# TASKS

**TASK:** Examine the processor elements included in Figure 1 in the Verilog code and explain how they work.

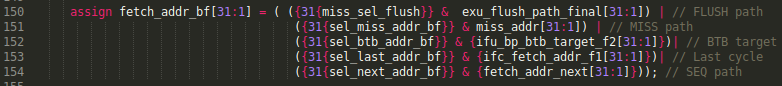
* The elements shown in the Decode stage (Register File, Instruction Register and Control Unit) can be found in modules **dec**, **dec\_decode\_ctl** and **dec\_gpr\_ctl**.
* The elements shown in the EX1 stage can be found in modules **exu** and **exu\_alu\_ctl**.
* The elements shown in the FC1 stage can be found in modules **ifu** and **ifu\_ifc\_ctl**.

**FC1 Stage:**

* 2:1 Multiplexer: Module **ifu\_ifc\_ctl**



* 5:1 Multiplexer: Module **ifu\_ifc\_ctl**



* Adder for sequential address: Module **ifu\_ifc\_ctl**



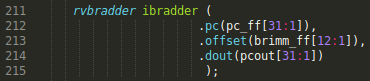
**EX1 Stage:**

* Comparator: Module **exu\_alu\_ctl**



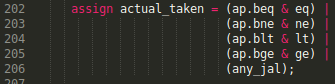
It compares the two operands:

* If they are equal: eq=1.
* If they are different: eq=0.
* Adder for the branch target address: Module **exu\_alu\_ctl**



It computes the addition of the PC and the offset.

* LOGIC: Module **exu\_alu\_ctl**



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



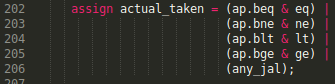
The branch has been mispredicted (cond\_mispredict=1) if it was predicted taken (ap.predict\_t = 1) and it must be not-taken (actual\_taken = 0), or if it was predicted not-taken (ap.predict\_nt = 1) and it must be taken (actual\_taken = 1)



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:** 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**



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



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)



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.

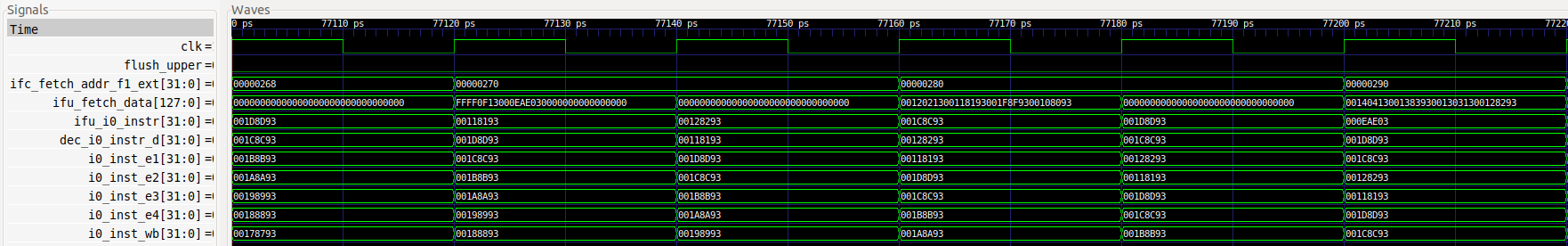
Also modify the program from Figure 2 to create new scenarios. For example, you can add some A-L instructions after the taken branch and see how they are flushed after the redirection.

**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

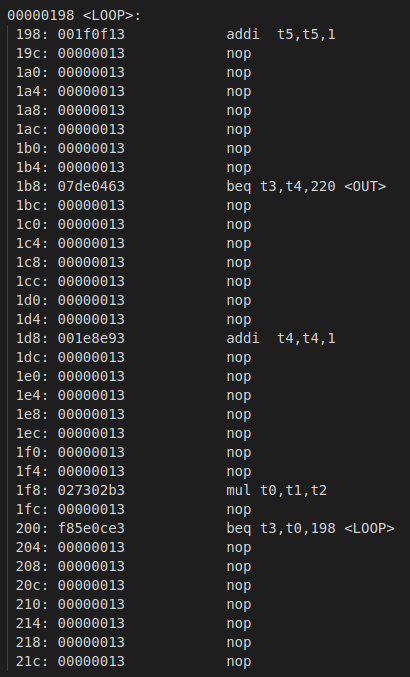
288: 00138393 addi t2,t2,1

28c: 00140413 addi s0,s0,1

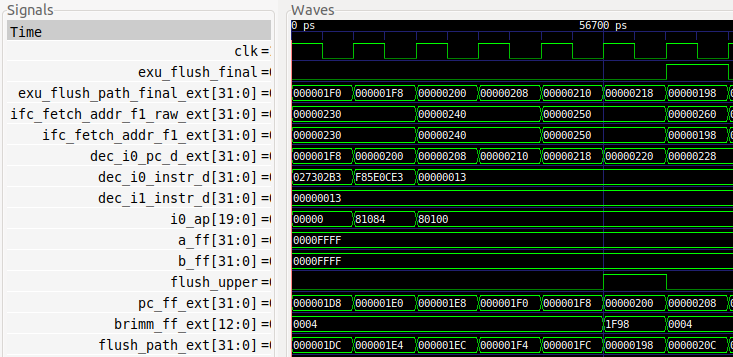
We can see that every two cycles a new 128-bit bundle is fetched.

**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:**



**Verilator Simulation:**



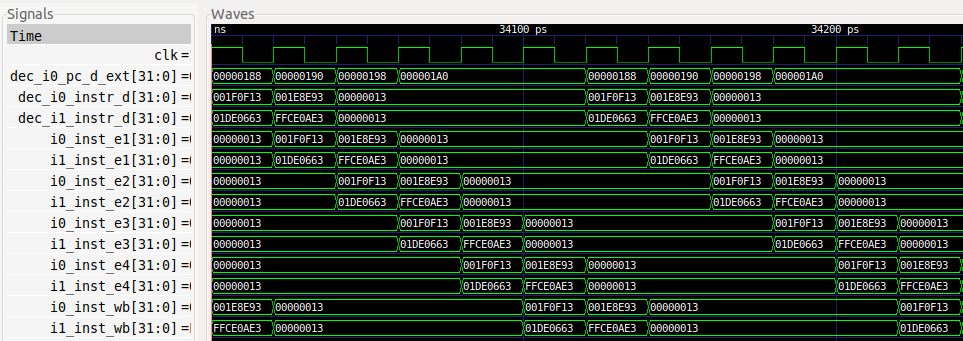
The beq instruction (0xf85e0ce3) is decoded, goes through EX1 (where it executes on the wrong operands), goes through EX2 and EX3, and then it goes through Commit where it executes again on the correct operands, triggering a flush and redirection (flush\_upper = exu\_flush\_final = 1).

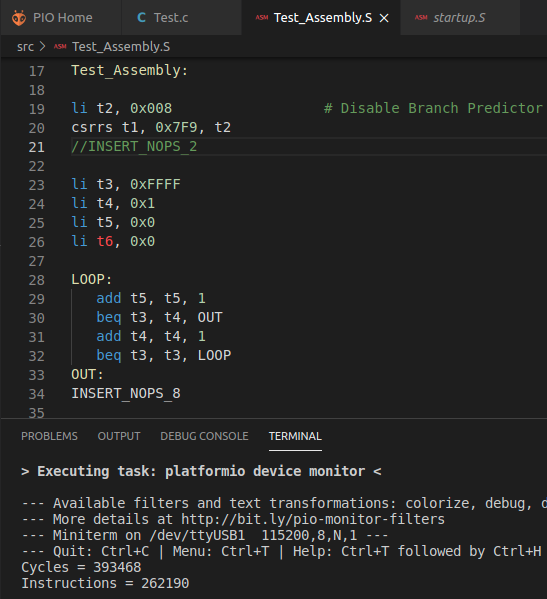
**TASK:** In the example from Figure 2, remove all the nop instructions and analyse the simulation. Then compute the IPC with the Performance Counters by executing the program on the board.

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.

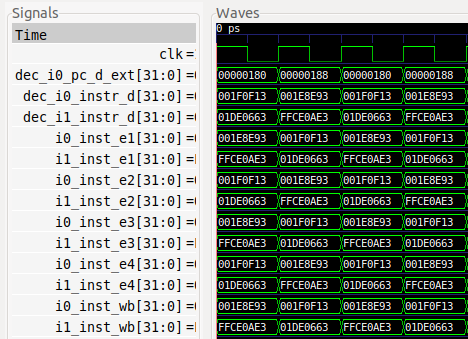
Naïve Branch Predictor:

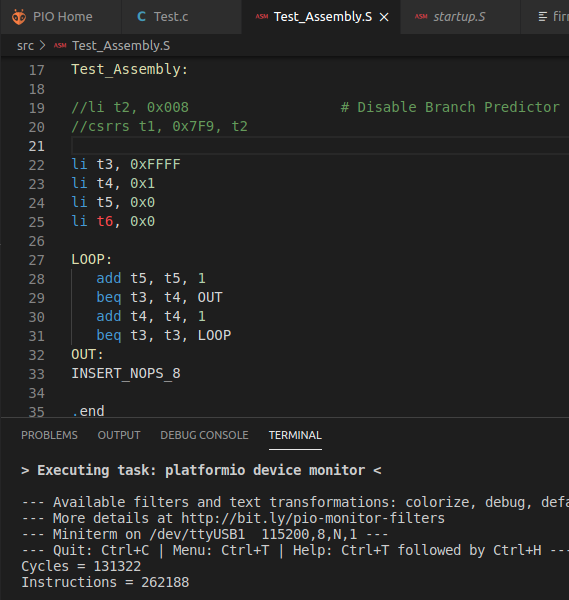




IPC = 262 / 393 = 0.67

Gshare Branch Predictor:





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.

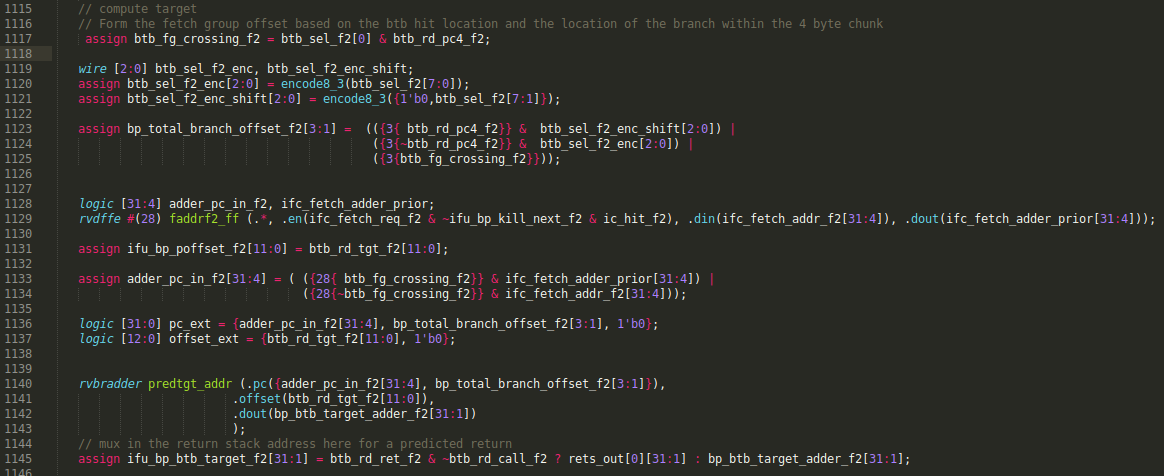
Solution not provided.

**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**:



**TASK**: 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.

**TASK**: 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.

1. (*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, LOOPAssume 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.

* 1. 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*.

