THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 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 VeeR EL2 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 VeeR EL2 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-cycle and multicycle 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, pipelined 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 D Stage in VeeR EL2, the pipeline would have to be stalled for one cycle at every branch. This would degrade the system performance if branches occur often, which is typically the case in real programs, thus this solution is not implemented in VeeR EL2.

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 VeeR EL2 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 VeeR 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 propose some exercises where you will analyse more high-performance branch predictors including the Gshare BP.

2. Execution of the `beq` Instruction and PC Calculation

In this section we analyse the execution of a `beq` instruction in VeeR EL2. First, in Section 2.A, we explain how `beq` instructions are executed in the D stage and how the Fetch Address and the Next Fetch Address are computed prior to the F Stage, in what is called the BF Stage. 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 and 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 BF 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 the current cycle). The figure also shows the structures needed to execute a `beq` instruction in the D 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 VeeR EL2 processor.

i. Fetch Address Computation

As shown in Figure 1, the BF Stage includes a 4:1 multiplexer that calculates the **Next Fetch Address** as signal `fetch_addr_bf[31:1]`. This address, after being renamed to signal `ifc_fetch_addr_bf[31:1]`, is provided to the Memory Controller to access the ICCM and the I$ to fetch the next instruction that must be executed by the processor. The inputs to this 4:1 multiplexer are the following:

- The address computed by a branch/jump instruction in the D Stage (`exu_flush_path_final`), and which is used when the path predicted earlier was wrong in order to redirect the execution through the correct path.
- The address predicted by the Branch Target Buffer (`ifc_bp_btb_target_f`), which is one of the main structures of the Branch Predictor, and which is used as the Next Fetch Address when a branch is predicted to be taken.
- The next sequential address (`fetch_addr_next`), which is computed as the Fetch Address (`ifc_fetch_addr_f`) + 4, and which points to the next sequential instruction.
- The Fetch Address (`ifc_fetch_addr_f`), which is used when the PC stays the same from one cycle to the next.

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 VeeR EL2 (see Figure 1):

- **Branch Target Address computation**: a new adder is used in the D Stage for computing the branch target address and placing it into signal

`exu_flush_path_final[31:1]`.

- **Condition resolution**: a new module is used in the D Stage, inside the **`el2_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 the control packet `ap`), signal `exu_flush_final` is computed and provided to the BF stage, where it is used, combined with other signals, as the control signal of the 4:1 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` = 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` = 1. In this case, as we will explain in Section 2.B.ii, the following actions are triggered in the VeeR EL2 pipeline (see Figure 1).

- When `exu_flush_final` = 1, the instruction fetch is redirected to the target address of the branch (i.e. `fetch_addr_bf[31:1]` = `exu_flush_path_final[31:1]`), which contains the branch target address computed in the D Stage as explained above.

- The pipeline stages preceding the D Stage are flushed.

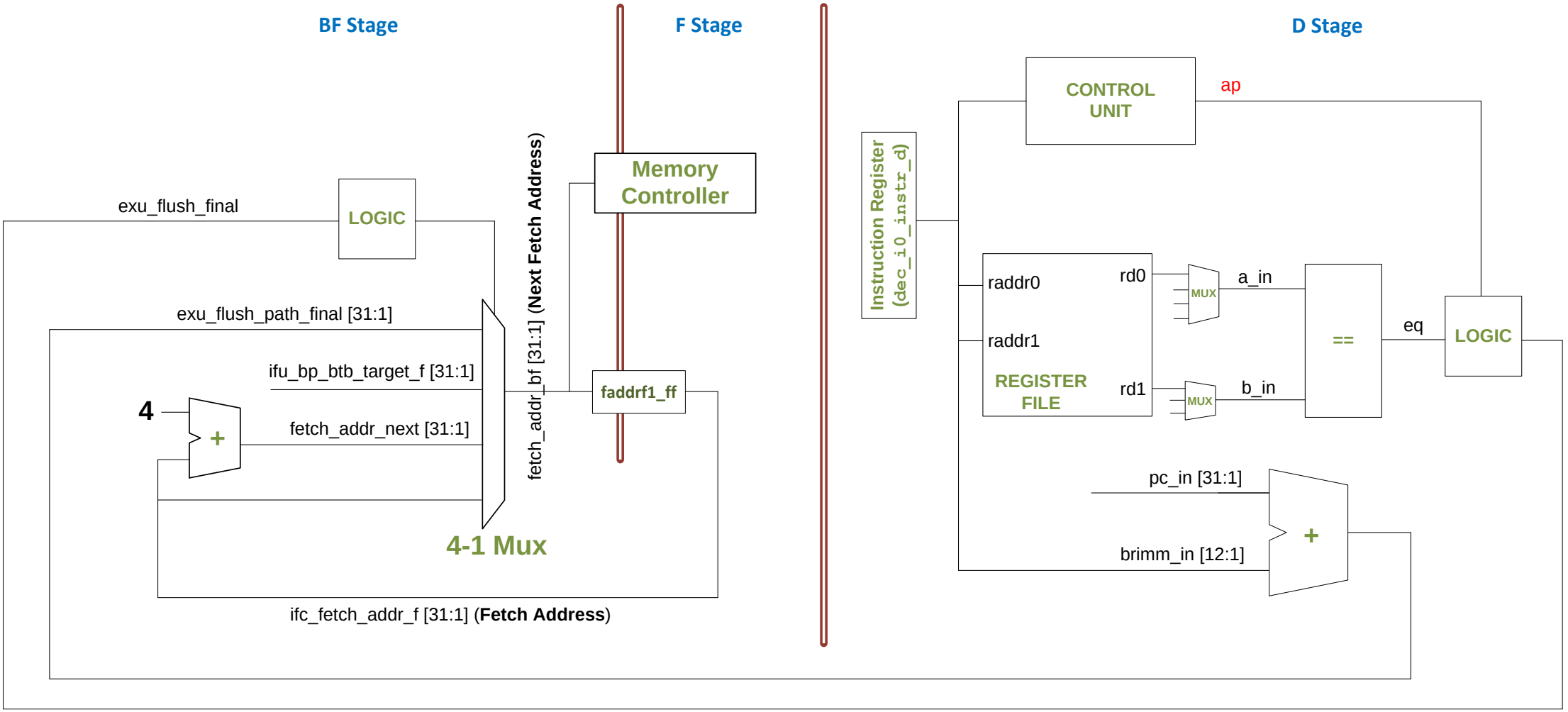

Figure 1. High-level view of the `beq` instruction executing through VeeR EL2

B. Experiments

Now that we have described the main concepts in the execution of a `beq` instruction in the D Stage and the computation of the Fetch Address and Next Fetch Address in the BF Stage and the F Stage, 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 (i.e. 65,535) iterations 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 *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab16/BEQ_Instruction* provides the Catapult 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_2
   beq t3, t4, OUT
   INSERT_NOPS_2
   add t4, t4, 1
   INSERT_NOPS_2
   beq t3, t3, LOOP
   INSERT_NOPS_2

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 VeeR EL2 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 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 avoid here). Open the project in Catapult, build it, and open the disassembly file. Notice that the first `beq` instruction is located at address 0x00000438:

```
0x00000438:      03de0263                beq   t3,t4,45c <OUT>
```

We next simulate the program from Figure 2 in RVfpgaEL2-Trace 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 significate bit as it is always 0); specifically:

Verilog code: `exu_flush_path_final[31:1]`	→	Simulation: `exu_flush_path_final_ext[31:0]`
Verilog code: `ifu_bp_btb_target_f[31:1]`	→	Simulation: `ifu_bp_btb_target_f_ext[31:0]`
Verilog code: `fetch_addr_next[31:1]`	→	Simulation: `fetch_addr_next_ext[31:0]`
Verilog code: `ifc_fetch_addr_f[31:1]`	→	Simulation: `ifc_fetch_addr_f_ext[31:0]`
Verilog code: `ifc_fetch_addr_bf[31:1]`	→	Simulation: `ifc_fetch_addr_bf_ext[31:0]`
Verilog code: `pc_in[31:1]`	→	Simulation: `pc_in_ext_ext[31:0]`
Verilog code: `brimm_in[12:1]`	→	Simulation: `brimm_in_ext[12:0]`

File *test_1.tcl* is provided with the project. To use it in GTKWave, click on *File → Read Tcl Script File*, and open the *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab13/BEQ_Instruction/test_1.tcl* file. Then, 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.

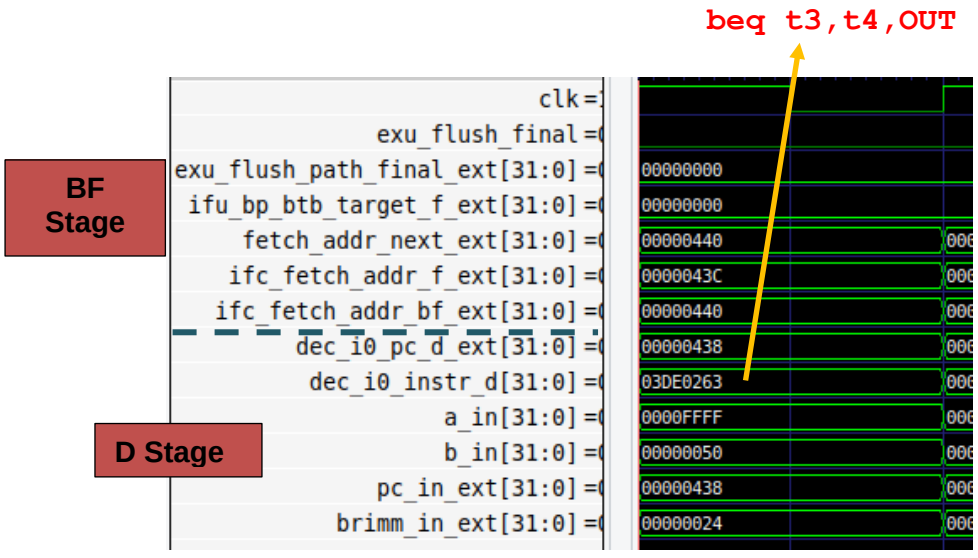

Figure 3. Verilator simulation for the execution of the first `beq` in Figure 2

Analyse the waveform from Figure 3 and the diagram from Figure 1 at the same time. Figure 3 shows the cycle when the first `beq` is at the D Stage.

- In this cycle, 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 0x03DE0263 (in binary: 0000 0011 1101 1110 0000 0010 0110 0011).

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

  $imm_{12,10:5}$ | rs2 | rs1 | 000 | $imm_{4:1,11}$ | 1100011

  so you can verify that 0x03DE0263 corresponds to: `beq t3,t4,OUT (`$imm_{12:0}$ `= 0x24)`. Recall that the immediate gives the offset from the current PC of the target address. The target address (indicated by label "OUT:") is 9 instructions past the current PC (specifically: 1 beq + 2 nops + 1 add + 2 nops + 1 beq + 2 nops = 9 instructions). This is 9*4 = 36 bytes past the current PC.

- During this stage, the **Register File is read** and **the branch instruction is executed**. Signals `a_in` and `b_in` (0xFFFF and 0x50, respectively, in this example) contain the inputs to the comparator.

- Signals `a_in` and `b_in` are compared. Given that the two numbers (0xFFFF and 0x50) are different, the branch is not taken. In this configuration all branches are predicted *not taken*. Thus, the branch has been correctly predicted (`exu_flush_final` = 0) and execution can continue as it is.

- Given that the branch was both predicted and resolved as not taken, fetching simply continues sequentially. In
- Figure 3, notice that the **Next Fetch Address** is the sequential address: `ifc_fetch_addr_bf_ext[31:0]` = `fetch_addr_next_ext[31:0]` = 0x00000440. This address points to the next sequential instruction.

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. Notice that the second `beq` instruction is located at address 0x000001E8:

```
0x00000450:      fdce0ee3                beq   t3,t3,42c <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).

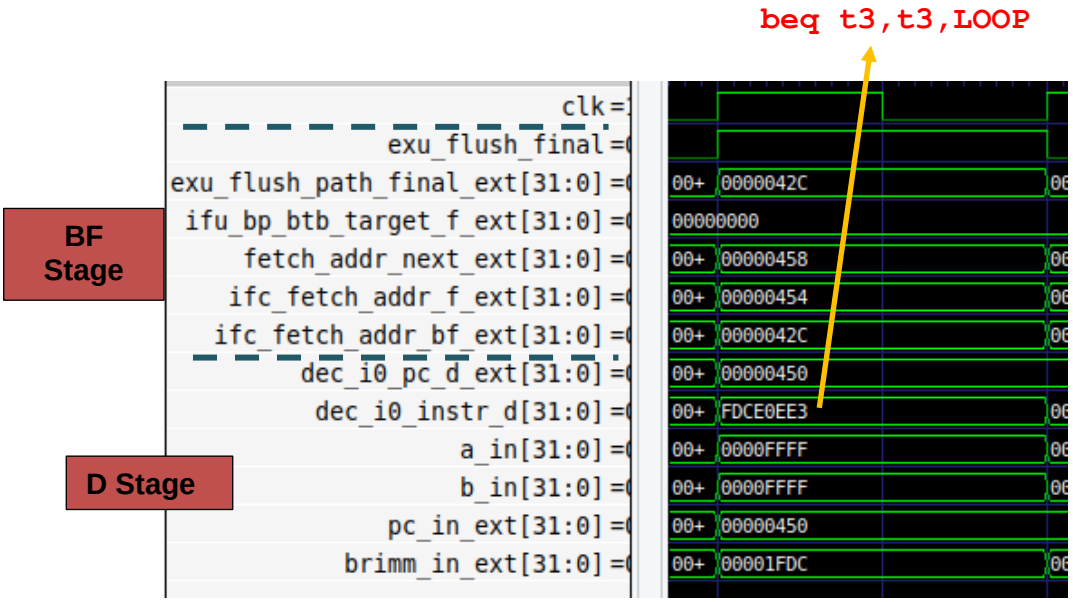

Figure 4. Verilator simulation for the second branch in the example from Figure 2

Analyse the waveform from Figure 4 and the diagram from Figure 1 at the same time. Figure 4 shows the cycle when the second `beq` is at the D Stage.

- In this cycle, signal `dec_i0_pc_d_ext` is 0x00000128, and the instruction (signal `dec_i0_instr_d`) is 0xFDCE0EE3 (in binary: 1111 1101 1100 1110 0000 1110 1110 0011). In RISC-V, the opcode for the `beq` instruction is (see Appendix B of [DDCARV]):

  $imm_{12,10:5}$ | rs2 | rs1 | 000 | $imm_{4:1,11}$ | 1100011

  So you can verify that 0xFDCE0EE3 corresponds to: `beq t3,t3,LOOP` ($Immediate_{12:0}$ = 0x1FDC). Recall that the immediate gives the offset from the current PC of the target address. The target address (indicated by label “LOOP:”) is 9 instructions (i.e., 2 nops + 1 add + 2 nops + 1 beq + 2 nops + 1 add = 9 instructions) *before* the current PC (i.e., `beq t3,t3,LOOP`). This is 9*4 = 36 bytes before the current PC. So, the immediate encodes -36, which is 0x1FDC, written in 13-bit two’s complement representation.

- During this stage, the **Register File is read** and **the branch instruction is executed**. Signals `a_in` and `b_in` (0xFFFF for both of them) contain the inputs to the comparator used by the ALU.

- Signals `a_in` and `b_in` 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*. Given that the branch has been mispredicted, instructions must be fetched from the branch target address and the initial pipeline stages must be flushed.

  The target address is computed as the addition between `pc_in_ext` (0x450) and `brim_in_ext` (0x1FDC). The result is placed into signal `exu_flush_path_final_ext` (0x0000042C).

- Given that the branch was mispredicted and resolved as taken, execution must be redirected to the branch target address. In Figure 4 you can see that `exu_flush_final` = 1 and the **Next Fetch Address** is `ifc_fetch_addr_bf_ext` = `exu_flush_path_final_ext` = 0x0000042C. This address corresponds to the first instruction of the loop (note that this is a backward branch).

3. Exercises

1) In Section 2, we discussed the VeeR EL2 configuration that includes a naïve branch predictor that always predicts branches as not taken. However, as we mentioned at the beginning of this lab, this processor includes a more efficient branch predictor called Gshare. Analyse the implementation of this Branch Predictor, both by analysing the Verilog code available in module **`el2_ifu_bp_ctl`** and by performing a simulation using RVfpgaEL2-Trace with the example used in Section 2 enabling the Gshare BP.

   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 (BP). 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.

2) In the examples shown in this lab we used the naïve Branch Predictor instead of the Gshare Branch Predictor. Test them again, but this time enablie the Gshare Branch Predictor (configure it through the CSR register as explained before) and analyse the differences in performance with respect to using the naïve one. Use the following RVfpgaEL2 tools:
   a. RVfpgaEL2-Trace: Remember that you must include the control instruction.
   b. RVfpgaEL2-NexysA7: Remember that you must uncomment the configuration of the performance counters in file Test.c and remove the nops in file Test_Assembly.S. If you don't have the physical board, simply skip this and test the program in simulation only.
   c. RVfpgaEL2-ViDBo: Remember that you must uncomment the configuration of the performance counters in file Test.c and remove the nops in file Test_Assembly.S.

3) In the examples shown in previous labs we used the naïve Branch Predictor instead of the Gshare Branch Predictor. Test those examples again, but this time enable the Gshare Branch Predictor (configure it through the CSR register as explained before) and analyse the differences in performance with respect to using the naïve one. Use the following RVfpgaEL2 tools: RVfpgaEL2-NexysA7 (if you have the physical board), RVfpgaEL2-ViDBo, and RVfpgaEL2-Trace.

4) Use a Bimodal Branch Predictor (you can implement it yourself or download one from the internet) and compare its performance to the Gshare BP.