



Imagination大学计划

RVfpga实验16

控制冒险：分支指令

1. 简介

在本实验中，我们将完成冒险分析。在之前两个实验中，我们研究了SweRV EH1处理器存在的**结构冒险**和**数据冒险**，本实验将专门分析**控制冒险**。正如S.Harris和D.Harris在《数字设计和计算机体系结构》（RISC-V版本，简称DDCARV）中所述，当未能在规定时间内决定下一条要取的指令时，就会发生**控制冒险**。

注：在分析SweRV EH1控制冒险逻辑之前，建议用户先阅读DDCARV第7.5节，了解如何在流水线处理器中执行beq指令和解除控制冒险。有关控制冒险的具体章节为第7.5.3节。另外，在开始本实验的第3部分之前，建议用户先阅读第7.7.3节有关分支预测的内容。

控制冒险是由分支和跳转指令引起的，因为这些指令必须计算下一个取指的地址。分支指令还必须计算是否要采取分支。相比之下，对于所有其他类型的指令，下一个取指的地址均为PC + 4，无需计算。

一些处理器永远不会发生控制冒险。举例来说，在处理器中，如果给定指令在下次取指前便完全执行，则不会发生控制冒险。DDCARV中的单周期和多周期处理器均符合这一条件。具体而言，由于分支指令能够完全执行，因此在下次取指前，处理器便能够确定是否采取分支并确定下一条要取的指令。与之相反，流水线处理器在做出这些决定之前便要进行下次取指。

控制冒险的一种处理方法是暂停流水线，直至确定分支后要取的指令。由于决定是在SweRV EH1的EX1阶段做出的（将在第2部分进行介绍），因此，流水线在每条分支处必须暂停四个周期（参见实验11的图1中所示的流水线）。在实际程序中，分支往往会频繁出现，导致系统性能严重降低，因此SweRV EH1中并未实现此解决方案。

另一种方法是预测是否会采取分支，并从预测的路径中取指。分支确定后，如果与预测相反，处理器可以清除取的指令（在这种情况下，必须承担分支预测错误的损失）；如果与预测相同，则继续执行取的指令（在这种情况下不存在性能损失）。在本实验中，我们分析了SweRV EH1提供的两种分支预测器（Branch Predictor, BP）：一种是**简单分支预测器**，该预测器始终假定不会采取分支，虽然无需花费硬件成本，但会导致系统性能低下；另一种是**Gshare分支预测器**，该预测器需要花费额外的硬件成本，能够提供更高的性能。

在第2部分中，我们将介绍如何在SweRV中执行beq指令，然后会使用简单BP进行一些示例仿真（采用DDCARV等教材中假定的典型场景）。然后，在第3部分，我们将展示SweRV EH1如何使用Gshare分支预测器更高效地处理控制冒险。

2. beq指令执行和PC计算

在本部分中，我们将分析SweRV EH1中beq指令的执行情况。首先，在第2.A部分，我们将介绍如何在EX1阶段执行beq指令，以及如何在FC1阶段计算取指地址和下次取指地址（对实验11的第2.B.i部分中有关FC1阶段的说明进行补充）。随附的图示（图1）和大多数描述适用于所有指令，但我们重点关注beq指令在使用简单BP的处理器上的执行情况，采用该配置时，预测结果始终为不发生分支（类似于DDCARV或PaHe中设定的情景）。在接下来的第2.B部分，我们会进行一些实验，通过实例来阐明相关概念。对于这些实验，我们同样不会使用真正的分支预测器，而是将所有条件分支预测为不发生（即采用所谓的简单BP）。

A. 理论说明

图1所示为FC1阶段中用于确定**取指地址**（程序计数器（Program Counter, PC）中的值，在DDCARV中定义为保存当前指令存储器地址的寄存器）和**下次取指地址**（用于在每个周期结束时更新PC的值）的主要结构。图中还展示了在EX1阶段执行beq指令所需的结构（图中所示的大多数硬件也用于执行其他分支指令）。与其他实验一样，图中所示的信号名称均为SweRV EH1处理器的Verilog模块所使用的实际名称。

i. 取指地址计算

如图1所示，FC1阶段包含两个多路开关：一个是2:1多路开关，用于在ifc_fetch_addr_f1[31:1]中生成取指地址；另一个是5:1多路开关，用于计算下次取指地址并将该地址放入信号fetch_addr_bf[31:1]。

- **2:1多路开关：**生成信号ifc_fetch_address_f1[31:1]，即当前周期中取指的指令的存储器地址，正如实验11的图3中的分析结果，该地址将提供给存储器控制器，用于从指令高速缓存中读取128位指令束。该多路开关的两个输入为：
 - o 分支目标地址（exu_flush_path_final[31:1]），在EX1阶段计算，相关分析如下文所述。
 - o 下次取指地址（ifc_fetch_addr_f1_raw[31:1]），在上一个周期中计算并寄存，用作本阶段中5:1多路开关的输出，相关分析如下文所述（fetch_addr_bf[31:1]）。

该多路开关的控制信号称为exu_flush_final，在执行阶段提供。如果必须从分支目标地址取指，则exu_flush_final = 1，使用exu_flush_path_final[31:1]作为取指地址；否则exu_flush_final = 0，使用ifc_fetch_addr_f1_raw[31:1]作为取指地址。

请注意，DDCARV中所述的处理器使用类似的2:1多路开关在每个周期更新PC。

- **5:1多路开关：**生成信号`fetch_addr_bf[31:1]`，地址来自以下五个来源之一：
 - o 取指地址 (`ifc_fetch_addr_f1`)，用于PC在下一周期保持不变的情况。
 - o 序列中的下一个地址 (`fetch_addr_next`)，计算方式为取指地址 (`ifc_fetch_addr_f1`) + 16，指向下一个128位指令束。
 - o 分支目标缓冲区 (`ifc_bp_btb_target_f2`) 预测的地址，分支目标缓冲区是分支预测器的主要结构之一，如果预测结果为发生分支，则该地址将用作取指地址。
 - o 分别与 *miss path* 和 *flush path* 相对应的另外两个输入信号 (`miss_addr` 和 `exu_flush_path_final`)，但我们在本实验中不会分析这些信号。

该多路复用器提供的信号 (`fetch_addr_bf[31:1]`) 将被寄存，并在下一个周期中用作上述2:1多路开关的输入。

请注意，DDCARV所述的处理器设计更为简单，不包含此5:1多路开关。

ii. beq指令的执行

条件分支必须计算分支目标地址，并测试是否满足条件。具体以SweRV EH1为例（参见图1）：

- **分支目标地址计算：**EX1使用一个新的加法器来计算分支目标地址，并将地址放入信号 `flush_path[31:1]` 中。该信号将通过一些逻辑和寄存器，作为输入 (`exu_flush_path_final[31:1]`) 提供给FC1中的2:1多路开关。
- **条件解析：**EX1使用 `exu_alu_ctl` 模块内部的一个新模块来检查两个操作数是否相等 (`eq = 1` 表示相等，`eq = 0` 表示不相等)。系统会根据 `eq` 信号（以及一些其他信号，如您将在拟定的任务中分析的 `ap.beq` 信号）计算 `flush_upper` 和 `exu_flush_final` 信号，并将两个信号提供给FC1阶段，后者将用作2:1多路开关的控制信号。分支预测错误时，该控制信号 (`exu_flush_final`) 为1，其他情况下则为0。

具体地说，使用beq指令以及上文所述的简单BP（预测所有分支均不采取）时，如果分支的两个操作数不相等，则不得采取分支，此时预测正确：`exu_flush_final = flush_upper = eq = 0`。在这种情况下，处理器可以继续按顺序取指并执行指令，不会出现性能损失。我们将在第2.B.i部分分析该情况。

相反，如果两个操作数相等，则必须采取分支，对于预测不发生分支的简单BP而言，其预测是错误的：`exu_flush_final = flush_upper = eq = 1`。在这种情况下，如下文第2.B.ii部分所述，SweRV EH1流水线中将触发以下操作（参见图1）。

- 当`exu_flush_final = 1`时，通过选择FC1中2:1多路开关的输入1（其中包含在上述EX1阶段中计算的分支目标地址）（`ifc_fetch_addr_f1[31:1] = exu_flush_path_final[31:1]`），取指地址将重定向到分支的目标地址。
- EX1之前的流水线阶段将被清除。为此，将向先前的阶段提供多个信号（`exu_flush_final`、`exu_flush_upper_e2`、`exu_i0_flush_final`和`exu_i1_flush_final`）（图1中未标明这些信号的用途）。

任务：检查图1的Verilog代码中包含的处理器元素，并解释其工作原理。

- 译码阶段显示的元素（寄存器文件、指令寄存器和控制单元）位于模块**dec**、**dec_decode_ctl**和**dec_gpr_ctl**中。
- EX1阶段显示的元素位于模块**exu**和**exu_alu_ctl**中。
- FC1阶段显示的元素位于模块**ifu**和**ifu_ifc_ctl**中。

任务：解释如何在模块**exu_alu_ctl**中通过信号`eq`、控制信号`ap.beq`、`ap.predict_t`和`ap.predict_nt`以及部分其他信号生成信号`flush_upper`。

任务：在Verilog代码中分析信号`exu_flush_final`、`exu_flush_upper_e2`、`exu_i0_flush_final`和`exu_i1_flush_final`对EX1及其之前各阶段（FC1、FC2、对齐和译码）的影响。对于该分析，第2.B部分的仿真非常有用，您可以在其中加入所需的信号。

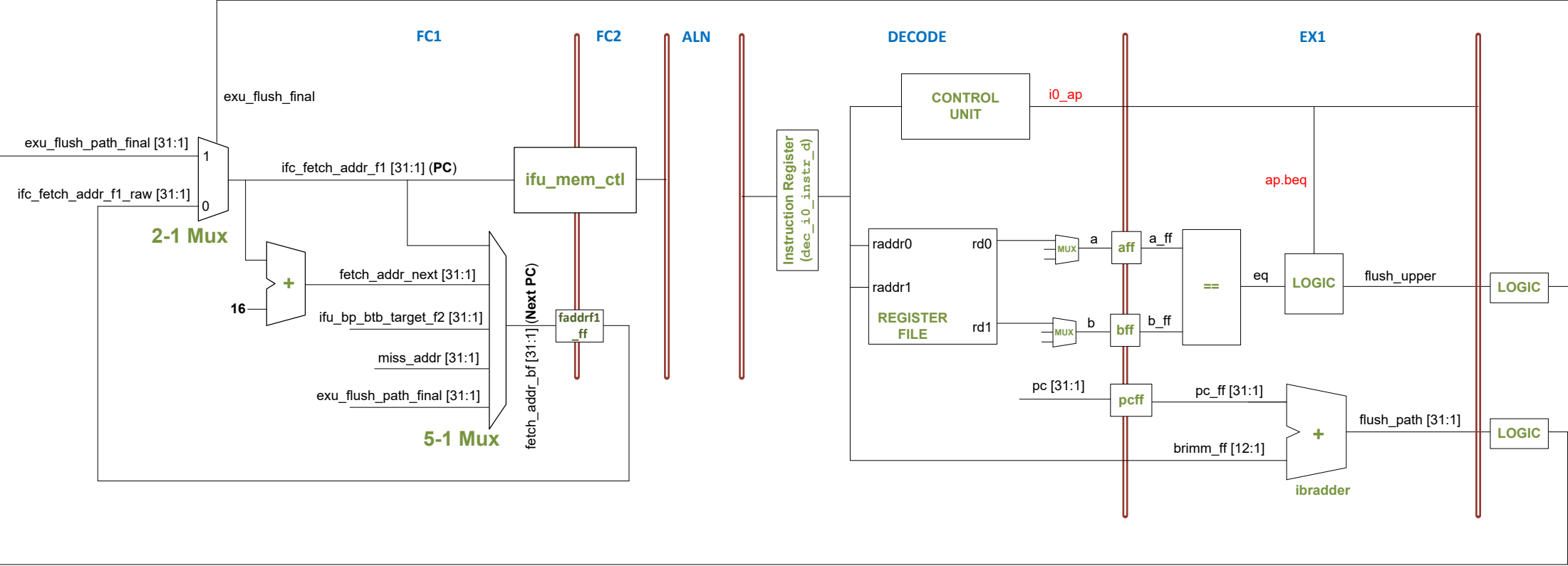


图1. 通过SweRV EH1执行的`beq`指令的高级视图

B. 实验

我们已经介绍了在EX1阶段执行beq指令以及在FC1阶段计算取指地址和下次取指地址时涉及的主要概念，接下来我们将进行一些仿真演示，以加强对于这些概念的理解。

在本部分中，我们将使用图2中所示的示例，该示例会执行一个重复0xFFFF迭代的循环（十进制形式为65,535），并包含两条beq指令：第一条beq指令始终不发生（循环的最后一次迭代除外），第二条指令将始终发生。与往常一样，我们要分析的指令（在本例中，beq指令以红色突出显示）使用多个nop圈起，以便与前面和后面的指令分隔开来。文件夹[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction中提供PlatformIO项目，可根据需要对其进行分析、仿真和修改。

```
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
```

图2. 包含beq指令的程序

在实验中，我们禁止使用压缩指令。此外，如上所述，在本部分中，SweRV EH1中提供的Gshare分支预测器被禁用，并且分支预测结果始终为不发生（简单BP）。上述设置需通过加入两条指令实现，这两条指令允许用户在执行期间配置处理器。如实验11的附录B所述，必须在代码中包含以下两条指令，从而禁用分支预测器并预测不发生每条分支。

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

在该配置中，程序（图2）中的第一条分支始终能够预测正确（循环的最后一次迭代除外，我们暂不分析该情况），而第二条分支始终会预测错误，这将导致前面四个阶段被清除并重新定向。接下来，我们将分析两条beq指令的执行。

i. 执行第一条分支: `beq t3, t4, OUT`

在本部分中，我们将分析图2中第一条分支指令的执行情况，该分支始终能够预测正确（循环的最后一次迭代除外，我们暂不分析该情况）。在PlatformIO中打开、编译项目，然后打开反汇编文件（位于

`[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis`）。

请注意，第一条`beq`指令位于地址`0x000001a8`处：

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


接下来，我们按照GSG中的说明，在Verilator中仿真图2所示的程序，然后在GTKWave上打开仿真器生成的波形文件。图3为循环中随机某次迭代的放大图示（应避免采用第一次迭代，因为该次迭代包含`IS`未命中，会增加分析难度，也应避免最后一次迭代，该次迭代将预测错误），其中重点展示第一条`beq`指令的执行。

图中包含的大多数信号为图1的框图中所示的信号。但必须考虑到，为了清楚起见，在仿真时，包含指令地址的信号（带有后缀`_ext`）均在右边增加了一个数值为0的位（注意，Verilog代码中原始的未扩展信号不包括最低有效位，因为该位始终为0）；具体信号为：

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

该项目随附文件`test_1.tcl`。要在GTKWave中使用该文件，请单击 **“File”**（文件）→ **“Read Tcl Script File”**（读取Tcl脚本文件），并打开文件

`[RVfpgaPath]/RVfpga/Labs/Lab13/BEQ_Instruction/test_1.tcl`。然后单击几次 **“Zoom In”**

（放大）（）移动至循环的任意一次迭代（第一次或最后一次除外）。可以看到两条`beq`指令的执行情况；图3所示为执行第一条分支指令时应观察到的内容。

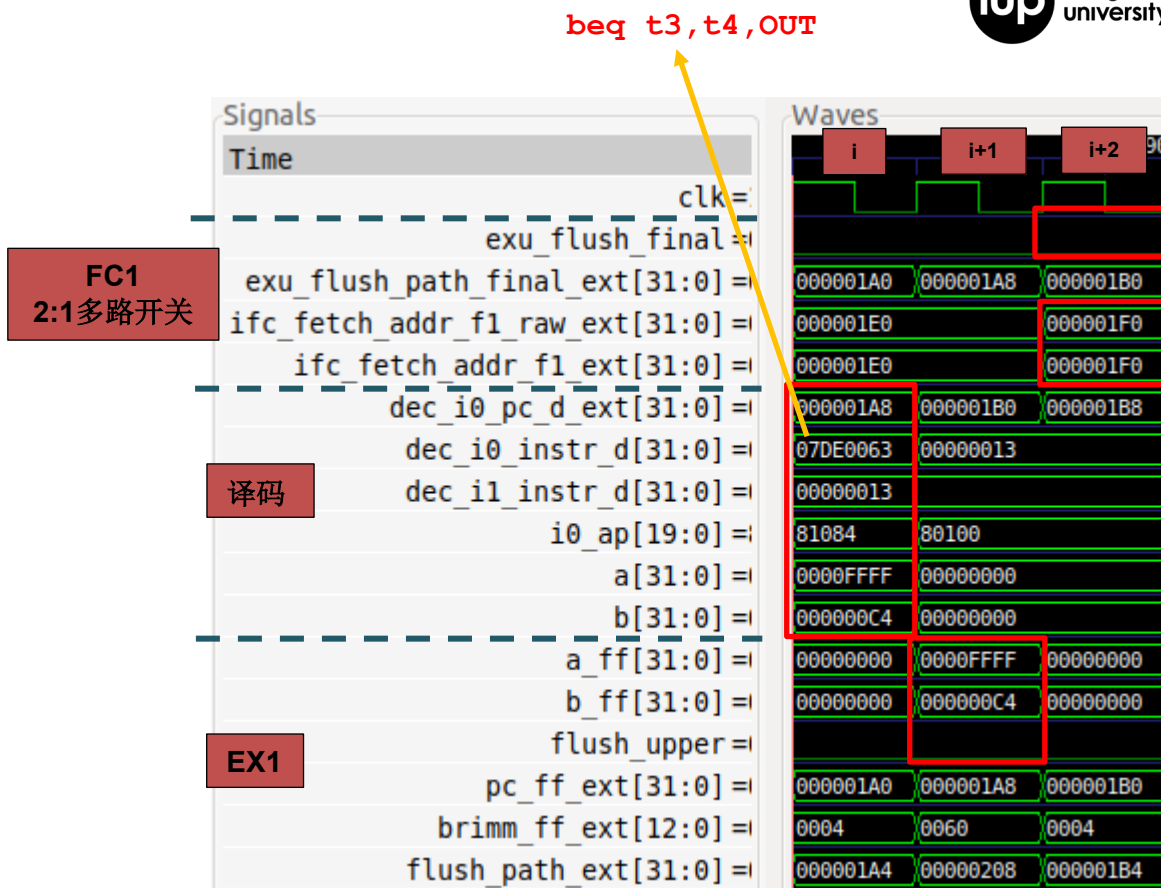


图3. 执行图2中第一条beq指令时的Verilator仿真

同时分析图3中的波形以及图1中的图。图3显示了三个连续的周期：beq的译码阶段（周期*i*）、beq的EX1阶段（周期*i+1*）、在解析beq后的FC1阶段选择下一PC（周期*i+2*）。

- **周期*i* - beq指令的译码阶段：**信号dec_i0_pc_d_ext包含译码阶段中指令的地址（位于路径0中），第一个beq的地址为0x000001A8，信号dec_i0_instr_d（在教材中通常称为指令寄存器（Instruction Register, IR））包含32位机器指令，第一个beq的地址为0x07DE0063（二进制形式为：0000 0111 1101 1110 0000 0000 0110 0011）。

在RISC-V中，beq指令的操作码如下（参见[DDCARV]的附录B）：

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

因此，您可以验证0x07DE0063对应于：beq t3,t4,OUT（imm_{12:0} = 0x060）。请记住，立即数给出了目标地址当前PC的偏移量。目标地址（由标签“OUT:”指示）是当前PC（即beq t3,t4,OUT）之后的24条指令（7条nop + 1条add + 7条nop + 1条beq + 7条nop + 1条nop = 24条指令）。当前PC之后共有24*4 = 96（0x60）个字节。

在此阶段，将产生流水线控制信号。对于第一条beq指令，i0_ap（对于该指令，其值为0x81084 – 参见SweRVref.docx）的以下位置1：

- o valid: 指示该指令为使用ALU的有效指令。
- o beq: 指示该指令为如相等则分支指令。

- **sub:** 指示ALU必须执行减法操作。一些分支指令会使用减法运算的结果来进行比较（然而，如下文所示，**beq**的情况并非如此）。
- **predict_nt:** 指示分支预测结果为不发生。

此外，将读取寄存器文件并将分支指令传送至IO管道。信号a和b（本例中分别为0xFFFF和0xC4）中包含下一阶段所使用的比较器的输入，在本例中，输入为从寄存器文件读取的值（在其他情况下，操作数可通过转发的形式提供，如实验15的分析所示）。

- **周期*i+1* - beq指令的EX1阶段:** 在下一周期中，将执行**beq**指令。比较信号a_ff和b_ff。如果两个数（0xFFFF和0xC4）不同，则不发生分支。如上文所述，在此配置中，所有分支的预测结果均为不发生（i0_ap.predict_nt = 1）。因此，分支预测结果正确（flush_upper = 0），可以正常执行。
- **周期*i+2* - FC1 阶段:** 在下一周期中，假设已预测分支并确定不发生分支，则正常按顺序取指。在图 3 中，应注意 exu_flush_final = 0, ifc_fetch_addr_f1_ext[31:0] = ifc_fetch_addr_f1_raw_ext[31:0] = 0x000001F0。该地址指向下一个连续的 128 位指令束。可以看出，在前两个周期中，之前的 128 位指令束已取指（ifc_fetch_addr_f1_ext[31:0] = 0x000001E0）。

任务: 修改图1，将图3的周期*i*、*i+1*和*i+2*中所示的每个信号的值包含在内。

任务: 修改图2中的程序，让第一条分支指令通过转发的形式检索其输入操作数。

ii. 第二条分支的执行: **beq t3, t3, OUT**

接下来我们分析第二条分支，系统总是错误预测为不发生该分支，但实际会发生该分支。打开反汇编文件（位于

[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/.pio/build/swervolf_nexys/firmware.dis）。

请注意，第二条**beq**指令位于地址0x000001E8处：

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

图4所示为循环中随机某次迭代期间的信号（应避免采用第一次迭代，因为该次迭代包含指令高速缓存（I\$）未命中）。

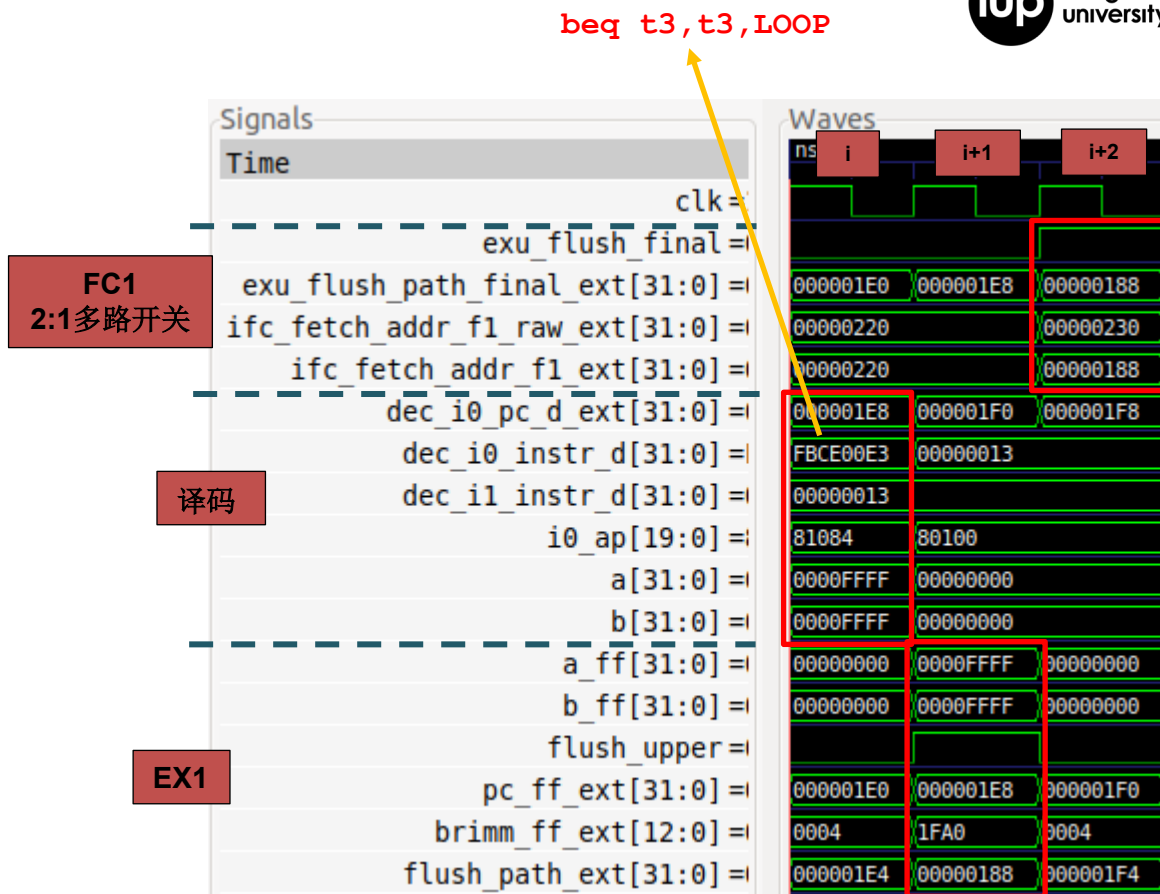


图4. 图2的示例中第二条分支的Verilator仿真

同时分析图4中的波形以及图1中的图。以红色突出显示的值表示执行第二条beq指令期间三个连续的周期：beq的译码阶段（周期*i*）、beq的EX1阶段（周期*i+1*）、在解析beq后的FC1阶段选择下一PC（周期*i+2*）。

- **周期*i* - beq指令的译码阶段：**PC（信号dec_i0_pc_d_ext）为0x000001E8，指令（信号dec_i0_instr_d）为0xFBCE00E3（二进制形式为：1111 1011 1100 1110 0000 0000 1110 0011）。

在RISC-V中，beq指令的操作码如下（参见[DDCARV]的附录B）：

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

因此，您可以验证0xFBCE00E3对应于：beq t3,t3,LOOP（Immediate_{12:0} = 0x1FA0）。请记住，立即数给出了目标地址当前PC的偏移量。目标地址（由标签“LOOP:”指示）是当前PC（即beq t3,t3,LOOP）之前的24条指令（7条nop + 1条add + 7条nop + 1条beq + 7条nop + 1条add = 24条指令）。当前PC之前共有24*4 = 96个字节。因此，立即数编码为-96，即0x1FA0，以13位二进制补码形式写入。

在此阶段，将产生流水线控制信号。此beq指令的控制信号与第一条beq指令的控制信号相同（参见上一部分）。

此外，将读取寄存器文件并将分支指令传送至I0管道。信号a和b（均为0xFFFF）中包含下一阶段所使用的比较器的输入，在本例中，输入为从寄存器文件读取的值。

- **周期*i+1* - beq指令的EX1阶段:** 在下一周期中，将执行beq指令。在一侧比较a_ff和b_ff信号。如果两个值相同，则必须发生分支。然而，如上文所述，在我们的配置中，所有分支的预测结果均为不发生（i0_ap.predict_nt = 1）。也就是说，分支预测结果错误（flush_upper = 1）。因此，必须从分支目标地址取指，并且必须清除初始流水线阶段。

在此阶段，目标地址的计算方法为将pc_ff_ext（0x1E8）与brim_ff_ext（0x1FA0）相加。结果将放入信号flush_path_ext（0x00000188）。

- **周期*i+2* - FC1阶段:** 在下一周期中，指令必须在分支目标地址处继续执行。如图4中所示，exu_flush_final = 1, ifc_fetch_addr_f1_ext = exu_flush_path_final_ext = 0x00000188。该地址对应于分支目标地址，即循环的第一条指令的地址（请注意，这是一个向后分支）。

任务: 修改图1，将图4的周期*i*、*i+1*和*i+2*中所示的每个信号的值包含在内。

任务: 根据图2中的示例，检查不同情况下的信号，分析FC1中两个多路开关的操作。

例如，分析在按顺序执行指令（即一组没有分支的指令）的情况下如何完成取指。在这种情况下，您将看到SweRV EH1处理器进行如下操作：

- 在偶数周期中，使用5:1多路开关选择fetch_addr_next，该多路开关包含的值为当前取指地址（ifc_fetch_addr_f1）+ 16，因此会读取下一个连续的128位指令束（请记住，I\$读操作提供128位）。

- 在奇数周期中，使用5:1多路开关选择ifc_fetch_addr_f1，因此不会取出新的指令。这样，每2个周期会取出4条32位指令，这与译码阶段所需的取指速率相同（每个周期2条指令）。

注意，在DDCARV所述的处理器中，每个周期只需将PC的值加4（适用于按顺序执行指令的情况），从而在每个周期取一条指令。

还可以修改图2中的程序以创建新的场景。例如，可以在跳转的分支后添加一些A-L指令，查看如何在重定向后清除这些指令。

任务: 在实验15中，我们已分析过如何在提交阶段通过辅助ALU消除写后读RAW数据冒险。与该实验探讨的A-L指令类似，如果先前执行过多周期操作，则条件分支指令可能产生原始数据冒险，必须在提交阶段解除冒险。如果确定分支预测错误，则必须在提交阶段清除流水线并重定向。请使用[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction_HazardCommit文件夹中的程序（对图2中的程序进行了少许修改）和.tcl文件分析该情况。

3. SweRV EH1使用的Gshare分支预测器

在第2部分，我们已讨论了仅包含一个简单分支预测器（预测结果始终为不发生）的SweRV EH1配置，本部分将继续分析SweRV EH1中提供的Gshare分支预测器的操作。Gshare BP会对每条分支指令执行更智能的预测，能够提高性能，但需要使用额外的硬件。在描述Gshare BP如何在SweRV EH1中工作之前，我们先比较两个BP的性能。

任务：在图2的示例中，删除所有nop指令并分析仿真。然后通过开发板上执行程序，用性能计数器计算IPC。

使能SweRV EH1中使用的分支预测器（方法为注释掉图2中的两条初始指令），并分析开发板上的仿真和执行情况。

比较两个实验并解释结果。

注：Scott McFarling曾在1993年发表过一篇题为“Combining Branch Predictors”的经典论文（<https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>）。该论文的第7章介绍了Gshare分支预测器的工作原理。也可搜索其他文档，如<https://people.engr.ncsu.edu/efg/521/f02/common/lectures/notes/lec16.pdf>。我们建议您在开始本部分之前先阅读这些资料，了解Gshare BP的工作原理。

图5所示为SweRV EH1中可用的Gshare BP的简化视图。所有BP结构均在模块ifu_bp_ctl（位于文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_bp_ctl.sv` 中）内部实现。图中，与Gshare BP相关的结构由蓝色虚线框圈出。

该BP由分支历史记录表（Branch History Table, BHT）和分支目标缓冲区（Branch Target Buffer, BTB）组成，前者预测分支的方向（发生或不发生），后者预测发生分支时的目标地址。在默认配置中，BHT包含128个2位条目。可在模块ifu_bp_ctl的第1615-1705行查看BHT。在默认配置中，BTB包含32个13位条目。可在模块ifu_bp_ctl的第1439-1613行查看BTB。

预测分支时，每个周期中将发生以下事件（参见图5）：

1. 通过模块ifu_bp_ctl内的几个哈希模块传递取指地址（`ifc_fetch_addr_f1[31:1]`）和一些其他信号：`flhash`、`rdtagf1`和`fghrhs`等。所有这些哈希模块均在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/beh/beh_lib.sv` 中实现，具体将使用 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh` 中定义的宏。

例如，可以看到fghrns模块会接收来自取指地址

(f1hash(.pc(ifu_fetch_addr_f1[31:1]),
.hash(btb_rd_addr_f1[`RV_BTБ_ADDR_HI:`RV_BTБ_ADDR_LO]))) 和fghr_ns
(全局历史记录寄存器)的哈希的btb_rd_addr_f1信号，并输出
bht_rd_addr_hashed_f1信号。

```
rvbtb_ghr_hash fghrns (.hashin(btb_rd_addr_f1[`RV_BTБ_ADDR_HI:`RV_BTБ_ADDR_LO]), .ghr(fghr_ns[`RV_BHT_GHR_RANGE]), .hash(bht_rd_addr_hashed_f1[`RV_BHT_ADDR_HI:`RV_BHT_ADDR_LO]));
```

该信号用于访问Gshare支路预测器的BHT表。

任务：分析上述所有哈希模块，尝试了解其工作原理及其在Gshare BP结构中的使用方式。

- 使用所有这些哈希信号（btb_rd_addr_f1、bht_rd_addr_hashed_f1和fetch_rd_tag_f1等）访问Gshare BP的两个主要结构：BHT和BTB。

任务：分析如何对这两个结构进行访问。

- 访问BHT后，可在信号ifu_bp_kill_next_f2中获得方向预测结果，如果预测不发生分支，则信号值为0；如果预测发生分支，则信号值为1。使用该信号与其他信号（在此不具体介绍）共同计算FC1中5:1多路开关的控制信号。

任务：分析如何计算5:1多路开关的选择信号。

- 访问BTB后，可在信号ifu_bp_btb_target_f2 [31:1]的加法器中获得所发生分支的预测目标地址。（注意，如果对ret指令进行预测，预测地址也可以来自返回地址堆栈（Return Address Stack, RAS）。）该信号为FC1中5:1多路开关的输入之一。

任务：分析如何通过BTB中读取的值（btb_rd_tgt_f2[11:0]）和FC2中的取指地址（ifu_fetch_addr_f2[31:4]）获得预测目标地址（ifu_bp_btb_target_f2）。

任务：分析SweRV EH1处理器中实现的RAS。可通过搜索互联网获得有关该结构的更多操作信息（如访问http://www-classes.usc.edu/engr/ee-s/457/EE457_Classnotes/ee457_Branch_Prediction/EE560_05_Ras_Just_FYI.pdf）。

- 在FC1的5:1多路开关中，如果ifu_bp_kill_next_f2 = 1，则预测目标地址将用作下次取指地址：fetch_addr_bf [31:1] = ifu_bp_btb_target_f2 [31:1]（需确保未清除流水线）。相反，如果ifu_bp_kill_next_f2 = 0，则使用其他四个输入之一作为下次取指地址。

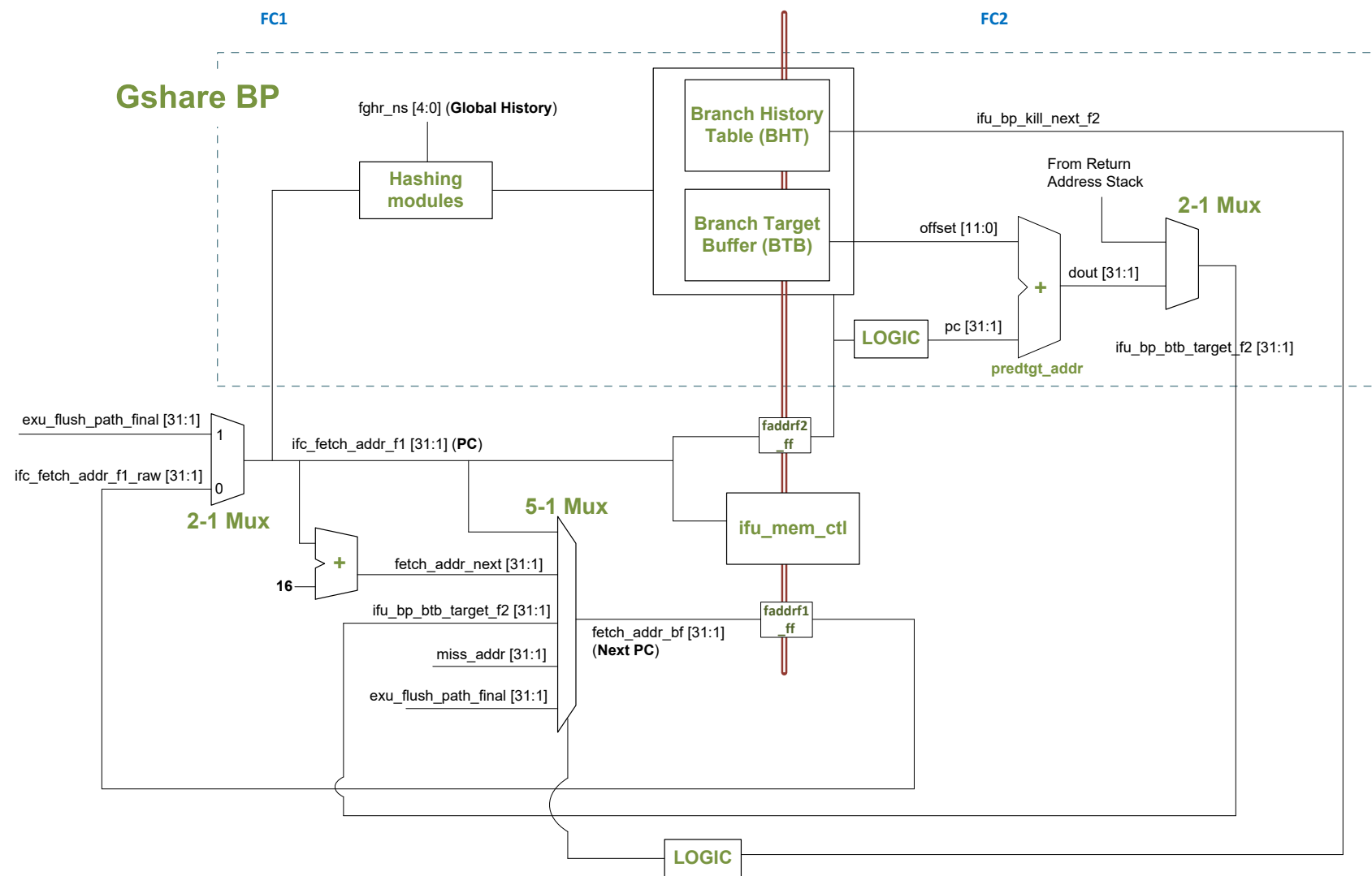



图5. SweRV EH1中提供的Gshare分支预测器的主要结构（由蓝色虚线框圈出）

在本部分中，我们将继续使用图2中的示例代码。本部分唯一的不同是，我们使用两条nop指令替换原有的两条指令（原有的指令会禁用Gshare BP），以使能Gshare分支预测器（插入两条nop指令是为了确保指令地址与上一部分相同）。

接下来分析程序中第二条分支指令的执行情况，如第2.B.ii部分所述。请记住，在我们的程序中，第二条beq指令位于地址0x000001E8处，这意味着该指令包含在地址范围0x1E0-0x1EF中所映射的128位指令束中：

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

图6为循环中随机某次迭代的放大图示。与往常一样，应避免采用第一次迭代，因为该次迭代包含I\$未命中。此外，对于此分支指令的第一次迭代，预测将会出错。图中包含的大多数信号为图5中所示的信号。该项目随附文件test_1_BP.tcl。要在GTKWave中使用该文件，请单击“File”（文件）→“Read Tcl Script File”（读取Tcl脚本文件），并打开文件[RVfpgaPath]/RVfpga/Labs/Lab16/BEQ_Instruction/test_1_BP.tcl。然后单击几次“Zoom In”（放大）（）移动至循环的任意一次迭代（第一次除外）。可以看到两条beq指令的执行情况；图6所示为执行第二条分支指令时应观察到的内容。

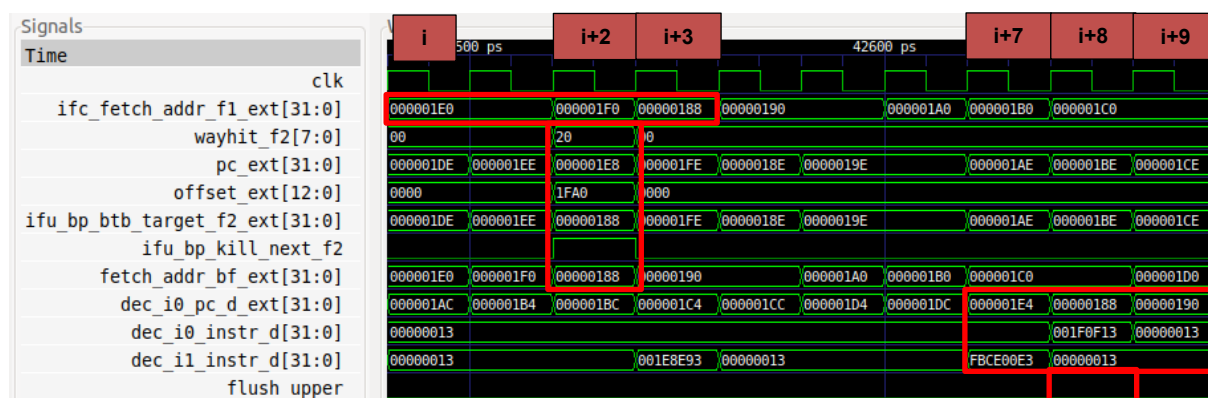


图6. 图2中示例的Verilator仿真

同时分析图6中的波形以及图5中的图。以红色突出显示的值对应于第二条beq指令，因为该指令遍历流水线阶段。

- **周期i:** 包含第二条分支的指令束的地址将提供给指令高速缓存：
ifc_fetch_addr_f1_ext = 0x000001E0。系统使用该地址读取分支目标缓冲区（BTB）。
- **周期i+2:** BTB中发生命中：wayhit_f2 = 0x20（该信号不包含在图5中，当信号不为零时，表示发生命中）。将分支地址（pc_ext = 0x000001E8）与BTB提供的偏移量（offset_ext = 0x1FA0，该值为负值）相加，即可得到预测目标地址（ifu_bp_btb_target_f2_ext = 0x00000188）。如果BHT预测将发生分支（ifu_bp_kill_next_f2 = 1），则预测目标地址将用作下次取指PC（fetch_addr_bf_ext = 0x00000188）。

- **周期*i*+3:** 取指地址为上一周期中计算的分支预测目标地址:
`ifc_fetch_addr_fl_ext = 0x00000188`。
- **周期*i*+7:** 在通路1中对分支进行译码 (`dec_i1_instr_d = 0xFBCE00E3`)。
- **周期*i*+8:** 执行分支。预测正确，因此无需触发清除操作 (`flush_upper = 0`)。
- **周期*i*+9:** 如果预测正确，将通过分支目标地址正常继续执行分支。

任务: 解释如何在模块 `ifu_bp_ctl` 处更新全局历史记录寄存器。

4. 练习

- 1) 实现一个双模分支预测器，并将其性能与Gshare BP的性能进行比较。

- 2) (以下练习基于《计算机组织结构和设计》(RISC-V版本, *Patterson & Hennessy* ([HePa])) 中的练习4.25。)

请看下面的循环:

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

假设采用了完美分支预测 (在SweRV EH1中, 只需避免使用第一次迭代即可模拟该行为), 流水线具有完备的转发支持 (同样是在SweRV EH1中), 并且可在EX1阶段确定分支执行情况。

- a. 展示该循环的第二次和第三次迭代的仿真结果。解释出现的行为。可使用 `[RVfpgaPath]/RVfpga/Labs/Lab16/HePa_Exercise-4-25` 中提供的程序。