



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 16

Conflitos de Controlo: Instruções de Branch

1. INTRODUÇÃO

Neste laboratório, concluímos a nossa análise dos conflitos (*hazards*). Nos dois laboratórios anteriores, estudámos os conflitos *estruturais* e *de dados* no processador SweRV EH1 e, agora, concentramo-nos nos **conflitos de controlo**. Como explicado por S. Harris e D. Harris em "*Digital Design and Computer Architecture: RISC-V Edition*" (a que chamamos DDCARV), um *conflito de controlo* ocorre quando a decisão sobre qual a instrução a ir buscar a seguir não foi tomada até ao momento em que tem de ser lida.

NOTA: Antes de analisar a lógica dos conflitos de controlo do SweRV EH1, recomendamos a leitura da forma como as instruções `beq` são executadas e como os conflitos de controlo são resolvidos no processador em pipeline descrito na Secção 7.5 do DDCARV. Especificamente, os conflitos de controlo são discutidos na Secção 7.5.3. Recomendamos também a leitura da Secção 7.7.3 sobre previsão de salto antes de completar a Secção 3 deste Lab.

Os conflitos de controlo são causados pelas instruções de salto condicional e incondicional (*branch* e *jump*), porque estas instruções têm de calcular qual a instrução a ir buscar a seguir. E, no caso das instruções de salto condicional, têm também de calcular se o salto é efetuada ou não. Em contrapartida, para todas as outras instruções, a instrução a ir obter a seguir está em $PC + 4$.

Nalguns processadores, os conflitos de controlo nunca ocorrem. Por exemplo, os conflitos de controlo não ocorrem em processadores em que uma dada instrução é completamente executada antes de a instrução seguinte ser obtida. Isto é verdade tanto para os processadores de ciclo único como para os de ciclo múltiplo no DDCARV. Especificamente, porque um *branch* é executado completamente, as decisões sobre se o *branch* é efetuado e que instrução ir buscar a seguir são resolvidas antes de a instrução seguinte ser obtida. Em contrapartida, os processadores com pipeline vão buscar a instrução seguinte antes dessas decisões serem resolvidas.

Um mecanismo para lidar com os conflitos de controlo é parar o pipeline até que seja tomada a decisão de qual a instrução a ir buscar após o *branch*. Uma vez que esta decisão é tomada no andar EX1 do SweRV EH1 (como veremos na Secção 2), o pipeline teria de ser bloqueado durante quatro ciclos em cada salto (ver Figura 1 no Lab 11 - que mostra o pipeline). Isto iria degradar gravemente o desempenho do sistema se os *branch* ocorressem frequentemente, o que é tipicamente o caso em programas reais, pelo que esta solução não é implementada no SweRV EH1.

Uma alternativa é prever se o *branch* será efetuado ou não e começar a obter instruções a partir do caminho previsto. Assim que a decisão sobre o *branch* estiver disponível, o processador pode apagar as instruções obtidas (*flush*) se a previsão estiver errada (caso em que deve ser paga uma penalização por previsão errada do *branch*), ou pode continuar a execução das instruções obtidas se a previsão estiver correta (caso em que não há perda de desempenho). No SweRV EH1 estão disponíveis dois preditores de saltos condicionais (*Branch Predictors* - BPs), que analisamos neste Lab: um **Branch Predictor naïve**, que prevê sempre que os *branch* não são executados e, por isso, oferece um desempenho fraco sem custos de hardware, e um **Branch Predictor Gshare**, que oferece um desempenho superior à custa de hardware adicional.

Na Secção 2, descrevemos a execução de uma instrução `beq` no SweRV e, em seguida, realizamos algumas simulações de exemplo utilizando o BP naïve (este é o cenário típico assumido em manuais como o DDCARV). Em seguida, na Secção 3, explicamos como os

conflitos de controlo podem ser tratados de forma mais eficiente utilizando o Branch Predictor Gshare que o SweRV EH1 implementa.

2. EXECUÇÃO DA INSTRUÇÃO `beq` E CÁLCULO DO PC

Nesta secção, analisamos a execução de uma instrução `beq` no SweRV EH1. Em primeiro lugar, na Secção 2.A, explicamos como as instruções `beq` são executadas no andar EX1 e como o endereço Fetch e o endereço Next Fetch são calculados no andar FC1 (isto completa a explicação do andar FC1 que iniciámos na Secção 2.B.i do Lab 11). Embora a figura incluída (Figura 1) e a maior parte das descrições sejam válidas para qualquer instrução, concentramo-nos na execução de uma instrução `beq` numa configuração de processador que usa o BP naïve, em que os *branch* são sempre previstos como *não tomados* (como é feito no DDCARV ou no PaHe). Em seguida, na Secção 2.B, realizamos algumas experiências para exemplificar estes conceitos. Mais uma vez, para estas experiências, desativamos a utilização do *Branch Predictor* e, em vez disso, utilizamos uma previsão de *não tomada* para todos os *branch* (ou seja, o que chamamos de BP naïve).

A. Explicação teórica

Figura 1 mostra as principais estruturas no andar FC1 que são utilizadas para determinar o **endereço de Fetch** (que é o valor no contador de programa (PC), definido em DDCARV como um registo que contém o endereço de memória da instrução atual) e o **próximo endereço de Fetch** (que é o valor utilizado para atualizar o PC no final de cada ciclo). A figura também mostra as estruturas necessárias para executar uma instrução `beq` no andar EX1 (a maior parte do hardware mostrado também é usado na execução de outras instruções de salto). Tal como noutros laboratórios, os nomes dos sinais utilizados na figura são os nomes reais utilizados nos módulos Verilog do processador SweRV EH1.

i. Computação do endereço de Fetch

Como mostra a Figura 1 o andar FC1 inclui dois multiplexers: Um multiplexer 2:1 que produz o endereço de Fetch em `ifc_fetch_addr_f1[31:1]`, e um multiplexer 5:1 que calcula o endereço de Fetch seguinte e o coloca no sinal `fetch_addr_bf[31:1]`.

- **Multiplexer 2:1:** produz o sinal `ifc_fetch_address_f1[31:1]`, o endereço de memória da instrução obtida no ciclo atual, que, como analisámos na Figura 3 do Lab 11, é fornecido ao Controlador de Memória para ler o pacote de instruções de 128 bits da Cache de Instruções. As duas entradas para este multiplexer são:
 - o O endereço do destino do salto (`exu_flush_path_final[31:1]`) é calculado no andar EX1, como analisaremos mais adiante.
 - o O endereço do próximo Fetch (`ifc_fetch_addr_f1_raw[31:1]`), calculado e registado no ciclo anterior como a saída do multiplexer 5:1 incluído nesta fase e analisado a seguir (`fetch_addr_bf[31:1]`).

O sinal de controlo deste multiplexer é designado por `exu_flush_final` e é fornecido pelo andar de execução. Se o *flush* tiver de ocorrer a partir do endereço de destino do salto, `exu_flush_final = 1` e `exu_flush_path_final[31:1]` é

utilizado como endereço de *flush*; caso contrário, `exu_flush_final = 0` e `ifc_fetch_addr_f1_raw[31:1]` é utilizado como endereço de *flush*.

Note-se que um multiplexer 2:1 análogo é utilizado nos processadores explicados em DDCARV para atualizar o PC em cada ciclo.

- **Multiplexer 5:1:** produz o sinal `fetch_addr_bf[31:1]`, o endereço proveniente de uma das cinco fontes seguintes:
 - O endereço de Fetch (`ifc_fetch_addr_f1`), que é utilizado nalguns casos quando o PC permanece o mesmo de um ciclo para o outro.
 - O endereço sequencial seguinte (`fetch_addr_next`), que é calculado como o endereço de Fetch (`ifc_fetch_addr_f1`) + 16, e que aponta para o pacote de 128 bits seguinte.
 - O endereço previsto pelo Branch Target Buffer (`ifc_bp_btb_target_f2`), que é uma das principais estruturas do preditor de saltos, e que é utilizado como endereço de Fetch quando se prevê que seja efectuado um salto.
 - Mais dois sinais de entrada (`miss_addr` e `exu_flush_path_final`) que correspondem ao *caminho de miss* e ao *caminho de flush*, respetivamente, mas que não serão analisados neste Lab.

O sinal fornecido por este multiplexer (`fetch_addr_bf[31:1]`) é registado e utilizado no ciclo seguinte como entrada para o multiplexer 2:1 analisado acima.

Note-se que este multiplexer 5:1 não existe nos processadores da DDCARV, que têm desenhos mais simples.

ii. Execução da Instrução *beq*

Um salto condicional deve calcular o endereço de destino do *branch* e testar se a condição é satisfeita. Especificamente, no caso do SweRV EH1 (ver Figura 1):

- **Cálculo do Branch Target Address:** um novo somador é utilizado em EX1 para calcular o endereço de destino do *branch* e colocá-lo no sinal `flush_path[31:1]`. Este sinal é fornecido como entrada para o multiplexer 2:1 em FC1 (`exu_flush_path_final[31:1]`) através de alguma lógica e registos.
- **Resolução de condições:** um novo módulo é utilizado em EX1, dentro do módulo `exu_alu_ctl`, para verificar se os dois operandos são iguais (`eq = 1`) ou não (`eq = 0`). Com base no sinal `eq` (e em alguns outros sinais, como `ap.beq`, que analisará numa tarefa proposta), os sinais `flush_upper` e `exu_flush_final` são calculados e fornecidos à fase FC1, onde o último é utilizado como sinal de controlo do multiplexer 2:1. Este sinal de controlo (`exu_flush_final`) é 1 quando o *branch* foi mal previsto e 0 caso contrário.

Especificamente, no caso de uma instrução *beq* e assumindo a utilização do BP naïve explicada acima, em que todos os *branch* são previstos como não efetuados, se os dois operandos do *branch* não forem iguais, então o *branch* não deve ser efetuado e a previsão está correta: `exu_flush_final = flush_upper = eq = 0`. Neste caso, o processador

pode continuar a ir buscar e a executar instruções sequencialmente e não há perda de desempenho. Analisaremos esta situação na secção 2.B.i.

Em contrapartida, se os dois operandos forem iguais, o *branch* tem de ser tomado e, no caso do BP naïve que prevê que não é tomado, ocorreu um erro de previsão:

`exu_flush_final = flush_upper = eq = 1`. Neste caso, como explicaremos na Secção 2.B.ii, são desencadeadas as seguintes ações na pipeline EH1 do SweRV (ver Figura 1).

- Quando `exu_flush_final = 1`, a obtenção de instruções é redirecionada para o endereço de destino do salto, selecionando a entrada 1 do multiplexer 2:1 em FC1 (`ifc_fetch_addr_f1[31:1] = exu_flush_path_final[31:1]`), que contém o endereço de destino do salto calculado no andar EX1, como explicado acima.
- As fases da pipeline que precedem EX1 são descarregadas. Para esse efeito, são fornecidos vários sinais (`exu_flush_final`, `exu_flush_upper_e2`, `exu_i0_flush_final` e `exu_i1_flush_final`) às fases anteriores (a utilização destes sinais não é especificada na Figura 1).

TAREFA: Examinar os elementos do processador incluídos na Figura 1 no código Verilog e explique como funcionam.

- Os elementos apresentados no andar de descodificação (ficheiro de registos, 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**.

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

TAREFA: Analise no código Verilog o efeito dos sinais `exu_flush_final`, `exu_flush_upper_e2`, `exu_i0_flush_final` e `exu_i1_flush_final` em EX1 e nos andares que o antecedem: FC1, FC2, Align e Decode. Para esta análise, pode ser útil utilizar as simulações da Secção 2.B, onde se podem incluir os sinais necessários.

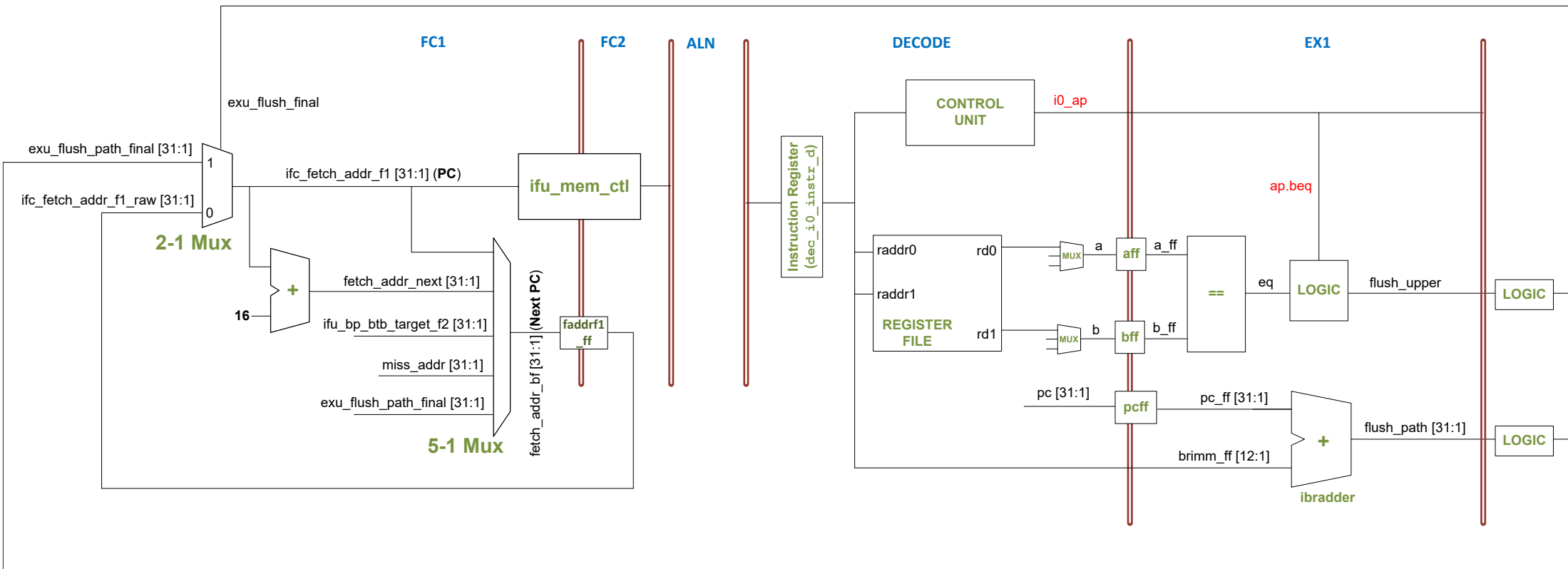


Figura 1. Vista de alto-nível da instrução `beq` em execução através do SweRV EH1

B. Experiências

Agora que descrevemos os principais conceitos da execução de uma instrução `beq` em EX1 e o cálculo do endereço de Fetch e do endereço de Fetch seguinte em FC1, apresentamos algumas simulações para solidificar estes conceitos.

Ao longo desta secção, trabalhamos com o exemplo apresentado na Figura 2 que executa um ciclo que se repete durante 0xFFFF iterações (ou seja, 65.535 em decimal) e que contém duas instruções `beq`: a primeira `beq` será sempre *não tomada* (exceto na última iteração do ciclo) e a segunda será sempre *tomada*. Como é habitual, as instruções que queremos analisar (neste caso as instruções `beq`, destacadas a vermelho) estão rodeadas de vários nops para as isolar das instruções anteriores e posteriores. A pasta `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction` fornece o projeto PlatformIO que pode ser analisado, simulado e modificado à vontade.

```
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
```

Figura 2. Programa com instruções `beq`

Nas nossas experiências, desativamos a utilização de instruções comprimidas. Além disso, como mencionámos acima, nesta secção o Preditor de Salto Gshare disponível no SweRV EH1 está desativado e prevê-se sempre que os *branch* não sejam *tomados* (BP naïve). Isto é feito através da inclusão de duas instruções que permitem ao utilizador configurar o processador durante a execução. Tal como descrito no Apêndice B do Lab 11, deve incluir as duas instruções seguintes no seu código para desativar o Branch Predictor e, em vez disso, utilizar uma previsão de *não tomada* para cada salto.

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

Nesta configuração, o primeiro *branch* do programa (Figura 2) será sempre corretamente previsto (exceto na última iteração do ciclo, que não analisaremos aqui) e o segundo *branch* será sempre mal previsto, o que provocará um *flush* das quatro etapas anteriores e um redirecionamento da execução. De seguida vamos analisar a execução das duas instruções `beq`.

i. Execução do primeiro Branch: `beq t3, t4, OUT`

Nesta secção, analisamos a execução da primeira instrução de *branch* da Figura 2 que é sempre prevista corretamente (exceto na última iteração do ciclo, que não analisamos aqui). Abra o projeto no PlatformIO, construa-o e abra o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis`). Observe que a primeira instrução `beq` é colocada no endereço `0x000001a8`:


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

De seguida, simulamos o programa da Figura 2 no Verilator, como explicado no GSG, e depois abrimos o ficheiro de *trace* gerado pelo simulador no GTKWave. Figura 3 faz zoom numa iteração aleatória do ciclo (a primeira iteração deve ser evitada, pois contém I\$ misses que dificultam a análise, bem como a última iteração, que falha a predição) e foca a execução da primeira instrução `beq`.

A maioria dos sinais incluídos na figura são os que mostrámos no diagrama da Figura 1. No entanto, deve ter em conta que os sinais que contêm endereços de instruções (marcados com um sufixo `_ext`) foram alargados para a simulação com 1 bit à direita igual a 0 por uma questão de clareza (note-se que os sinais originais não alargados no código Verilog não incluem o bit menos significativo, uma vez que é sempre 0); especificamente:

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

O ficheiro `test_1.tcl` é fornecido com o projeto. Para usá-lo no GTKWave, clique em *File* → *Read Tcl Script File*, e abra o ficheiro `[RVfpgaPath]/RVfpga/Labs/Lab13/BEQ_Instruction/test_1.tcl`. Depois, clique várias vezes

em *Zoom In* () e passe para qualquer iteração do ciclo, exceto a primeira ou a última. Verá a execução das duas instruções `beq`; Figura 3 mostra o que você deve observar para a primeira instrução de salto.

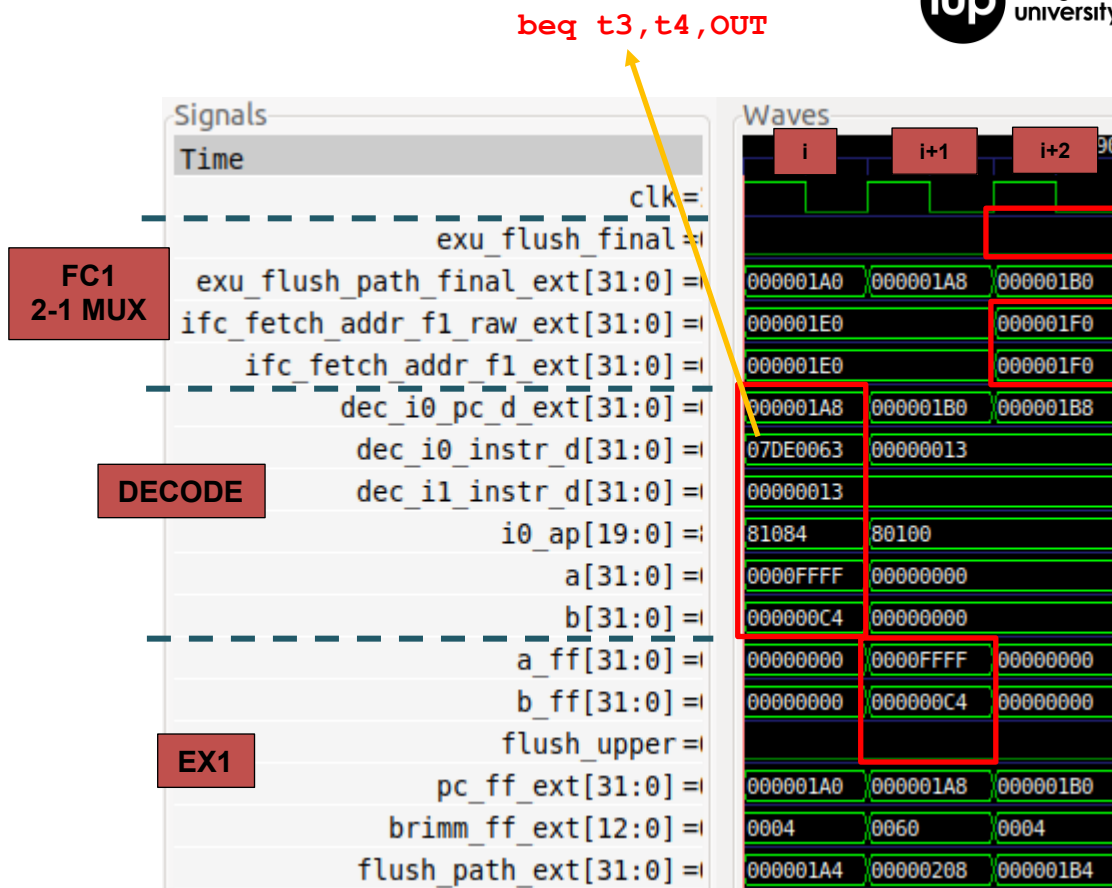


Figura 3. Simulação do Verilator para a execução do primeiro `beq` da Figura 2

Analisar a forma de onda da Figura 3 e o diagrama da Figura 1 em simultâneo. Figura 3 mostra três ciclos consecutivos: Decode do `beq` (ciclo i), EX1 do `beq` (ciclo $i+1$), e seleção do PC seguinte em FC1 após resolução do `beq` (ciclo $i+2$).

- **Ciclo i - Etapa de descodificação para a instrução `beq`:** O sinal `dec_i0_pc_d_ext` contém o endereço da instrução no andar de descodificação (na Via 0), que para o primeiro `beq` é 0x000001A8, e o sinal `dec_i0_instr_d` (normalmente chamado de Instruction Register (IR) nos manuais) contém a instrução de máquina de 32 bits, que para o primeiro `beq` é 0x07DE0063 (em binário: 0000 0111 1101 1110 0000 0000 0000 0110 0011).

Em RISC-V, o *opcode* para a instrução `beq` é (ver Apêndice B de [DDCARV]):

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

para poder verificar que 0x07DE0063 corresponde a: `beq t3, t4, OUT` ($\text{imm}_{12:0} = 0 \times 060$). Lembre-se que o imediato dá o *offset* do endereço de destino a partir do PC atual. O endereço de destino (indicado pela etiqueta "OUT:") é 24 instruções (i.e., 7 nops + 1 add + 7 nops + 1 beq + 7 nops + 1 nop = 24 instruções) depois do PC atual (i.e., `beq t3, t4, OUT`). Isto é $24 \times 4 = 96$ (0x60) bytes para além do PC atual.

Neste andar, **são gerados os sinais de controlo do pipeline**. Para a primeira instrução `beq`, são definidos os seguintes bits de `i0_ap` (que para esta instrução é igual a 0x81084 - ver SweRVref.docx):

- o `valid`: indica que se trata de uma instrução válida que utiliza a ALU.
- o `beq`: indica que se trata de uma instrução *de salto se igual*.
- o `sub`: indica que a UAL deve efetuar uma subtração. Algumas

instruções *branch* utilizam o resultado da subtração para calcular a comparação (no entanto, este não é o caso para *beq*, como mostraremos).

- o `predict_nt`: indica que o *branch* está previsto como *não tomado*.

Além disso, o **Register File é lido e a instrução de salto é encaminhada para o Pipe I0**. Os sinais *a* e *b* (0xFFFF e 0xC4, respetivamente, neste exemplo) contêm as entradas para o comparador utilizado no andar seguinte, que neste caso coincidem com os valores lidos do Register File (noutros casos, os operandos poderiam ser fornecidos através de *forwarding*, como analisado no Lab 15).

- **Ciclo *i+1* - Fase EX1 para a instrução *beq***: No ciclo seguinte, a instrução *beq* é executada. Os sinais *a_ff* e *b_ff* são comparados. Dado que os dois números (0xFFFF e 0xC4) são diferentes, o salto não é efetuado. Tal como descrito anteriormente, nesta configuração prevê-se que todas as *branch* não sejam tomadas (`i0_ap.predict_nt = 1`). Assim, o *branch* não foi previsto incorretamente (`flush_upper = 0`) e a execução pode continuar como está.
- **Ciclo *i+2* - fase FC1**: No ciclo seguinte, dado que o *branch* foi previsto e resolvido como não efectuado, a procura continua simplesmente de forma sequencial. Na Figura 3 repare que `exu_flush_final = 0` e `ifc_fetch_addr_fl_ext[31:0] = ifc_fetch_addr_fl_raw_ext[31:0] = 0x000001F0`. Este endereço aponta para o próximo conjunto sequencial de instruções de 128 bits. Pode ver que nos dois ciclos anteriores foi obtido o pacote de instruções de 128 bits anterior (`ifc_fetch_addr_fl_ext[31:0] = 0x000001E0`).

TAREFA: Modificar a Figura 1 para incluir os valores de cada sinal mostrado na Figura 3 nos ciclos *i*, *i+1* e *i+2*.

TAREFA: Modificar o programa da Figura 2 para que a primeira instrução de salto recupere os seus operandos de entrada através de encaminhamento.

ii. Execução do segundo Branch: *beq t3, t3, OUT*

Agora, analisamos o segundo *branch*, que é sempre tomado, mas incorretamente previsto como não tomado. Abra o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis`). Observe que a segunda instrução *beq* é colocada no endereço 0x000001E8:

```
0x000001e8:    fbce00e3    beqt3 ,t3,188 <LOOP>
```

Figura 4 mostra os sinais durante uma iteração aleatória do ciclo (mas não a primeira iteração - que evitamos devido a *miss* na cache de instruções (I\$)).

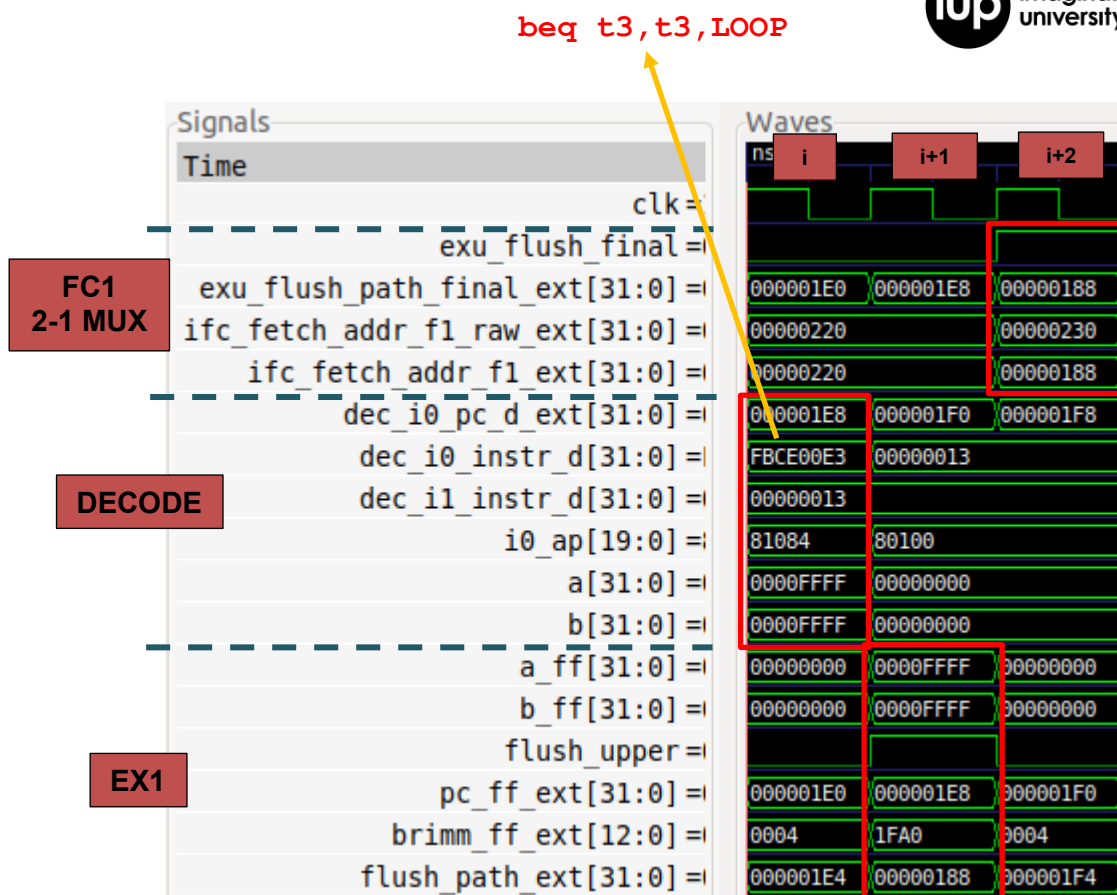


Figura 4. Simulação do Verilator para o segundo Branch do exemplo da Figura 2

Analisar a forma de onda da Figura 4 e o diagrama da Figura 1 em simultâneo. Os valores destacados a vermelho mostram três ciclos consecutivos durante a execução da segunda instrução `beq`: Descodificação do `beq` (ciclo i), EX1 do `beq` (ciclo $i+1$), e seleção do próximo PC em FC1 após resolução do `beq` (ciclo $i+2$).

- **Ciclo i - Etapa de descodificação da instrução `beq`:** O PC (sinal `dec_i0_pc_d_ext`) é `0x000001E8`, e a instrução (sinal `dec_i0_instr_d`) é `0xFBCE00E3` (em binário: 1111 1011 1100 1110 0000 0000 1110 0011).

Em RISC-V, o opcode para a instrução `beq` é (ver Apêndice B de [DDCARV]):

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

Assim, pode verificar que `0xFBCE00E3` corresponde a: `beq t3, t3, LOOP` (`Immediate12:0 = 0x1FA0`). Lembre-se que o imediato dá o *offset* do endereço de destino a partir do PC atual. O endereço de destino (indicado pela etiqueta "LOOP:") é 24 instruções (ou seja, 7 nops + 1 add + 7 nops + 1 beq + 7 nops + 1 add = 24 instruções) *antes* do PC atual (ou seja, `beq t3, t3, LOOP`). Isto é $24 \times 4 = 96$ bytes antes do PC atual. Assim, o imediato codifica -96, que é `0x1FA0`, escrito em representação de complemento de dois bits de 13 bits.

Durante esta fase, **são gerados os sinais de controlo do pipeline**. Para esta instrução `beq`, os sinais de controlo são os mesmos que para a primeira `beq` (ver secção anterior).

Além disso, o Register File é lido e a instrução de salto é encaminhada para o Pipe I0. Os sinais `a` e `b` (`0xFFFF` para ambos) contêm as entradas para o

comparador utilizado no andar seguinte, que neste caso são os valores lidos do Register File.

- **Ciclo $i+1$ - Fase EX1 para a instrução `beq`:** No ciclo seguinte, a instrução `beq` é executada. Por um lado, os sinais `a_ff` e `b_ff` são comparados. Se os dois valores forem iguais, o salto deve ser efetuado. No entanto, como explicado anteriormente, na nossa configuração prevê-se que todas as *branch não sejam tomadas* (`i0_ap.predict_nt = 1`). Assim, o salto foi mal previsto (`flush_upper = 1`). Portanto, as instruções devem ser obtidas no endereço de destino do salto e os andares iniciais do pipeline devem ser descartados.

Nesta fase, o endereço de destino é calculado como a soma entre `pc_ff_ext` (0x1E8) e `brim_ff_ext` (0x1FA0). O resultado é colocado no sinal `flush_path_ext` (0x00000188).

- **Ciclo $i+2$ - Fase FC1:** No ciclo seguinte, a execução deve continuar no endereço de destino do salto. Na Figura 4 pode ver que `exu_flush_final = 1` e `ifc_fetch_addr_f1_ext = exu_flush_path_final_ext = 0x00000188`. Este endereço corresponde ao endereço de destino do *branch*, que é o endereço da primeira instrução do ciclo (note que este é um *branch* para trás).

TAREFA: Modificar a Figura 1 para incluir os valores de cada sinal mostrado na Figura 4 nos ciclos i , $i+1$ e $i+2$.

TAREFA: Analisar o funcionamento dos dois multiplexers da FC1 com o exemplo da Figura 2 examinando os sinais em diferentes circunstâncias.

Por exemplo, analise como é feito o Fetch para uma execução sequencial (ou seja, um grupo de instruções sem *branch*). Verá que, no processador SweRV EH1, a operação neste caso é a seguinte:

- Nos ciclos pares, o `fetch_addr_next` é selecionado utilizando o multiplexer 5:1, que contém o endereço de Fetch atual (`ifc_fetch_addr_f1`) mais 16, lendo assim o próximo conjunto sequencial de instruções de 128 bits (lembre-se que uma leitura I\$ fornece 128 bits).

- Nos ciclos ímpares, o `ifc_fetch_addr_f1` é selecionado utilizando o multiplexer 5:1, pelo que não são obtidas novas instruções.

Desta forma, quatro instruções de 32 bits são obtidas a cada 2 ciclos, o que corresponde à mesma taxa de instruções necessárias para a fase de decodificação (2 instruções por ciclo).

Note-se que nos processadores da DDCARV o PC é simplesmente incrementado de quatro em cada ciclo (para execução sequencial) para obter uma instrução por ciclo.

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

TAREFA: No Lab 15, analisámos como os conflitos de dados RAW são resolvidos no andar de Commit através das ALUs secundárias. À semelhança das instruções A-L que estudámos nesse Lab, uma instrução de Branch 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 redirecionando a partir do andar de confirmação. Analise esta situação utilizando uma versão ligeiramente modificada do programa da Figura 2 fornecida em `[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction_HazardCommit`, e o ficheiro `.tcl` fornecido nessa mesma pasta.

3. O Branch Predictor Gshare utilizado pelo SweRV EH1

Na Secção 2, discutimos a configuração do SweRV EH1 que inclui apenas um preditor de saltos naïve que prevê sempre que não foi tomado, mas nesta secção analisamos o funcionamento do Preditor de Saltos Gshare disponível no SweRV EH1. O BP Gshare efetua uma previsão mais inteligente para cada instrução de salto, o que melhora o desempenho, mas requer hardware adicional. Antes de descrevermos o funcionamento do BP Gshare no SweRV EH1, comparamos o desempenho dos dois BPs.

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

Ativar o preditor de salto utilizado no SweRV EH1 (comentando as duas instruções iniciais na Figura 2) e analisar a simulação e a execução na placa.

Compara as duas experiências e explique os resultados.

NOTA: Um artigo clássico publicado por Scott McFarling em 1993 chama-se "Combining Branch Predictors" (<https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>). Descreve, na Secção 7, o funcionamento do Branch Predictor Gshare. Você também pode procurar outros documentos, como <https://people.engr.ncsu.edu/efg/521/f02/common/lectures/notes/lec16.pdf>. Recomendamos a sua leitura para compreender o funcionamento do BP Gshare antes de iniciar esta secção.

Figura 5 mostra uma visão simplificada do BP Gshare disponível no SweRV EH1. Todas as estruturas do BP são implementadas no módulo `ifu_bp_ctl` (no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_bp_ctl.sv`). As estruturas relacionadas com o BP Gshare estão rodeadas por um quadrado azul na figura.

Esta BP é composta pela tabela de histórico de salto (Branch History Table - BHT), que prevê a direção do salto (*tomada* ou *não tomada*), e pelo buffer de destino de salto (Branch Target Buffer - BTB), que prevê o endereço de destino no caso de saltos tomados. Na nossa configuração padrão, o BHT contém 128 entradas de 2 bits. Pode ser encontrado nas linhas 1615-1705 do módulo `ifu_bp_ctl`. Na nossa configuração por defeito, o BTB contém 32 entradas de 13 bits. Pode encontrá-lo nas linhas 1439-1613 do módulo `ifu_bp_ctl`.

Para efetuar uma previsão de salto, ocorre o seguinte em cada ciclo (ver Figura 5):

1. O endereço de Fetch (`ifc_fetch_addr_f1 [31:1]`) e alguns outros sinais são transmitidos através de vários módulos de *hashing* dentro do módulo `ifu_bp_ctl`: `f1hash`, `rdtagf1`, `fghrfs`... Todos estes módulos de *hashing* são implementados no ficheiro

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/beh/beh_lib.sv, utilizando algumas das macros definidas em [RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh.

Como exemplo, o módulo `fghrhs` recebe o sinal `btb_rd_addr_f1`, que provém de um *hashing* do endereço de Fetch (`f1hash(.pc(ifc_fetch_addr_f1[31:1]), .hash(btb_rd_addr_f1[RV_BTBT_ADDR_HI:RV_BTBT_ADDR_LO]))`) e `fghr_ns` (que é o Global History Register), e produz o sinal `bht_rd_addr_hashed_f1`.

```
rvbhtb_ghr_hash fghrhs (.hashin(btb_rd_addr_f1[RV_BTBT_ADDR_HI:RV_BTBT_ADDR_LO]), .ghr(fghr_ns[RV_BHT_GHR_RANGE]), .hash(bht_rd_addr_hashed_f1[RV_BHT_ADDR_HI:RV_BHT_ADDR_LO]));
```

Este sinal é utilizado para aceder à tabela BHT do Branch Predictor Gshare.

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

2. Todos estes sinais *hash* (`btb_rd_addr_f1`, `bht_rd_addr_hashed_f1`, `fetch_rd_tag_f1`, etc.) são utilizados para aceder às duas principais estruturas que constituem o BP Gshare: o BHT e o BTB.

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

3. Como resultado do acesso à BHT, obtém-se uma previsão de direção no sinal `ifu_bp_kill_next_f2`, que é 0 se se prevê que o Branch *não seja tomado* e 1 se se prevê que seja *tomado*. Este sinal é utilizado, para além de outros sinais que não descrevemos aqui, para calcular o sinal de controlo do multiplexer 5:1 de FC1.

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

4. Como resultado do acesso ao BTB, o endereço de destino previsto para os *branch* efetuados é obtido a partir de um somador no sinal `ifu_bp_btb_target_f2` [31:1]. (Note-se que o endereço previsto também pode vir da pilha de endereços de retorno (Return Address Stack - RAS) no caso de ser prevista uma instrução `ret`). Este sinal é uma das entradas do multiplexer 5:1 do FC1.

TAREFA: Analisar o modo como o endereço de destino previsto (`ifu_bp_btb_target_f2`) é obtido a partir do valor lido no BTB (`btb_rd_tgt_f2` [11:0]) e do endereço de Fetch no FC2 (`ifc_fetch_addr_f2` [31:4]).

TAREFA: Analisar o RAS implementado no processador SweRV EH1. Uma pesquisa na Internet também fornecerá 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).

5. No multiplexer 5:1 da FC1, se `ifu_bp_kill_next_f2` = 1, então o endereço de destino previsto é utilizado como o endereço do próximo Fetch: `fetch_addr_bf` [31:1] = `ifu_bp_btb_target_f2` [31:1] (a não ser que o pipeline esteja a ser descarregado). Em vez disso, se `ifu_bp_kill_next_f2` = 0, uma das outras quatro entradas é utilizada como endereço da próximo Fetch.

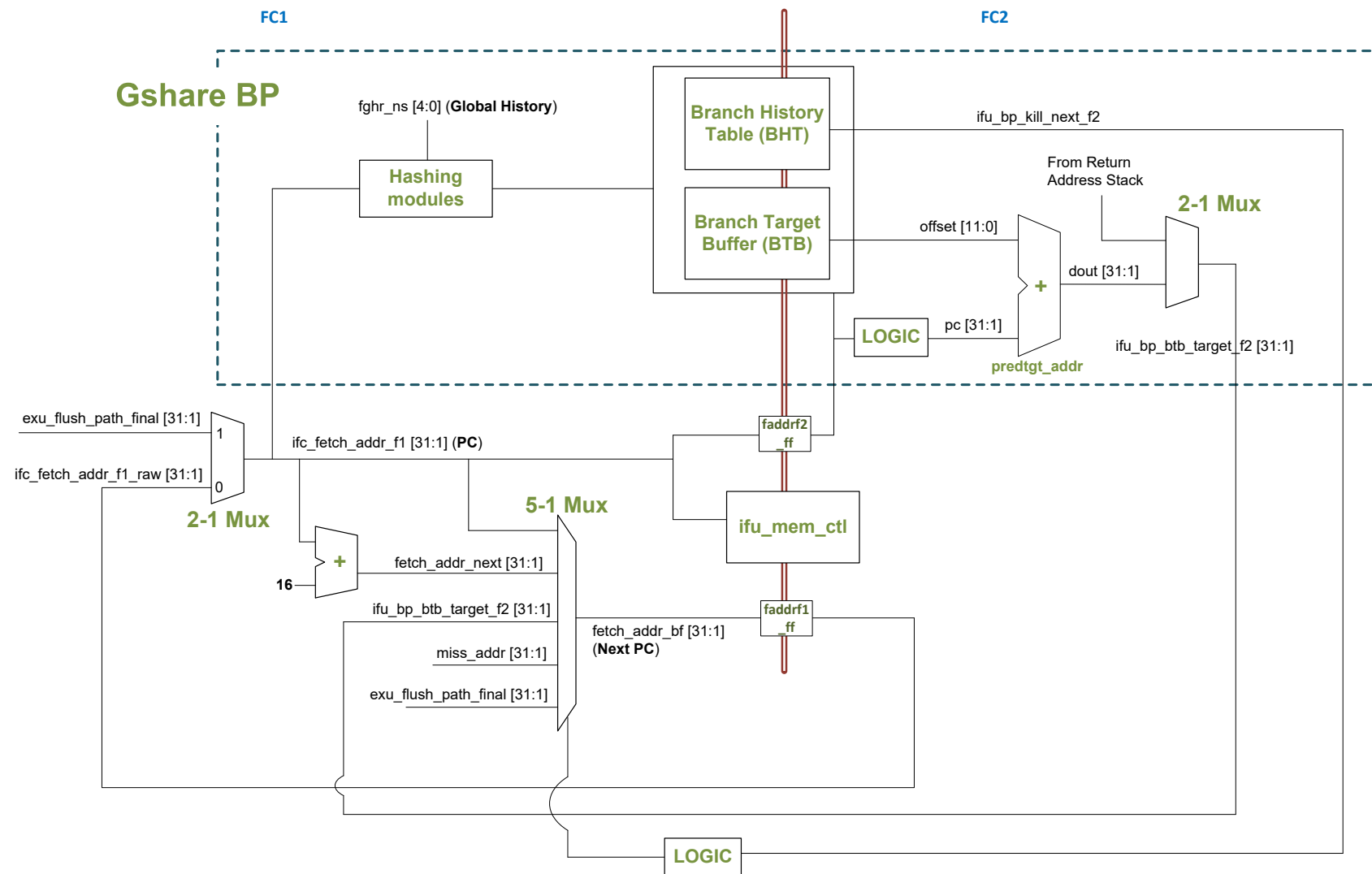


Figura 5. Principais estruturas (envoltas no quadrado azul) que compõem o Branch Predictor Gshare disponível no SweRV EH1

Analisamos agora a execução da segunda instrução de salto no programa, tal como foi feito na Secção 2.B.ii. Lembre-se que a segunda instrução `beq` é colocada no endereço `0x000001E8` no nosso programa, o que significa que está contida no pacote de 128 bits mapeado no intervalo de endereços `0x1E0-0x1EF`:

Figura 6 faz um zoom numa iteração aleatória do ciclo. Como de costume, a primeira iteração é evitada, pois contém 1\$ misses - além disso, a previsão de salto falha para esta instrução de salto na sua primeira iteração. A maioria dos sinais incluídos na figura são os que mostrámos na Figura 5. O ficheiro *teste_1_BP.tcl* é fornecido com o projeto. Para usá-lo no GTKWave, clique em *File → Read Tcl Script File*, e abra o ficheiro *[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/test_1_BP.tcl*. Depois, clique várias

The timing diagram for the first instruction (i) shows the following signals and their values over time:

- ifc_fetch_addr f1 ext[31:0]**: 000001E0
- wayhit_f2[7:0]**: 00
- pc_ext[31:0]**: 000001DE
- offset_ext[12:0]**: 0000
- ifu_bp_btb_target f2 ext[31:0]**: 000001DE
- ifu_bp_kill_next_f2**: 000001EE
- fetch_addr_bf_ext[31:0]**: 000001E0
- dec_i0_pc_d_ext[31:0]**: 000001AC
- dec_i0_instr_d[31:0]**: 00000013
- dec_i1_instr_d[31:0]**: 00000013
- flush_upper**: 00000013

Analisar a forma de onda da Figura 6 e o diagrama da Figura 5 ao mesmo tempo. Os valores destacados a vermelho correspondem à segunda instrução `beq` à medida que esta atravessa as etapas do pipeline.

- Imagination University Programme - RV/fpga Lab 16: Conflitos de Controlo: Instruções de Branch
Versão 2.0 - 30 de outubro de 2021
© Copyright Imagination Technologies

- **Ciclo $i+3$:** O endereço de Fetch é o endereço de destino previsto do salto, que foi calculado no ciclo anterior: `ifc_fetch_addr_fl_ext = 0x00000188`.
- **Ciclo $i+7$:** O Branch é decodificado na Via 1 (`dec_i1_instr_d = 0xFBCE00E3`).
- **Ciclo $i+8$:** O salto é executado. A previsão estava correta, pelo que não é necessário desencadear uma descarga (`flush_upper = 0`).
- **Ciclo $i+9$:** A execução prossegue normalmente através do endereço de destino do salto, dado que a previsão estava correta.

TAREFA: Explicar como é que o Global History Register é atualizado no módulo `ifu_bp_ctl`.

4. EXERCÍCIOS

1) Implementar um Branch Predictor Bimodal e comparar o seu desempenho com o BP Gshare.

2) (O exercício seguinte é baseado no exercício 4.25 do livro "Computer Organization and Design - RISC-V Edition", de 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 usada a previsão perfeita de saltos condicionais (no caso do SweRV EH1, podemos emular este comportamento simplesmente evitando a primeira iteração), que o pipeline tem suporte de Full Forwarding (mais uma vez, este é o caso no SweRV EH1), e que os saltos condicionais 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`.