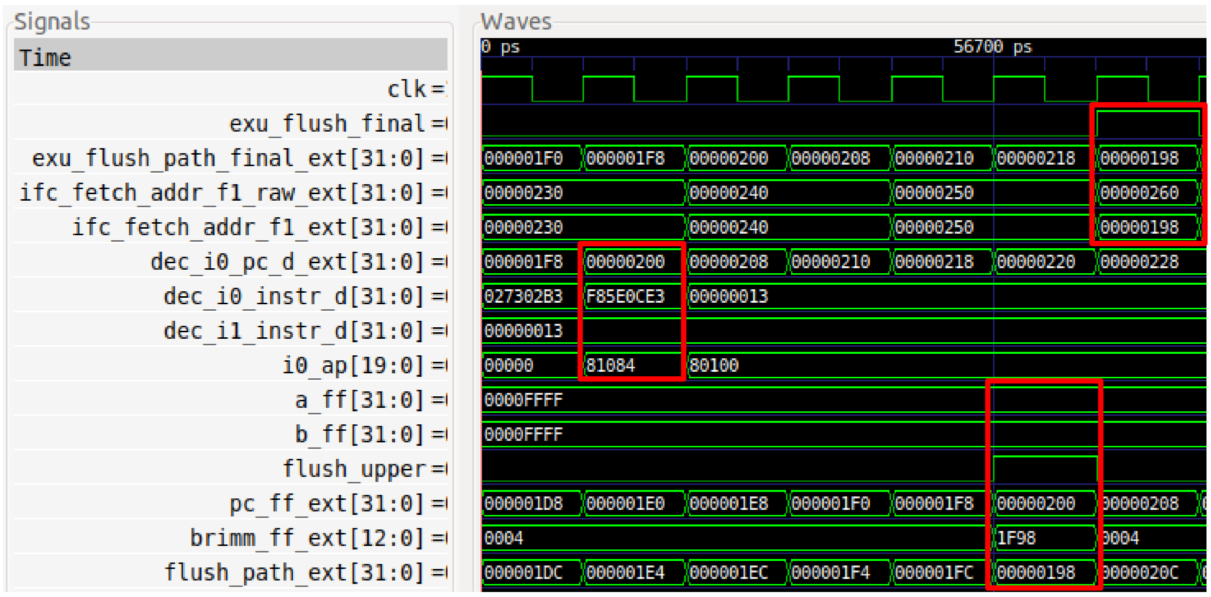
TASK: In Lab 15, we analysed how RAW data hazards are resolved in the Commit stage by means of the Secondary ALUs. Similar to the A-L instructions that we studied in that lab, a conditional branch instruction can have a RAW data hazard with a previous multi-cycle operation that must be resolved at commit time. If the branch is determined to have been mispredicted, the pipeline must be flushed and redirected from the Commit stage. Analyse this situation using a slightly modified version of the program from Figure 2, provided at *[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction_HazardCommit*, and the *.tcl* file provided in that same folder.

Code generated:



```
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 Simulation:


Signals
Waves
Time
0 ps
56700 ps
clk
exu_flush_final
exu_flush_path_final_ext[31:0]
ifc_fetch_addr_f1_raw_ext[31:0]
ifc_fetch_addr_f1_ext[31:0]
dec_i0_pc_d_ext[31:0]
dec_i0_instr_d[31:0]
dec_i1_instr_d[31:0]
i0_ap[19:0]
a_ff[31:0]
b_ff[31:0]
flush_upper
pc_ff_ext[31:0]
brimm_ff_ext[12:0]
flush_path_ext[31:0]
000001F0 000001F8 00000200 00000208 00000210 00000218 00000198
00000230 00000240 00000250 00000260
00000230 00000240 00000250 00000198
000001F8 00000200 00000208 00000210 00000218 00000220 00000228
027302B3 F85E0CE3 00000013
00000013
00000 81084 80100
0000FFFF
0000FFFF
000001D8 000001E0 000001E8 000001F0 000001F8 00000200 00000208
0004 1F98 0004
000001DC 000001E4 000001EC 000001F4 000001FC 00000198 0000020C

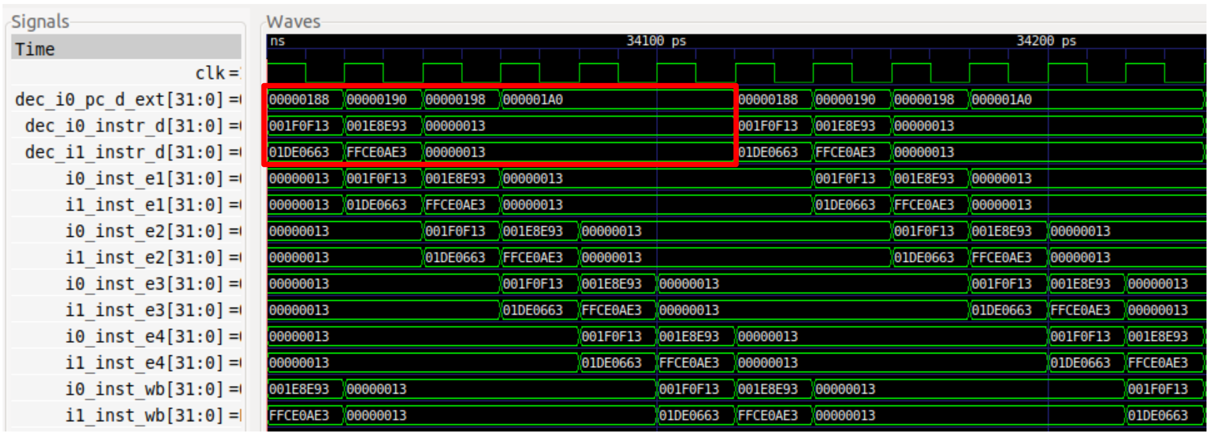
The `beq` instruction (0xf85e0ce3) is decoded, goes through EX1 (where it executes on the wrong operands), goes through EX2 and EX3, and then it goes through Commit where it executes again on the correct operands, triggering a flush and redirection (`flush_upper` = `exu_flush_final` = 1).

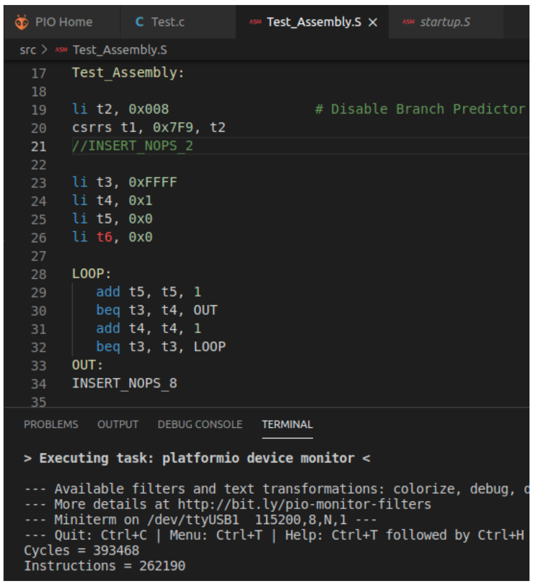
<u>TASK</u>: In the example from Figure 2, remove all the `nop` instructions and analyse the simulation. Then compute the IPC with the Performance Counters by executing the program on the board.

Enable the branch predictor used in SweRV EH1 (by commenting out the two initial instructions in Figure 2) and analyse the simulation and the execution on the board.

Compare the two experiments and explain the results.

Naïve Branch Predictor:


Signals
Waves
Time
ns
34100 ps
34200 ps
clk
dec_i0_pc_d_ext[31:0]
dec_i0_instr_d[31:0]
dec_i1_instr_d[31:0]
i0_inst_e1[31:0]
i1_inst_e1[31:0]
i0_inst_e2[31:0]
i1_inst_e2[31:0]
i0_inst_e3[31:0]
i1_inst_e3[31:0]
i0_inst_e4[31:0]
i1_inst_e4[31:0]
i0_inst_wb[31:0]
i1_inst_wb[31:0]
00000188 00000190 00000198 000001A0 00000188 00000190 00000198 000001A0
001F0F13 001E8E93 00000013 001F0F13 001E8E93 00000013
01DE0663 FFCE0AE3 00000013 01DE0663 FFCE0AE3 00000013
00000013 001F0F13 001E8E93 00000013 001F0F13 001E8E93 00000013
00000013 01DE0663 FFCE0AE3 00000013 01DE0663 FFCE0AE3 00000013
00000013 001F0F13 001E8E93 00000013 001F0F13 001E8E93 00000013
00000013 01DE0663 FFCE0AE3 00000013 01DE0663 FFCE0AE3 00000013
00000013 001F0F13 001E8E93 00000013 001F0F13 001E8E93 00000013
00000013 01DE0663 FFCE0AE3 00000013 01DE0663 FFCE0AE3 00000013
00000013 001F0F13 001E8E93 00000013 001F0F13 001E8E93
00000013 01DE0663 FFCE0AE3 00000013 01DE0663 FFCE0AE3
001E8E93 00000013 001F0F13 001E8E93 00000013 001F0F13
FFCE0AE3 00000013 01DE0663 FFCE0AE3 00000013 01DE0663

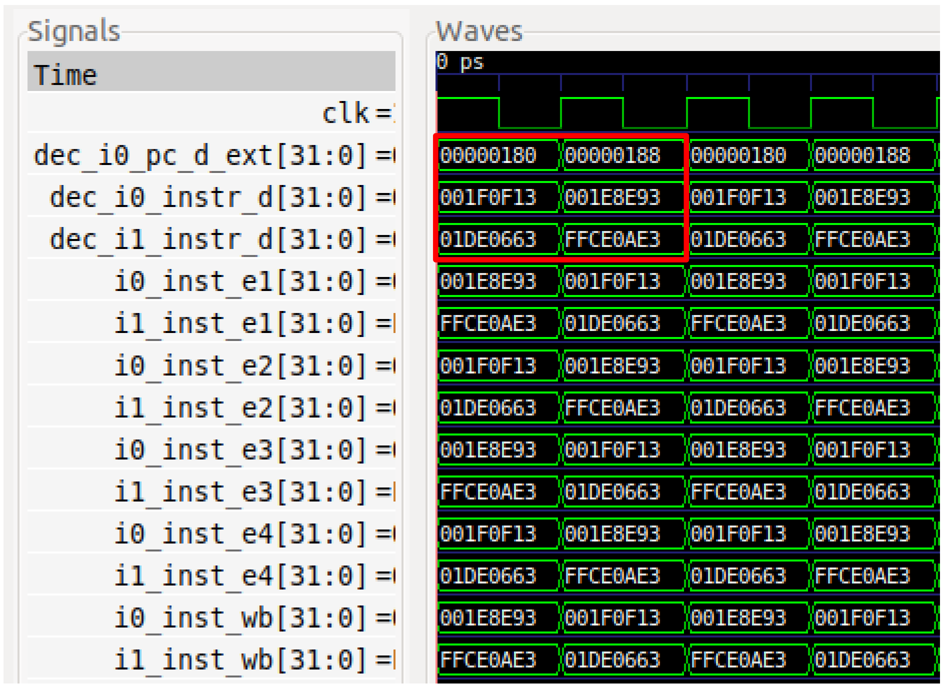


```
Test_Assembly:

li t2, 0x008                    # Disable Branch Predictor
csrrs t1, 0x7F9, t2
//INSERT_NOPS_2

li t3, 0xFFFF
li t4, 0x1
li t5, 0x0
li t6, 0x0

LOOP:
    add t5, t5, 1
    beq t3, t4, OUT
    add t4, t4, 1
    beq t3, t3, LOOP
OUT:
INSERT_NOPS_8
```

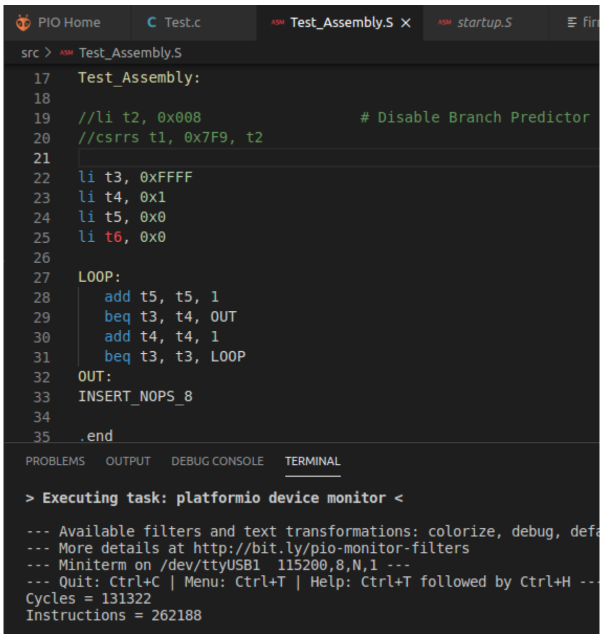
```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, d
--- 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 = 393468
Instructions = 262190
```

IPC = 262 / 393 = 0.67

Gshare Branch Predictor:



Signals				
dec_i0_pc_d_ext[31:0] =	00000180	00000188	00000180	00000188
dec_i0_instr_d[31:0] =	001F0F13	001E8E93	001F0F13	001E8E93
dec_i1_instr_d[31:0] =	01DE0663	FFCE0AE3	01DE0663	FFCE0AE3
i0_inst_e1[31:0] =	001E8E93	001F0F13	001E8E93	001F0F13
i1_inst_e1[31:0] =	FFCE0AE3	01DE0663	FFCE0AE3	01DE0663
i0_inst_e2[31:0] =	001F0F13	001E8E93	001F0F13	001E8E93
i1_inst_e2[31:0] =	01DE0663	FFCE0AE3	01DE0663	FFCE0AE3
i0_inst_e3[31:0] =	001E8E93	001F0F13	001E8E93	001F0F13
i1_inst_e3[31:0] =	FFCE0AE3	01DE0663	FFCE0AE3	01DE0663
i0_inst_e4[31:0] =	001F0F13	001E8E93	001F0F13	001E8E93
i1_inst_e4[31:0] =	01DE0663	FFCE0AE3	01DE0663	FFCE0AE3
i0_inst_wb[31:0] =	001E8E93	001F0F13	001E8E93	001F0F13
i1_inst_wb[31:0] =	FFCE0AE3	01DE0663	FFCE0AE3	01DE0663



```
Test_Assembly:

//li t2, 0x008                # Disable Branch Predictor
//csrrs t1, 0x7F9, t2

li t3, 0xFFFF
li t4, 0x1
li t5, 0x0
li t6, 0x0

LOOP:
    add t5, t5, 1
    beq t3, t4, OUT
    add t4, t4, 1
    beq t3, t3, LOOP
OUT:
INSERT_NOPS_8

.end
```

```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, defa
--- 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

The IPC is ideal when using the Gshare BP but it is far from ideal when using the Naïve BP due to the flush and redirect caused by the second branch instruction.

TASK: Analyse all these hashing modules and try to get an idea of how they work and how they are used in the Gshare BP structures.

Solution not provided.

TASK: Analyse how the access to these two structures is performed.

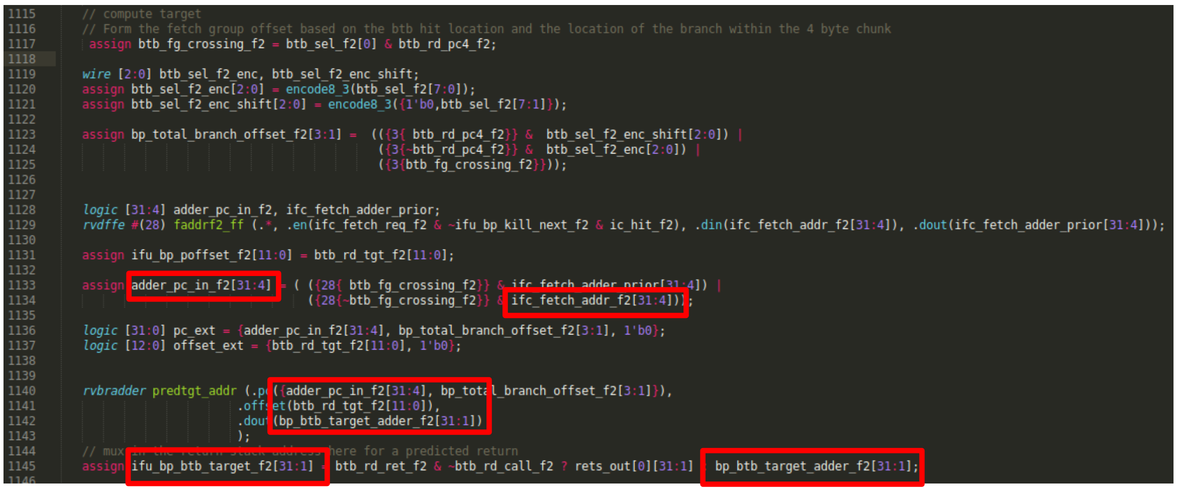
Solution not provided.

TASK: Analyse how select signal of the 5:1 multiplexer is computed.

Solution not provided.

TASK: Analyse how the predicted target address (`ifu_bp_btb_target_f2`) is obtained from the value read in the BTB (`btb_rd_tgt_f2[11:0]`) and the Fetch Address at FC2 (`ifc_fetch_addr_f2[31:4]`).

Module **ifu_bp_ctl**:



```
// compute target
// Form the fetch group offset based on the btb hit location and the location of the branch within the 4 byte chunk
 assign btb_fg_crossing_f2 = btb_sel_f2[0] & btb_rd_pc4_f2;

wire [2:0] btb_sel_f2_enc, btb_sel_f2_enc_shift;
assign btb_sel_f2_enc[2:0] = encode8_3(btb_sel_f2[7:0]);
assign btb_sel_f2_enc_shift[2:0] = encode8_3({1'b0,btb_sel_f2[7:1]});

assign bp_total_branch_offset_f2[3:1] =  (({3{ btb_rd_pc4_f2}} &  btb_sel_f2_enc_shift[2:0]) |
                                          ({3{~btb_rd_pc4_f2}} &  btb_sel_f2_enc[2:0]) |
                                          ({3{btb_fg_crossing_f2}}));


logic [31:4] adder_pc_in_f2, ifc_fetch_adder_prior;
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_adder_prior[31:4]));

assign ifu_bp_poffset_f2[11:0] = btb_rd_tgt_f2[11:0];

assign adder_pc_in_f2[31:4] = ( ({28{ btb_fg_crossing_f2}} & ifc_fetch_adder_prior[31:4]) |
                                ({28{~btb_fg_crossing_f2}} & ifc_fetch_addr_f2[31:4]));

logic [31:0] pc_ext = {adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1], 1'b0};
logic [12:0] offset_ext = {btb_rd_tgt_f2[11:0], 1'b0};


rvbradder predtgt_addr (.pc({adder_pc_in_f2[31:4], bp_total_branch_offset_f2[3:1]}),
                        .offset(btb_rd_tgt_f2[11:0]),
                        .dout(bp_btb_target_adder_f2[31:1])
                        );
// mux in the return stack address here for a predicted return
assign ifu_bp_btb_target_f2[31:1] = btb_rd_ret_f2 & ~btb_rd_call_f2 ? rets_out[0][31:1] : bp_btb_target_adder_f2[31:1];
```

<u>TASK</u>: Analyse the RAS implemented in the SweRV EH1 processor. An internet search will also give additional information about the operation of this structure (for example http://www-classes.usc.edu/engr/ee-s/457/EE457_Classnotes/ee457_Branch_Prediction/EE560_05_Ras_Just_FYI.pdf).

Solution not provided.

<u>TASK</u>: Analyse how the Global History Register is updated.

Solution not provided.

EXERCISES

1) Implement a Bimodal Branch Predictor and compare its performance with respect to the Gshare BP.

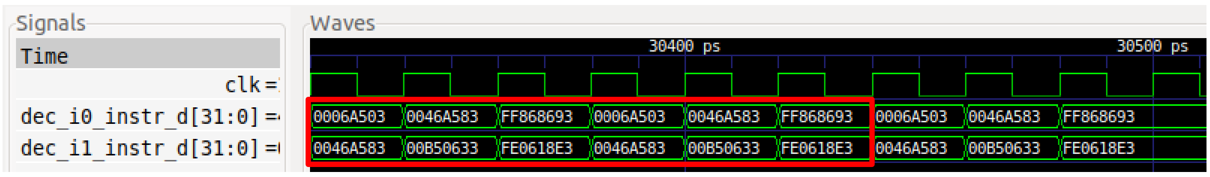
Solution not provided.

2) *(The following exercise is based on exercise 4.25 from the book "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy ([HePa]).)*
   Consider the following loop:

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

   Assume that perfect branch prediction is used (in the case of SweRV EH1, we can emulate this behaviour by simply avoiding the first iteration), that the pipeline has full forwarding support (again, this is the case in SweRV EH1), and that branches are resolved in the EX1 stage.

   a. Show a simulation for the second and third iterations of this loop. Explain the behaviour obtained. You can use the program provided at *[RVfpgaPath]/RVfpga/Labs/Lab16/HePa_Exercise-4-25*.



```
dec_i0_instr_d[31:0]  0006A503 0046A583 FF868693 0006A503 0046A583 FF868693 0006A503 0046A583 FF868693
dec_i1_instr_d[31:0]  0046A583 00B50633 FE0618E3 0046A583 00B50633 FE0618E3 0046A583 00B50633 FE0618E3
```