



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 16

Control Hazards: Branch Instructions

1. INTRODUCTION

In this lab, we complete our analysis of hazards. In the past two labs, we studied *structural* and *data hazards* on the SweRV EH1 processor, and we now focus on **control hazards**. As explained by S. Harris and D. Harris at “*Digital Design and Computer Architecture: RISC-V Edition*” (which we call DDCARV), a *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time it needs to be fetched.

NOTE: Before analysing the SweRV EH1 control hazard logic, we recommend reading how `beq` instructions are executed and how control hazards are resolved in the pipelined processor described in Section 7.5 of DDCARV. Specifically, control hazards are discussed in Section 7.5.3. We also recommend reading Section 7.7.3 about Branch Prediction before completing Section 3 of this lab.

Control hazards are caused by branch and jump instructions, because these instructions must calculate which instruction to fetch next. And, for branch instructions, they must also calculate whether the branch is taken or not. In contrast, for all other instructions, the instruction to fetch next is at $PC + 4$.

In some processors, control hazards never occur. For example, control hazards do not occur in processors where a given instruction executes completely before the next instruction is fetched. This is true for both the single- or multi-cycle processors in DDCARV. Specifically, because a branch instruction executes completely, the decisions about whether the branch is taken and what instruction to fetch next are resolved before the next instruction is fetched. In contrast, pipeline processors fetch the next instruction before those decisions are resolved.

One mechanism for dealing with control hazards is to stall the pipeline until the decision of what instruction to fetch after the branch has been made. Because this decision is made at the EX1 stage in SweRV EH1 (as we will see in Section 2), the pipeline would have to be stalled for four cycles at every branch (see Figure 1 in Lab 11 – which shows the pipeline). This would severely degrade the system performance if branches occur often, which is typically the case in real programs, thus this solution is not implemented in SweRV EH1.

An alternative is to predict whether the branch will be taken or not and begin fetching instructions from the predicted path. Once the branch decision is available, the processor can flush the fetched instructions if the prediction was wrong (in which case a branch misprediction penalty must be paid), or it can continue execution of the fetched instructions when the prediction was correct (in which case there is no performance loss). In SweRV EH1 two branch predictors (BPs) are available, which we analyse in this lab: a **naïve Branch Predictor**, which always predicts branches as not taken and thus offers a poor performance at no hardware cost, and a **Gshare Branch Predictor**, which offers higher performance at the cost of extra hardware.

In Section 2, we describe the execution of a `beq` instruction in SweRV and then perform some example simulations using the naïve BP (this is the typical scenario assumed in textbooks such as DDCARV). Then, in Section 3, we explain how control hazards can be handled more efficiently using the Gshare Branch Predictor that SweRV EH1 implements.

2. EXECUTION OF THE `beq` INSTRUCTION AND PC CALCULATION

In this section we analyse the execution of a `beq` instruction in SweRV EH1. First, in Section 2.A, we explain how `beq` instructions are executed in the EX1 stage and how the Fetch Address and the Next Fetch Address are computed in the FC1 stage (this completes the explanation of the FC1 stage that we started in Section 2.B.i of Lab 11). Although the figure included (Figure 1) and most of the descriptions are valid for any instruction, we focus on the execution of a `beq` instruction on a processor configuration that uses the naïve BP where branches are always predicted as *not taken* (as is done in DDCARV or in PaHe). Then, in Section 2.B, we perform some experiments to exemplify these concepts. Again, for these experiments, we disable the use of the Branch Predictor and instead use a *not taken* prediction for all conditional branches (i.e. what we have called naïve BP).

A. Theoretical explanation

Figure 1 shows the main structures in the FC1 stage that are used to determine the **Fetch Address** (which is the value in the Program Counter (PC), defined in DDCARV as a register that holds the memory address of the current instruction) and the **Next Fetch Address** (which is the value used to update the PC at the end of each cycle). The figure also shows the structures needed to execute a `beq` instruction in the EX1 stage (most of the hardware shown is also used in the execution of other branch instructions). As in other labs, the names of the signals used in the figure are the actual names used in the Verilog modules of the SweRV EH1 processor.

i. Fetch Address Computation

As shown in Figure 1, the FC1 stage includes two multiplexers: A 2:1 multiplexer that produces the Fetch Address in `ifc_fetch_addr_f1[31:1]`, and a 5:1 multiplexer that calculates the Next Fetch Address and puts it into signal `fetch_addr_bf[31:1]`.

- **2:1 multiplexer:** produces signal `ifc_fetch_address_f1[31:1]`, the memory address of the instruction fetched in the current cycle, which, as we analysed in Figure 3 of Lab 11, is provided to the Memory Controller for reading the 128-bit instruction bundle from the Instruction Cache. The two inputs to this multiplexer are:
 - o The branch target address (`exu_flush_path_final[31:1]`) computed at the EX1 stage as we will analyse below.
 - o The Next Fetch Address (`ifc_fetch_addr_f1_raw[31:1]`), computed and registered in the previous cycle as the output of the 5:1 multiplexer included in this stage and analysed below (`fetch_addr_bf[31:1]`).

The control signal of this multiplexer is called `exu_flush_final` and it is provided from the Execute stage. If fetching must occur from the branch target address, `exu_flush_final = 1` and `exu_flush_path_final[31:1]` is used as the Fetch Address; otherwise `exu_flush_final = 0` and `ifc_fetch_addr_f1_raw[31:1]` is used as the Fetch Address.

Note that an analogous 2:1 multiplexer is used in the processors explained in DDCARV for updating the PC in every cycle.

- **5:1 multiplexer:** produces signal `fetch_addr_bf[31:1]`, the address coming from one of the following five sources:

- The Fetch Address (`ifc_fetch_addr_f1`), which is used in some cases when the PC stays the same from one cycle to the next.
- The next sequential address (`fetch_addr_next`), which is computed as the Fetch Address (`ifc_fetch_addr_f1`) + 16, and which points to the next 128-bit bundle.
- The address predicted by the Branch Target Buffer (`ifc_bp_btb_target_f2`), which is one of the main structures of the Branch Predictor, and which is used as the Fetch Address when a branch is predicted to be taken.
- Two more input signals (`miss_addr` and `exu_flush_path_final`) which correspond to the *miss path* and the *flush path* respectively, but that we do not analyse in this lab.

The signal provided by this multiplexer (`fetch_addr_bf[31:1]`) is registered and used in the next cycle as an input to the 2:1 multiplexer analysed above.

Note that this 5:1 multiplexer does not exist in the processors from DDCARV, which have simpler designs.

ii. Execution of the `beq` Instruction

A conditional branch must calculate the branch target address and test if the condition is met. Specifically, in the case of SweRV EH1 (see Figure 1):

- **Branch Target Address computation**: a new adder is used in EX1 for computing the branch target address and placing it into signal `flush_path[31:1]`. This signal is provided as an input to the 2:1 multiplexer in FC1 (`exu_flush_path_final[31:1]`) through some logic and registers.
- **Condition resolution**: a new module is used in EX1, inside the `exu_alu_ctl` module, for checking if the two operands are equal (`eq = 1`) or not (`eq = 0`). Based on the `eq` signal (and some other signals, such as `ap.beq`, which you will analyse in a proposed task), signals `flush_upper` and `exu_flush_final` are computed and provided to the FC1 stage, where the latter is used as the control signal of the 2:1 multiplexer. This control signal (`exu_flush_final`) is 1 when the branch was mispredicted and 0 otherwise.

Specifically, in the case of a `beq` instruction and assuming the use of the naïve BP explained above where all branches are predicted as not taken, if the two operands of the branch are not equal then the branch must not be taken and the prediction is correct: `exu_flush_final = flush_upper = eq = 0`. In this case, the processor can continue fetching and executing instructions sequentially and there is no performance loss. We will analyse this situation in Section 2.B.i.

In contrast, if the two operands are equal, the branch must be taken and, in the case of the naïve BP that predicts not taken, a misprediction occurred: `exu_flush_final = flush_upper = eq = 1`. In this case, as we will explain in Section 2.B.ii, the following actions are triggered in the SweRV EH1 pipeline (see Figure 1).

- When `exu_flush_final = 1`, the instruction fetch is redirected to the target address of the branch, by selecting input 1 of the 2:1 multiplexer in FC1 (`ifc_fetch_addr_f1[31:1] = exu_flush_path_final[31:1]`), which contains the branch target address computed in the EX1 stage as explained above.
- The pipeline stages preceding EX1 are flushed. For that purpose, several signals (`exu_flush_final`, `exu_flush_upper_e2`, `exu_i0_flush_final` and `exu_i1_flush_final`) are provided to previous stages (the use of these signals is not specified in Figure 1).

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**.

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.

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.

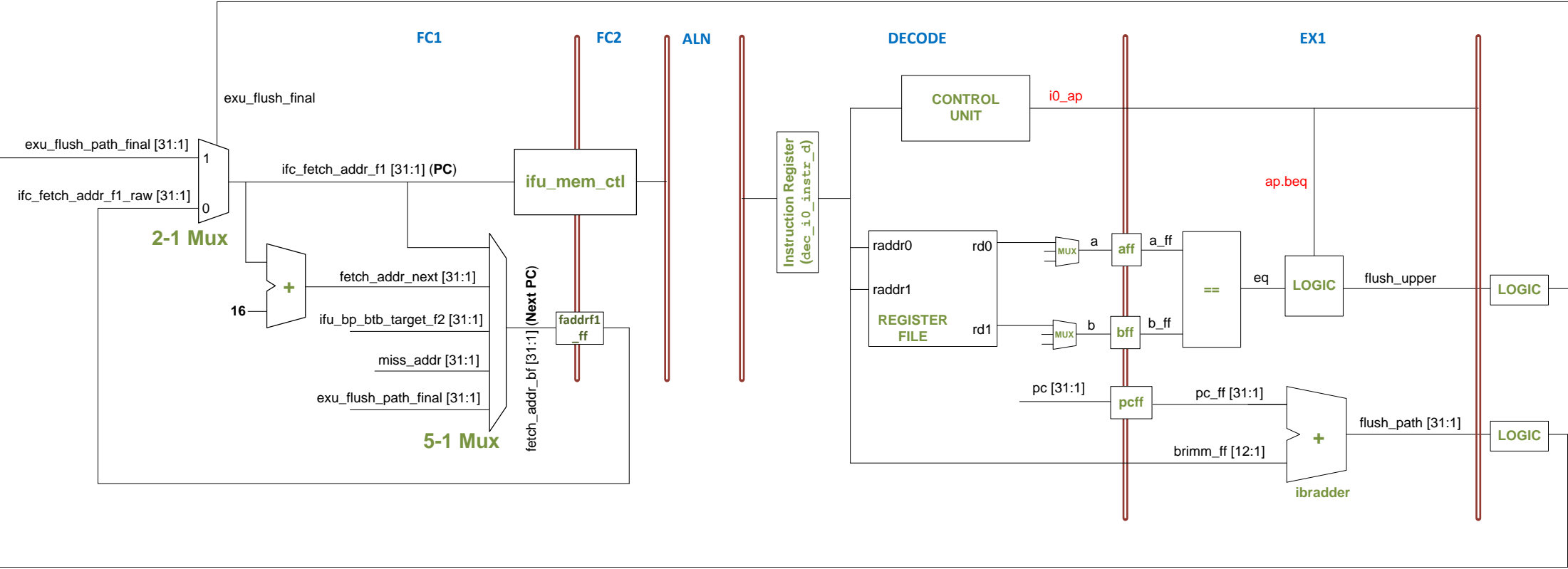


Figure 1. High-level view of the `beq` instruction executing through SweRV EH1

B. Experiments

Now that we have described the main concepts in the execution of a `beq` instruction in EX1 and the computation of the Fetch Address and Next Fetch Address in FC1, we now show some simulations to solidify these concepts.

Throughout this section we work with the example shown in Figure 2, which executes a loop that repeats for 0xFFFF iterations (i.e. 65,535 in decimal) and which contains two `beq` instructions: the first `beq` will always be *not taken* (except in the last iteration of the loop) and the second one will always be *taken*. As usual, the instructions that we want to analyse (in this case the `beq` instructions, highlighted in red) are surrounded by several nops in order to isolate them from preceding and subsequent instructions. Folder `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction` provides the PlatformIO project that you can analyse, simulate, and modify as desired.

```
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
    INSERT_NOPS_7
    beq t3, t4, OUT
    INSERT_NOPS_7
    add t4, t4, 1
    INSERT_NOPS_7
    beq t3, t3, LOOP
    INSERT_NOPS_7
OUT:
    INSERT_NOPS_8

.end
```

Figure 2. Program including `beq` instructions

In our experiments we disable the use of compressed instructions. Moreover, as we mentioned above, in this section the Gshare Branch Predictor available in SweRV EH1 is disabled and branches are always predicted to be *not taken* (naïve BP). This is done by including two instructions that allow the user to configure the processor during execution. As described in Appendix B of Lab 11, you must include the following two instructions in your code to disable the Branch Predictor and instead use a *not taken* prediction for every branch.

```
li t2, 0x008
csrrs t1, 0x7F9, t2
```

In this configuration, the first branch in the program (Figure 2) will always be correctly predicted (except in the last iteration of the loop, which we will not analyse here) and the second branch will always be mispredicted, which will cause a flush of the four preceding stages and an execution redirection. We will next analyse the execution of the two `beq` instructions.

i. Execution of the first branch: `beq t3, t4, OUT`

In this section we analyse the execution of the first branch instruction from Figure 2, which is always predicted correctly (except in the last iteration of the loop, which we don't analyse here). Open the project in PlatformIO, build it, and open the disassembly file (available at `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis`). Notice that the first `beq` instruction is placed at address `0x000001a8`:


```
0x000001a8:      07de0063      beq    t3, t4, 208 <OUT>
```

We next simulate the program from Figure 2 in Verilator as explained in the GSG and then open the trace file generated by the simulator on GTKWave. Figure 3 zooms into a random iteration of the loop (the first iteration should be avoided, as it contains I\$ misses which make it more difficult to analyse, as well as the last iteration, which misses the prediction) and focuses on the execution of the first `beq` instruction.

Most of the signals included in the figure are the ones that we showed in the diagram from Figure 1. However, you must take into account that those signals containing instruction addresses (marked with a suffix `_ext`) have been extended for the simulation with 1 bit to the right equal to 0 for the sake of clarity (note that the original non-extended signals in the Verilog code do not include the least significant bit as it is always 0); specifically:

Verilog code: <code>exu_flush_path_final[31:1]</code>	→	Simulation: <code>exu_flush_path_final_ext[31:0]</code>
Verilog code: <code>ifc_fetch_addr_f1_raw[31:1]</code>	→	Simulation: <code>ifc_fetch_addr_f1_raw_ext[31:0]</code>
Verilog code: <code>ifc_fetch_addr_f1[31:1]</code>	→	Simulation: <code>ifc_fetch_addr_f1_ext[31:0]</code>
Verilog code: <code>pc_ff[31:1]</code>	→	Simulation: <code>pc_ff_ext[31:0]</code>
Verilog code: <code>brim_ff[12:1]</code>	→	Simulation: <code>brim_ff_ext[12:0]</code>
Verilog code: <code>flush_path[31:1]</code>	→	Simulation: <code>flush_path_ext[31:0]</code>

File `test_1.tcl` is provided with the project. To use it in GTKWave, click on *File* → *Read Tcl Script File*, and open the `[RVfpgaPath]/RVfpga/Labs/Lab13/BEQ_Instruction/test_1.tcl` file.

Then, click on *Zoom In* () several times and move to any iteration of the loop, except the first or the last one. You will see the execution of the two `beq` instructions; Figure 3 shows what you should observe for the first branch instruction.

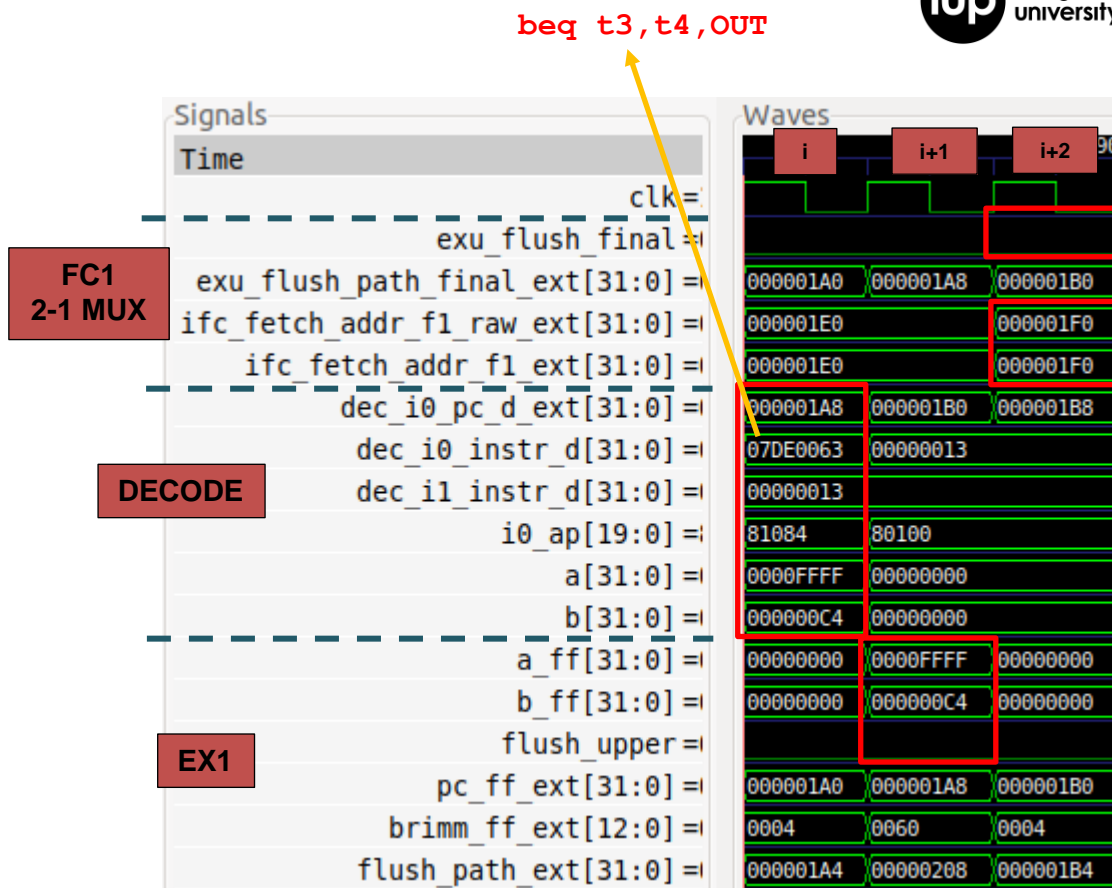


Figure 3. Verilator simulation for the execution of the first `beq` in Figure 2

Analyse the waveform from Figure 3 and the diagram from Figure 1 at the same time. Figure 3 shows three consecutive cycles: Decode of the `beq` (cycle i), EX1 of the `beq` (cycle $i+1$), and selection of the next PC at FC1 after resolving the `beq` (cycle $i+2$).

- **Cycle i - Decode stage for the `beq` instruction:** Signal `dec_i0_pc_d_ext` contains the address of the instruction in the Decode stage (in Way 0), which for the first `beq` is 0x000001A8, and signal `dec_i0_instr_d` (usually called the Instruction Register (IR) in textbooks) contains the 32-bit machine instruction, which for the first `beq` is 0x07DE0063 (in binary: 0000 0111 1101 1110 0000 0000 0110 0011).

In RISC-V, the opcode for the `beq` instruction is (see Appendix B of [DDCARV]):

`imm12,10:5 | rs2 | rs1 | 000 | imm4:1,11 | 1100011`

so you can verify that 0x07DE0063 corresponds to: `beq t3,t4,OUT` (`imm12:0 = 0x060`). Recall that the immediate gives the offset from the current PC of the target address. The target address (indicated by label "OUT:") is 24 instructions (i.e., 7 nops + 1 add + 7 nops + 1 `beq` + 7 nops + 1 nop = 24 instructions) past the current PC (i.e., `beq t3, t4, OUT`). This is $24 \times 4 = 96$ (0x60) bytes past the current PC.

During this stage, the **pipeline control signals are generated**. For the first `beq` instruction, the following bits of `i0_ap` (which for this instruction is equal to 0x81084 – see `SweRVref.docx`) are set:

- o `valid`: indicates it is a valid instruction that uses the ALU.
- o `beq`: indicates it is a *branch if equal* instruction.
- o `sub`: indicates that the ALU must perform subtraction. Some branch instructions use the result of the subtraction for computing the

comparison (however, this is not the case for `beq`, as we will show).

- o `predict_nt`: indicates that the branch is predicted as *not taken*.

Moreover, the **Register File is read** and the **branch instruction is routed to the I0 Pipe**. Signals `a` and `b` (0xFFFF and 0xC4, respectively, in this example) contain the inputs to the comparator used in the next stage, which in this case coincide with the values read from the Register File (in other cases, the operands could be provided through forwarding, as analysed in Lab 15).

- **Cycle *i+1* - EX1 stage for the `beq` instruction:** In the next cycle, the `beq` instruction is **executed**. Signals `a_ff` and `b_ff` are compared. Given that the two numbers (0xFFFF and 0xC4) are different the branch is not taken. As described before, in this configuration all branches are predicted *not taken* (`i0_ap.predict_nt = 1`). Thus, the branch has not been mispredicted (`flush_upper = 0`) and execution can continue as it is.
- **Cycle *i+2* - FC1 stage:** In the next cycle, given that the branch was predicted and resolved as not taken, fetching simply continues sequentially. In
- Figure 3, notice that `exu_flush_final = 0` and `ifc_fetch_addr_fl_ext[31:0] = ifc_fetch_addr_fl_raw_ext[31:0] = 0x000001F0`. This address points to the next sequential 128-bit bundle of instructions. You can see that in the previous two cycles the previous 128-bit bundle of instructions was fetched (`ifc_fetch_addr_fl_ext[31:0] = 0x000001E0`).

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.

ii. Execution of the second branch: `beq t3, t3, OUT`

Now we analyse the second branch, which is always taken but mispredicted as not taken. Open the disassembly file (available at `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis`). Notice that the second `beq` instruction is placed at address 0x000001E8:

```
0x000001e8:      fbce00e3      beq    t3,t3,188 <LOOP>
```

Figure 4 shows signals during a random iteration of the loop (but not the first iteration – which we avoid due to instruction cache (I\$) misses).

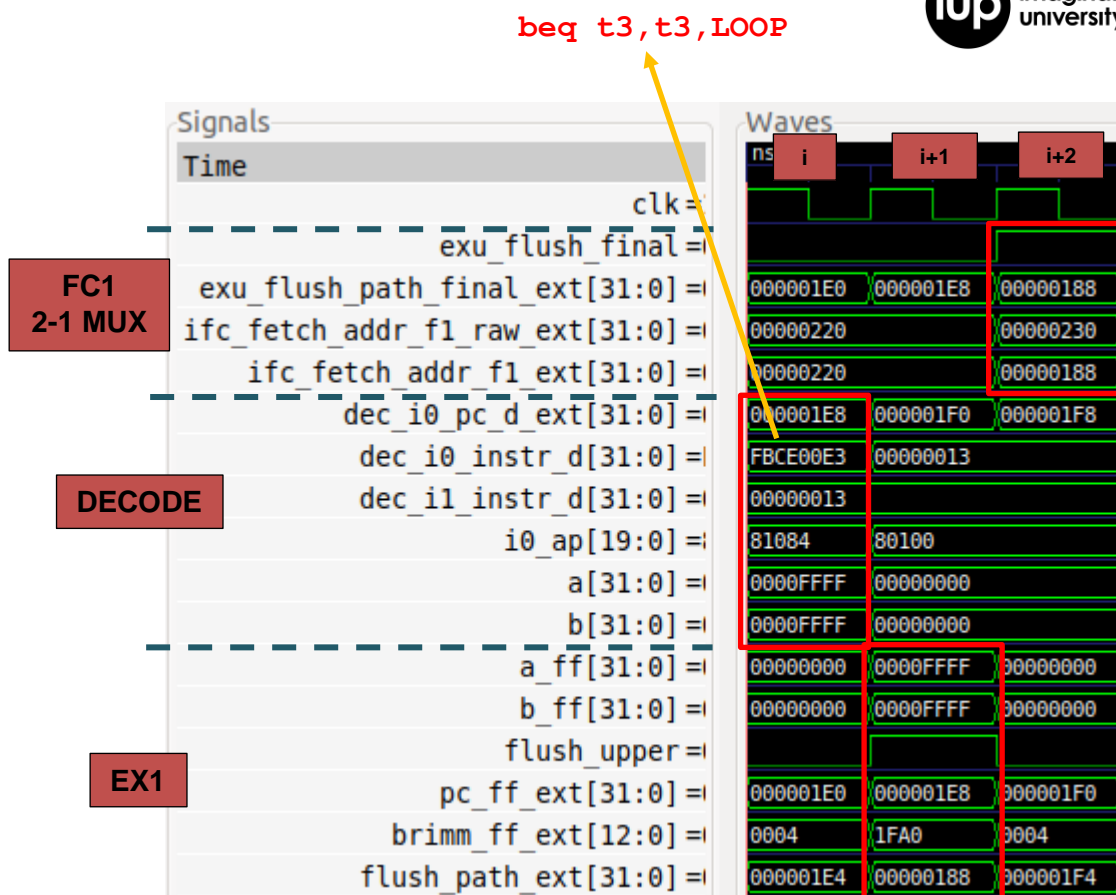


Figure 4. Verilator simulation for the second branch in the example from Figure 2

Analyse the waveform from Figure 4 and the diagram from Figure 1 at the same time. The values highlighted in red show three consecutive cycles during execution of the second `beq` instruction: Decode of the `beq` (cycle i), EX1 of the `beq` (cycle $i+1$), and selection of the next PC at FC1 after resolving the `beq` (cycle $i+2$).

- **Cycle i - Decode stage for the `beq` instruction:** The PC (signal `dec_i0_pc_d_ext`) is 0x000001E8, and the instruction (signal `dec_i0_instr_d`) is 0xFBCE00E3 (in binary: 1111 1011 1100 1110 0000 0000 1110 0011).

In RISC-V, the opcode for the `beq` instruction is (see Appendix B of [DDCARV]):

```
imm12,10:5 | rs2 | rs1 | 000 | imm4:1,11 | 1100011
```

So you can verify that 0xFBCE00E3 corresponds to: `beq t3, t3, LOOP` (Immediate_{12:0} = 0x1FA0). Recall that the immediate gives the offset from the current PC of the target address. The target address (indicated by label "LOOP:") is 24 instructions (i.e., 7 nops + 1 add + 7 nops + 1 beq + 7 nops + 1 add = 24 instructions) *before* the current PC (i.e., `beq t3, t3, LOOP`). This is $24 \times 4 = 96$ bytes before the current PC. So, the immediate encodes -96, which is 0x1FA0, written in 13-bit two's complement representation.

During this stage the **pipeline control signals are generated**. For this `beq` instruction, the control signals are the same as for the first `beq` (see previous section).

Moreover, the **Register File is read** and the **branch instruction is routed to the I0 Pipe**. Signals `a` and `b` (0xFFFF for both of them) contain the inputs to the comparator

used in the next stage, which in this case are the values read from the Register File.

- **Cycle $i+1$ - EX1 stage for the `beq` instruction:** In the next cycle, the `beq` instruction is **executed**. On the one side, signals `a_ff` and `b_ff` are compared. Given that the two values are equal the branch must be taken. However, as explained before, in our configuration all branches are predicted *not taken* (`i0_ap.predict_nt = 1`). Thus, the branch has been mispredicted (`flush_upper = 1`). So instructions must be fetched from the branch target address, and the initial pipeline stages must be flushed.

In this stage, the target address is computed as the addition between `pc_ff_ext` (0x1E8) and `brim_ff_ext` (0x1FA0). The result is placed into signal `flush_path_ext` (0x00000188).

- **Cycle $i+2$ - FC1 stage:** In the next cycle, execution must continue at the branch target address. In Figure 4 you can see that `exu_flush_final = 1` and `ifc_fetch_addr_f1_ext = exu_flush_path_final_ext = 0x00000188`. This address corresponds to the branch target address, which is the address of the first instruction of the loop (note that this is a backward branch).

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.

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.

3. The Gshare Branch Predictor used by SweRV EH1

In Section 2, we discussed the SweRV EH1 configuration that includes only a naïve branch predictor that always predicts not taken, but in this section we analyse the operation of the Gshare Branch Predictor available in SweRV EH1. The Gshare BP performs a more intelligent prediction for each branch instruction, which improves performance but requires extra hardware. Before we describe how the Gshare BP works in SweRV EH1, we compare the performance of the two BPs.

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.

NOTE: A classic paper published by Scott McFarling in 1993 is called “Combining Branch Predictors” (<https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>). It describes, in Section 7, the operation of the Gshare branch predictor. You can also search for other documents, such as <https://people.engr.ncsu.edu/efg/521/f02/common/lectures/notes/lec16.pdf>. We recommend reading them to understand how the Gshare BP works before beginning this section.

Figure 5 shows a simplified view of the Gshare BP available in SweRV EH1. All the BP structures are implemented inside module `ifu_bp_ctl` (in file [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_bp_ctl.sv). The structures related to the Gshare BP are surrounded by a blue square in the figure.

This BP is made up by the Branch History Table (BHT), which predicts the direction of the branch (*taken* or *not taken*), and the Branch Target Buffer (BTB), which predicts the target address in the case of taken branches. In our default configuration, the BHT contains 128 2-bit entries. You can find it in lines 1615-1705 of module `ifu_bp_ctl`. In our default configuration, the BTB contains 32 13-bit entries. You can find it in lines 1439-1613 of module `ifu_bp_ctl`.

To make a branch prediction, the following occurs in every cycle (see Figure 5):

1. The Fetch Address (`ifc_fetch_addr_f1 [31:1]`) and some other signals are passed through several hashing modules inside module `ifu_bp_ctl`: `f1hash`, `rdtagf1`, `fghrhs`... All these hashing modules are implemented in file [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/beh/beh_lib.sv, using some of the macros defined at [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh.

As an example, you can see that the `fghrhs` module receives signal

btb_rd_addr_f1, which comes from a hashing of the Fetch Address
 (f1hash(.pc(ifc_fetch_addr_f1[31:1]),
 .hash(btb_rd_addr_f1[`RV_BTБ_ADDR_HI:`RV_BTБ_ADDR_LO])) and fgfr_ns
 (which is the Global History Register), and outputs signal
 bht_rd_addr_hashed_f1.

```
rvbttb_gfr_hash fgfrns (.hashin(bttb_rd_addr_f1[`RV_BTБ_ADDR_HI:`RV_BTБ_ADDR_LO]), .gfr(fgfr_ns[`RV_BHT_GFR_RANGE]), .hash(bht_rd_addr_hashed_f1[`RV_BHT_ADDR_HI:`RV_BHT_ADDR_LO]));
```

This signal is used to access the BHT table of the Gshare Branch Predictor.

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.

2. All these hashed signals (btb_rd_addr_f1, bht_rd_addr_hashed_f1, fetch_rd_tag_f1, etc.) are used for accessing the two main structures that make up the Gshare BP: the BHT and the BTB.

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

3. As a result of the access to the BHT, a direction prediction is obtained in signal ifu_bp_kill_next_f2, which is 0 if the branch is predicted *not taken* and 1 if it is predicted *taken*. This signal is used, in addition with other signals that we do not describe here, to compute the control signal of the 5:1 multiplexer from FC1.

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

4. As a result of the access to the BTB, the predicted target address for taken branches is obtained from an adder in signal ifu_bp_btb_target_f2 [31:1]. (Note that the predicted address can also come from the Return Address Stack (RAS) in case a *ret* instruction is predicted.) This signal is one of the inputs of the 5:1 multiplexer from FC1.

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]).

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).


5. In the 5:1 multiplexer from FC1, if ifu_bp_kill_next_f2 = 1, then the predicted target address is used as the Next Fetch Address: fetch_addr_bf [31:1] = ifu_bp_btb_target_f2 [31:1] (unless the pipeline is being flushed). Instead, if ifu_bp_kill_next_f2 = 0, one of the other four inputs are used as the Next Fetch Address.

Throughout this section, we continue working with the example code in Figure 2. The only difference in this section is that we enable the Gshare Branch Predictor by substituting the two instructions that disable the Gshare BP with two nop instructions (the reason for inserting two nops is to maintain the same instruction addresses as in the previous section).

We now analyse the execution of the second branch instruction in the program, as done in Section 2.B.ii. Remember that the second `beq` instruction is placed at address 0x000001E8 in our program, which means that it is contained within the 128-bit bundle mapped in the address range 0x1E0-0x1EF:

```
0x000001e8:      fbce00e3          beq    t3,t3,188 <LOOP>
```

Figure 6 zooms into a random iteration of the loop. As usual, the first iteration is avoided, as it contains I\$ misses – additionally, the branch prediction misses for this branch instruction in its first iteration. Most of the signals included in the figure are the ones that we showed in Figure 5. File `test_1_BP.tcl` is provided with the project. For using it in GTKWave, click on *File* → *Read Tcl Script File*, and open the `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/test_1_BP.tcl` file. Then, click on *Zoom In*

() several times and move to any iteration of the loop, except the first one. You will see the execution of the two `beq` instructions; Figure 6 shows what you should observe for the second branch instruction.

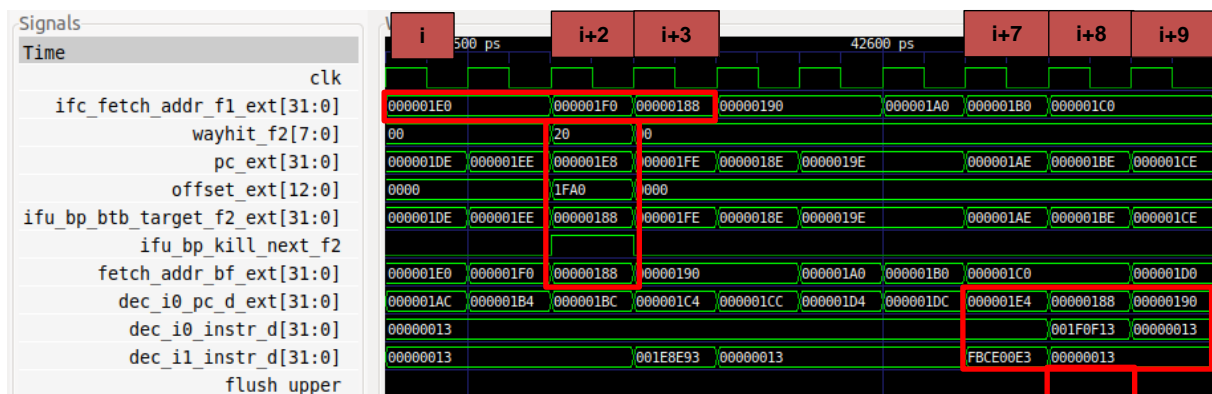


Figure 6. Verilator simulation for the example from Figure 2

Analyse the waveform from Figure 6 and the diagram from Figure 5 at the same time. The values highlighted in red correspond to the second `beq` instruction as it traverses the pipeline stages.

- **Cycle *i*:** The address of the bundle that contains the second branch is provided to the Instruction Cache: `ifc_fetch_addr_f1_ext` = 0x000001E0. The Branch Target Buffer (BTB) is read using this address.
- **Cycle *i+2*:** A hit takes place in the BTB: `wayhit_f2` = 0x20 (this signal, which is not included in Figure 5, indicates a hit when it is non-zero). The address of the branch (`pc_ext` = 0x000001E8) is added to the offset provided by the BTB (`offset_ext` = 0x1FA0, which is a negative value), which results in the predicted target address (`ifu_bp_btb_target_f2_ext` = 0x00000188). Given that the branch is predicted taken by the BHT (`ifu_bp_kill_next_f2` = 1), it is used as the Next Fetch PC (`fetch_addr_bf_ext` = 0x00000188).

- **Cycle $i+3$:** The Fetch Address is the predicted target address of the branch, which was computed in the previous cycle: `ifc_fetch_addr_f1_ext = 0x00000188`.
- **Cycle $i+7$:** The branch is decoded in Way 1 (`dec_i1_instr_d = 0xFBCE00E3`).
- **Cycle $i+8$:** The branch executes. The prediction was correct, so no flush needs to be triggered (`flush_upper = 0`).
- **Cycle $i+9$:** Execution continues normally through the branch target address given that the prediction was correct.

TASK: Explain how the Global History Register is updated at module `ifu_bp_ctl`.

4. EXERCISES

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

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](#).