哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2021 夏季

课程名称：计算机设计与实践

实验名称：CPU 设计

实验性质：综合设计型

实验学时：52 地点：T2

学生班级：7 班

学生学号：190110722

学生姓名：邹啟正

评阅教师：

报告成绩：

实验与创新实践教育中心制

2021 年 5 月

注：本设计报告中各个部分如果页数不够，请大家自行扩页，原则是一定要把报告写详细，能说明设计的成果和特色。报告中应该叙述设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

设计的功能描述（含所有实现的指令描述，以及单周期/流水线 CPU 频率）

单周期 CPU：

实现的指令：

add, sub, and, or, xor, sll, srl, sra, addi, andi, ori, xori, alli, aril, aria, lw, jalr, sw, beq, bne, blt, bge, lui, jal

CPU 频率：20MHz

流水线 CPU：

实现的指令：

add, sub, and, or, xor, sll, srl, sra, addi, andi, ori, xori, alli, aril, aria, lw, jalr, sw, beq, bne, blt, bge, lui, jal

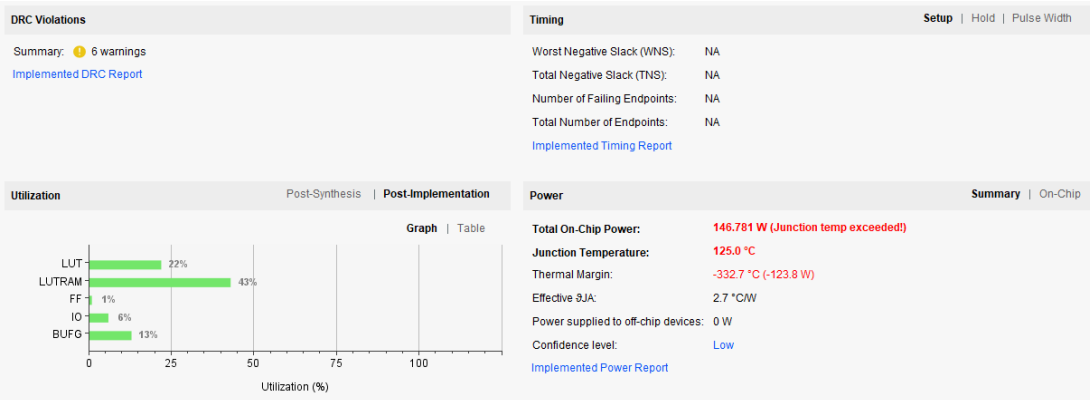
CPU 频率：50MHz

设计的主要特色（除基本要求以外的设计）

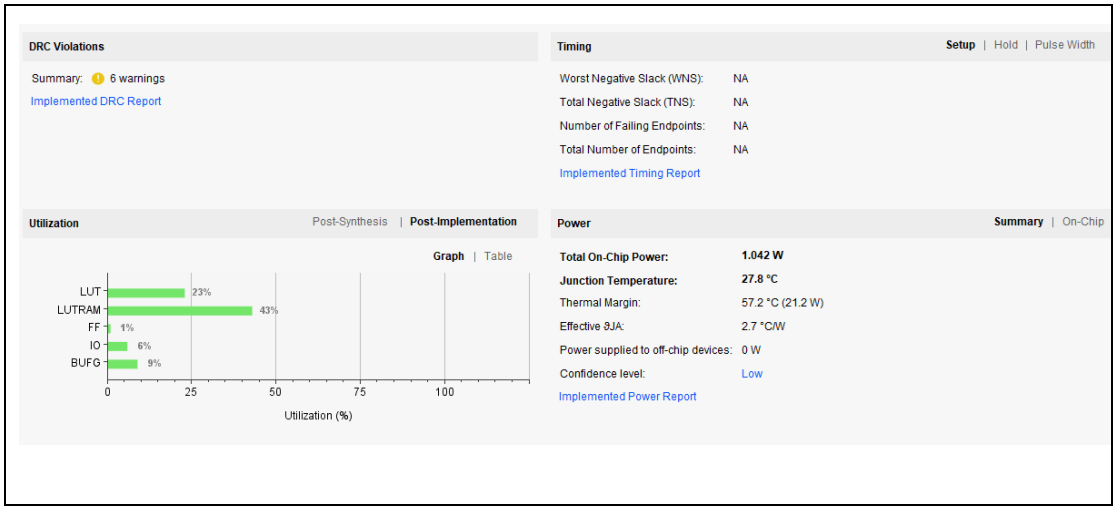
1. 流水线 CPU 实现了流水线暂停的功能，主要用于和前递配合解决 load_use 型数据冒险
2. 流水线 CPU 实现了数据前递的功能，用于解决普通的一级和二级数据冒险，以及辅助解决 load_use 型数据冒险
3. 流水线 CPU 实现了静态分支预测的功能，每当有跳转指令出现的时候总是预测不跳转
4. 流水线 CPU 实现了清除指令的功能，主要用于在静态分支预测失败的时候将 IF 和 ID 阶段正在执行的两条指令清除

资源使用情况、功耗数据截图（实现后）

单周期 CPU：



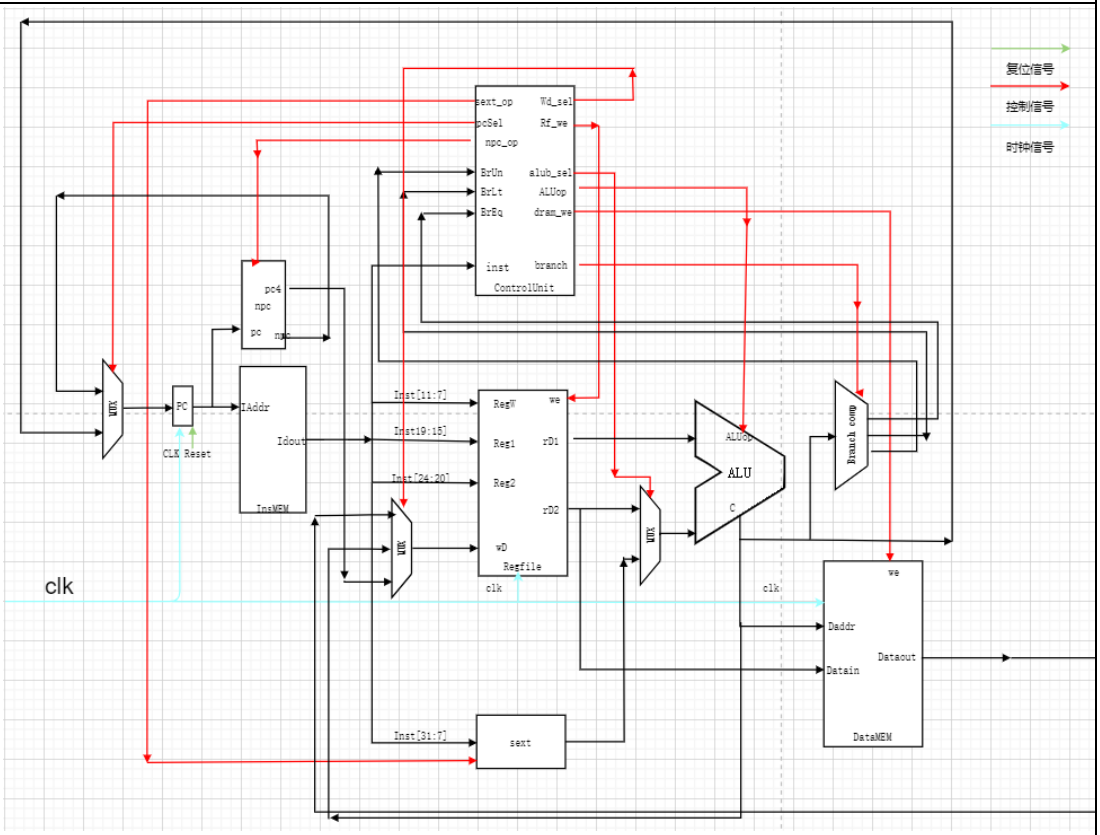
流水线 CPU：


DRC Violations
Summary: 6 warnings
Implemented DRC Report
Timing
Setup | Hold | Pulse Width
Worst Negative Slack (WNS): NA
Total Negative Slack (TNS): NA
Number of Failing Endpoints: NA
Total Number of Endpoints: NA
Implemented Timing Report
Utilization
Post-Synthesis | Post-Implementation
Graph | Table
LUT 23%
LUTRAM 43%
FF 1%
IO 6%
BUFG 9%
0 25 50 75 100
Utilization (%)
Power
Summary | On-Chip
Total On-Chip Power: 1.042 W
Junction Temperature: 27.8 °C
Thermal Margin: 57.2 °C (21.2 W)
Effective θJA: 2.7 °C/W
Power supplied to off-chip devices: 0 W
Confidence level: Low
Implemented Power Report

1 单周期 CPU 设计与实现

1.1 单周期 CPU 整体框图

（要求：无需画出模块内的逻辑，但要标出模块之间信号线的信号名和位宽，以及说明每个模块的功能含义）



每个模块的功能含义：

pc：

pc 是时序逻辑部件，是一个 32 位的指令地址寄存器，存储当前正在执行的指令的地址。

npc：

npc 是一个组合逻辑部件，主要用于计算下一条指令的地址，并且将计算的结果连到 pc 上去

imem：

imem 是一个组合逻辑部件，在哈佛架构中是指令存储器，是一个用于存放 32 位指令的 64KB 的 ROM，它支持数据的异步读写，也就是只要在输入端一给出地址，就可以立马在输出端得到相应的指令

cu：

cu 是一个组合逻辑部件，是控制单元，cu 接受并解析整条指令的格式，然后根据指令的类型来输出控制信号，控制各个部件的行为

reg_file：

reg_file 是一个包含了 32 个 32 位寄存器的寄存器堆，它是一个读的时候是组合逻辑，写的时候是时序逻辑的部件

sext:

sext 是一个组合逻辑部件，是立即数符号扩展部件，他接受整条指令，并且根据 cu 传过来的控制信号 sext_op 来判断要如何进行符号扩展

ALU:

ALU 是一个组合逻辑部件，是计算单元，ALU 接受两个操作数，然后根据 cu 传给它的 alu_op 控制信号来判断要对这两个操作数执行什么运算，可以执行的运算有 add，sub，or，and，xor，sll，srl，sra，仅仅输出第二个操作数等等。

Branch_cmp:

Branch_cmp 是一个组合逻辑部件，是分支判断单元，在执行分支跳转指令的时候 cu 传给 branch_cmp 的 branch 控制信号才会有效，此时它才会正常工作。在执行分支跳转指令的时候,cu 会控制 ALU 让他执行 sub 运算,Branch_cmp 接受两个操作数相减得到的结果，根据这个结果判断第一个操作数到底是大于，还是小于，还是等于第二个操作数，并且相应地把输出信号 GT，LT，EQ 给置位，发送给 cu。

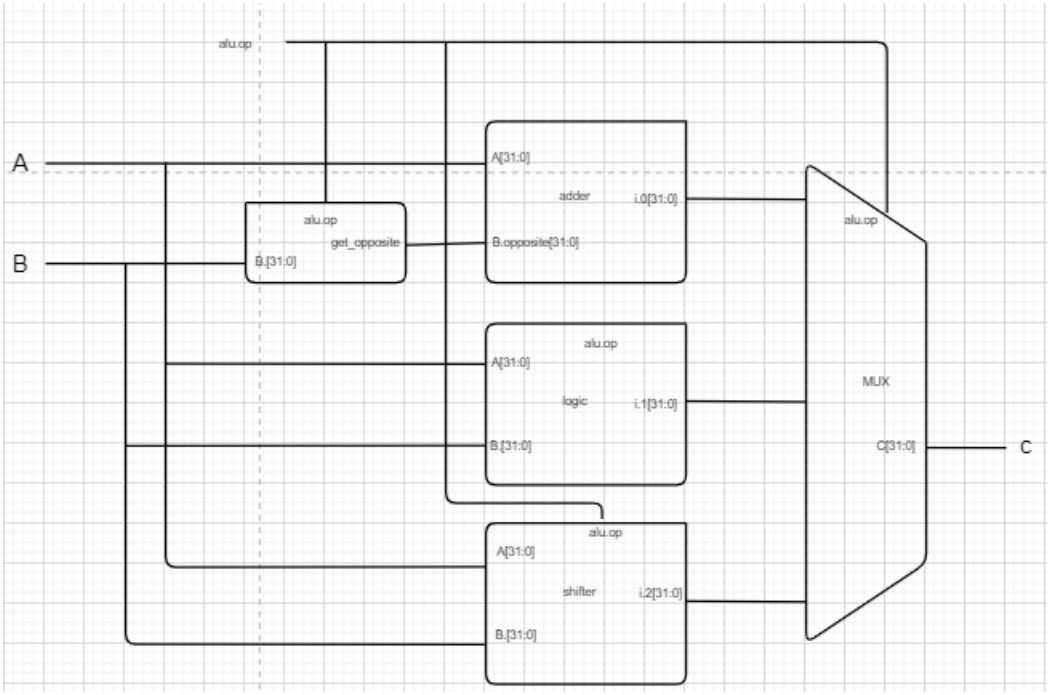
dmem:

dmem 是一个时序逻辑部件，在哈佛架构中是数据存储器，是一个存储 32 位数据的 64KB 的 RAM。dmem 读写数据的时候需要等待时钟沿的到来，而这里的时钟沿应该是下降沿。

1.2 单周期 CPU 模块详细设计

（要求：各个模块的详细设计图，要包含内部的子模块，以及关键性逻辑，标出信号名和位宽，并有详细说明）

ALU:



ALU 是运算单元，它的作用主要有三：执行加减法运算，执行逻辑运算，执行移位运算。ALU 执行运算功能的部件主要有三个，一个是执行补码加法运算的部件，一个是执行逻辑运算的部件，还有一个是执行移位运算的部件。同时在执行加法部件之前还有一个补码取反的部件 get_opposite，这个部件会把第二个操作数 B 取反，当执行减法指令的时候这个部件才会起作用，这样就可以只使用一个加法器就能够执行加减两种运算了。三个部件都会执行运算，并且输出结果，而最后会有一个 MUX，这个 MUX 会根据 alu_op 输出想要的结果

Sext:

Sext 是立即数扩展单元，sext 会根据指令的格式提取指令中的立即数，并且把立即数进行符号扩展

Reg File:

Reg File 是寄存器堆，里面有 32 个 32 位的寄存器，这些寄存器除了 x0 是只读并且值只能是 0 之外，其他的寄存器都是可读可写的。所有的寄存器都是读时逻辑，写时时序

1.3 单周期 CPU 仿真及结果分析

（要求：包含逻辑运算指令、访存指令、跳转指令的仿真截图，以及结果分析）

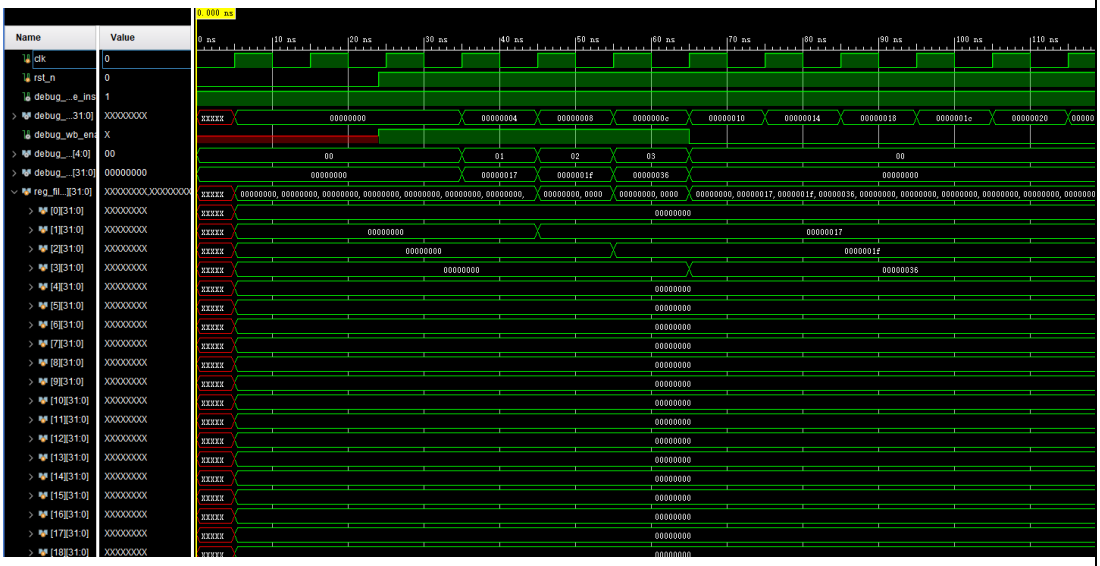
逻辑运算指令：

add 指令：

执行的指令如下：

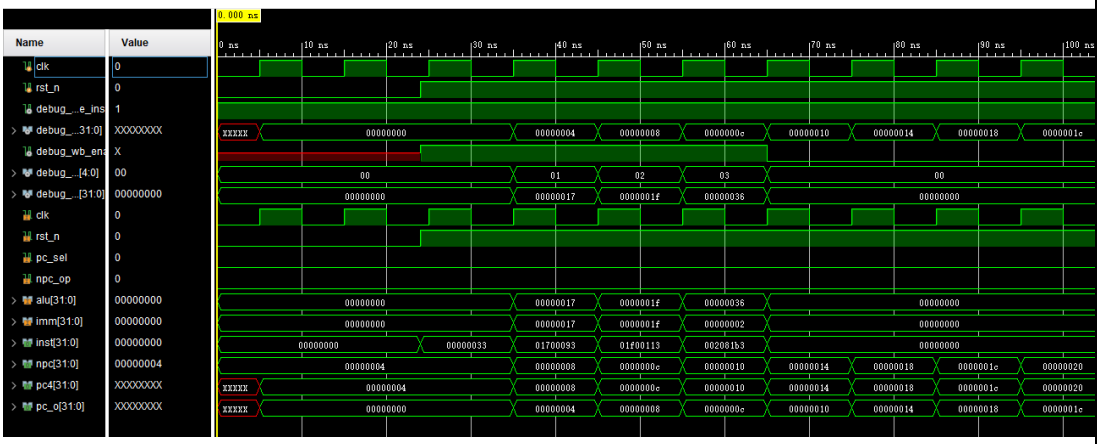
```
addi    x1, x0, 23
addi    x2, x0, 31
add      x3, x1, x2
```

前两条指令分别把 x1 寄存器和 x2 寄存器赋值为 23 和 31，第 3 条 add 指令把 x1 和 x2 的值加起来，存入 x3 中：



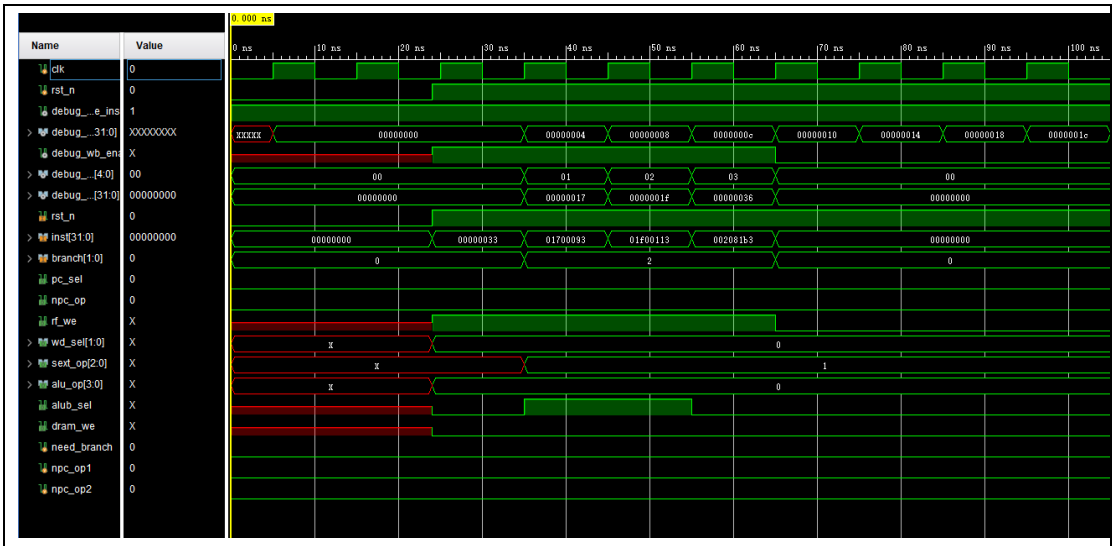
可以看到：这里的 x3 最后的值确实是正确的，是我们所期待的 23+31=54。

这条 add 指令首先会在 if 模块取值，把取出来的指令 inst 送到 id 模块：



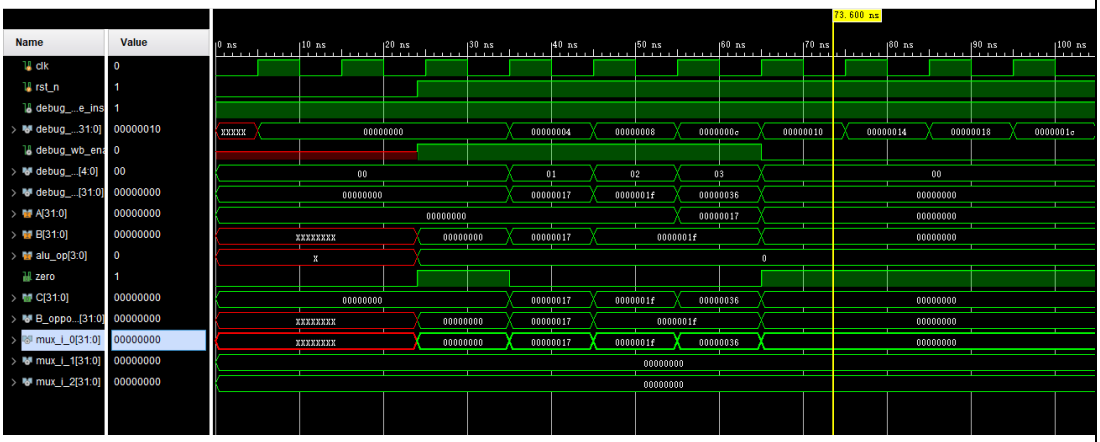
可以看到，此时的 02081b3 就是 add x3, x1, x2 的机器码的 16 进制形式。

id 模块得到了 inst 之后，会把 inst 送入到控制单元 cu 中：



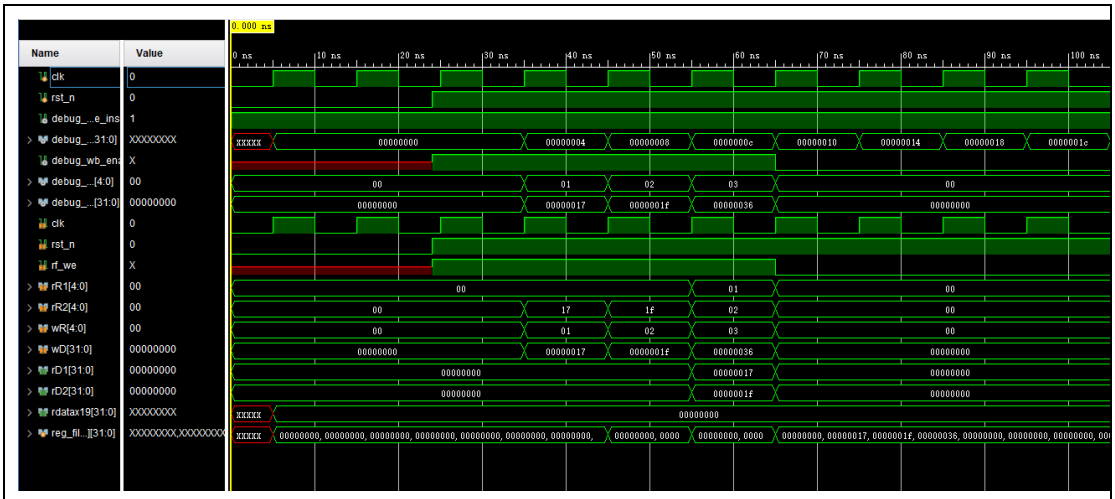
Cu 是一个组合逻辑部件，当它接受了 inst 的输入之后，会对 inst 的指令格式进行解析，把相应的控制信号进行赋值。在执行 add x3,x1,x2 的时候，cu 发现这是一条 R 型指令 add，于是把 rf_we 置为有效，表示寄存器堆写使能有效； 把 wd_sel 设置为 0，表示写入寄存器堆的数据是来自 alu 的计算结果；把 alub_sel 设置为 0，表示此时的 alu 的第二个操作数是来自第二个源寄存器；把 alu_op 置为 0，表示让 alu 执行 add 运算。

这些信号分别会连接到相应的部件上，控制这些部件的行为，比如 alu:



可以看到，执行 add x3,x1,x2 的时候 alu_op 是 0，第一个操作数 A 是 x1 寄存器的值 23，第二个操作数 B 是 x2 寄存器的值 31，那么最终得到的结果 C 就是 54

由于 add 指令是 R 型指令，没有访存阶段，所以 dram_we 是无效的，mem 模块并不工作。而最后到了写回的时候：

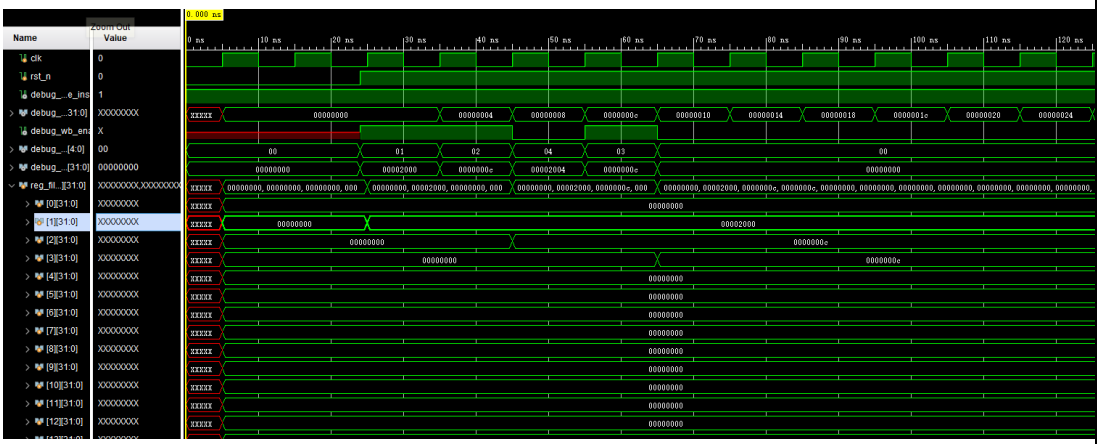


Rf_we 写使能信号有效，同时 wR 是 3，wD 是 alu 的计算结果，那么当时钟的上升沿到来的时候 alu 的计算结果 54 就会写入到 x3 寄存器中去。
至此，一条 add 指令执行完毕。

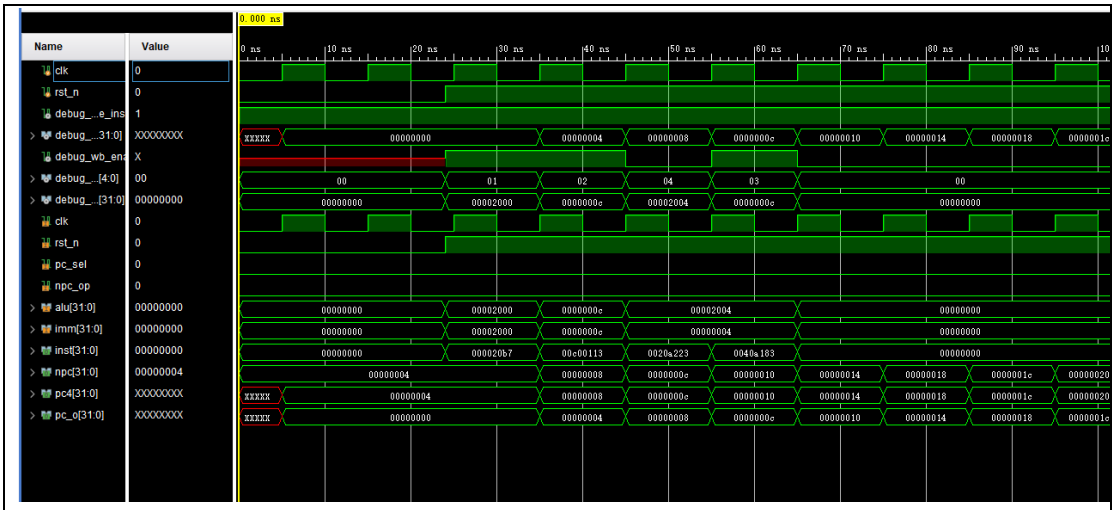
访存指令：
lw 指令：
执行的指令如下：

```
lui     x1,0x2
addi    x2,x0,12
sw      x2,4(x1)
lw      x3,4(x1)
```

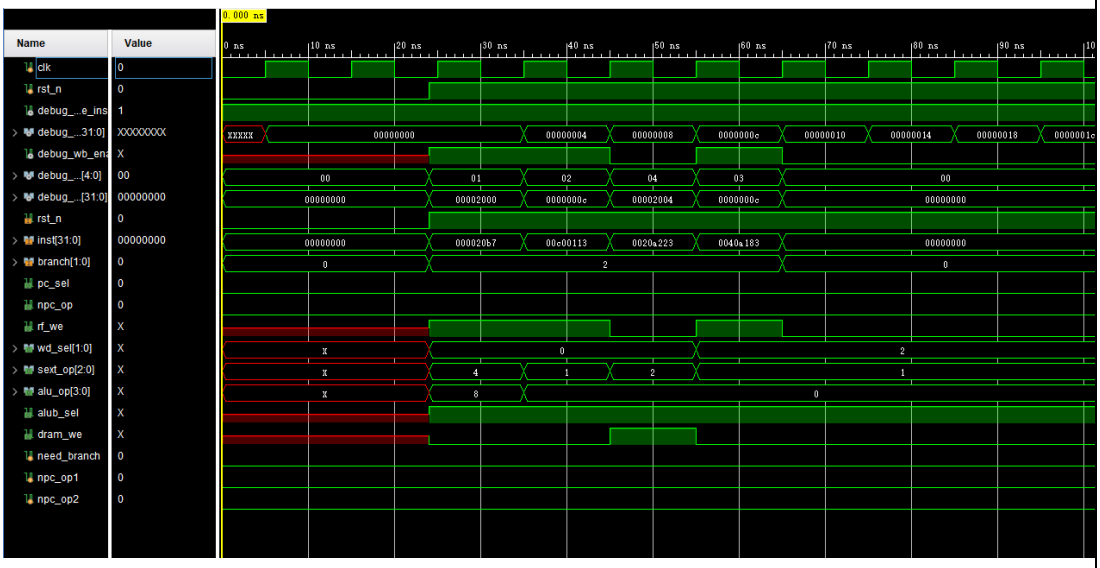
首先前三条指令把 12 这个数值放入了数据存储器中的 0x2004 中，然后第 4 句 lw 指令把数据存储器中的 0x2004 中的数据读出来存入 x3 寄存器中，所以最后 x3 的值是 12：



在执行 lw 指令的时候，首先需要在 if 模块中进行取值：

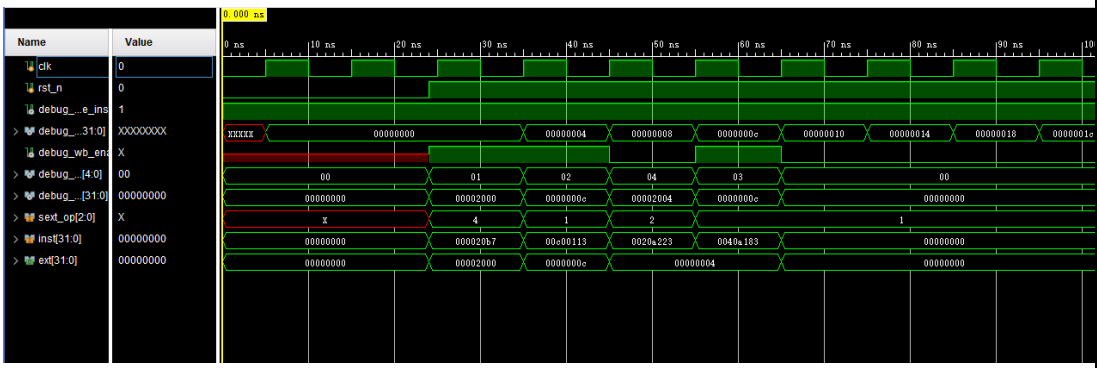


取出的指令 inst 是 0040a183，这是指令 lw x3,4(x1)的机器码的 16 进制形式。接着，这个信号会被送入 id 模块，id 模块的 cu 会对这个信号进行解析：



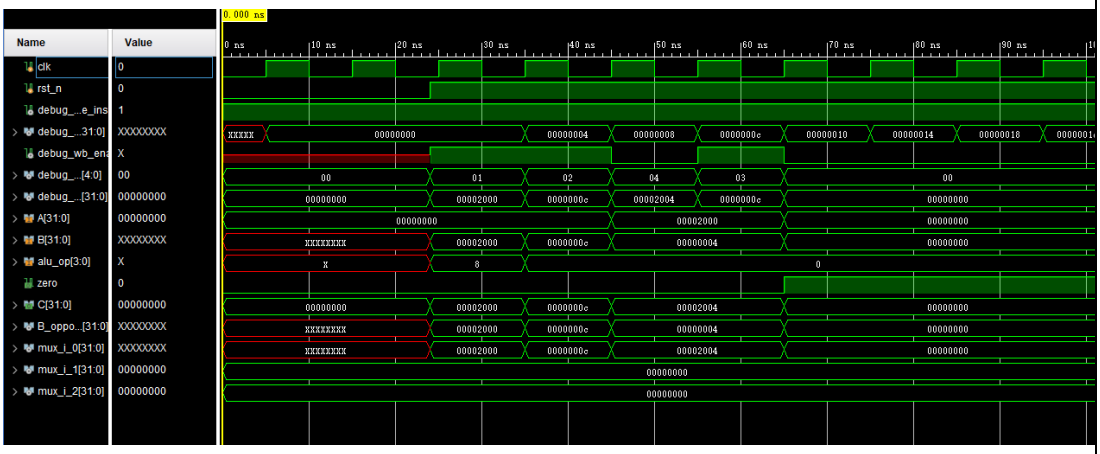
cu 会发现这是一条 I 型指令 lw，那么就会把 rf_we 置为有效，表示写回寄存器堆有效；把 wd_sel 置为 10，表示此时写回寄存器堆的数据是从数据存储器 data_mem 中读取出来的；sext_op 是 1，表示此时需要对 I 型指令进行立即数的有符号扩展；alu_op 为 0，表示此时 alu 需要执行 add 操作；alub_sel 是 1，表示此时的 alu 的第二个操作数是符号扩展之后的立即数。

这些信号会连接到相应的部件中，控制它们的后续行为，比如 sext：

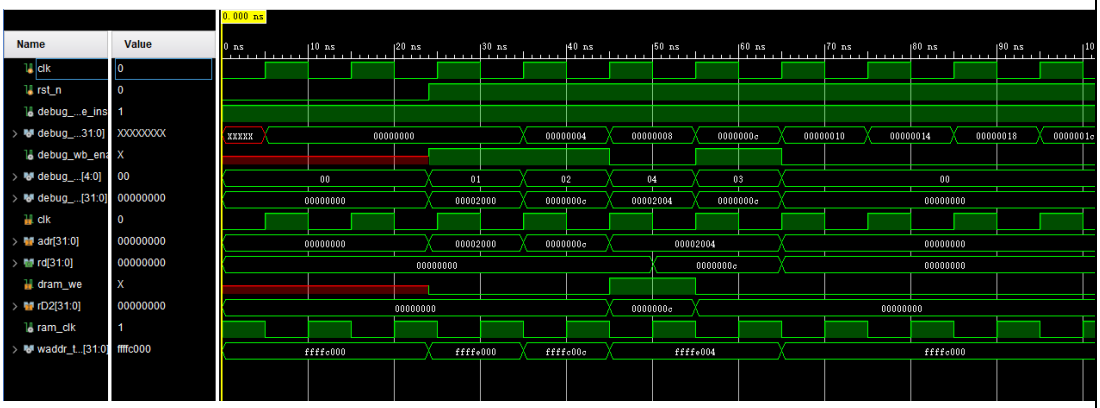


此时 sext_op 是 10，表示对 I 型指令进行立即数扩展，因此它会把输入 inst

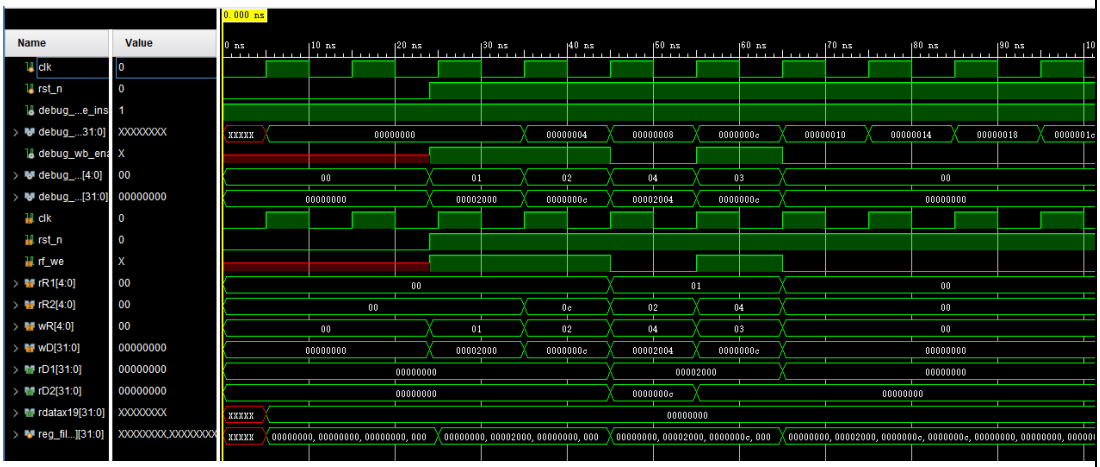
按照 I 型指令的格式把立即数的部分提取出来，并进行符号扩展，最终 4 进行符号扩展之后还是 4，并将 ext=4 输出。这个 ext 会被连接到 alu：



由于 alub_sel 为 1，因此 alu 的第二个操作数是来自 sext 的输出 ext=4，而第一个操作数是 x1 的值 0x2000，又因为 alu_op 是 0，因此 alu 执行 add 操作，把两个值相加就会得到最终的结果 C=0x2004。这个结果会连接到 data_mem 数据存储器作为地址输入：



可以看到，此时 data_mem 的输入地址 adr 是 0x2004，由于数据存储器是异步读取，因此一旦输入地址有效的时候，数据就会出现在输出端口 rd 上，这个 rd 会连接到寄存器堆中：



此时的 wR 是 3，wD 是刚刚数据存储器读取出来的值 12，而且写寄存器堆使能 rf_we 也是有效的，因此一旦时钟的上升沿到来，就会把 12 写入 x3 寄存器中。至此，一条 lw 指令执行完毕

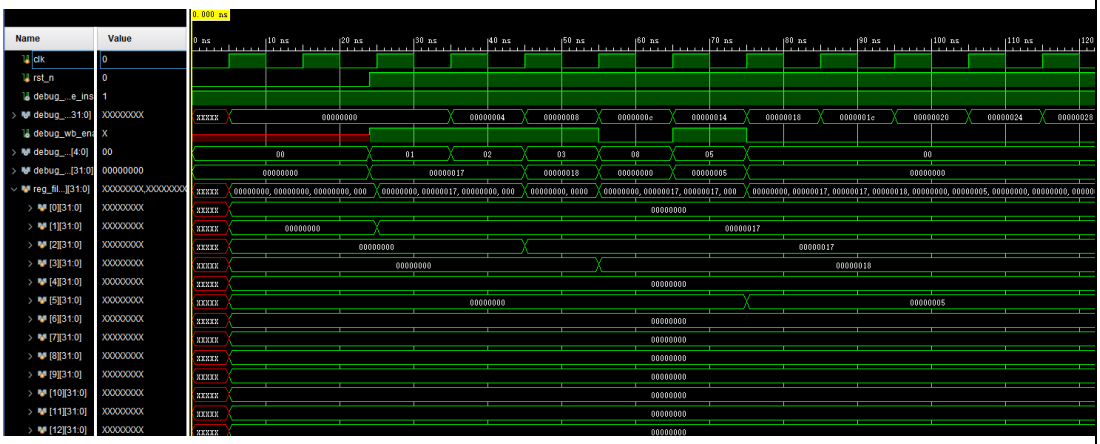
跳转指令：

beq 指令：

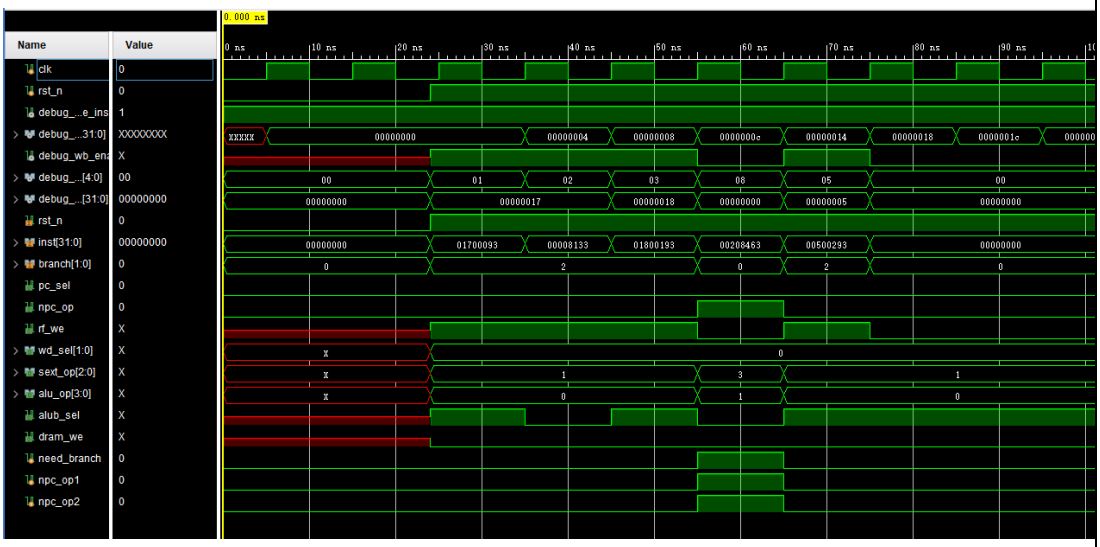
执行的指令如下：

```
addi    x1, x0, 23
add     x2, x1, x0
addi    x3, x0, 24
beq     x1, x2, A
addi    x4, x0, 4
A:
addi    x5, x0, 5
```

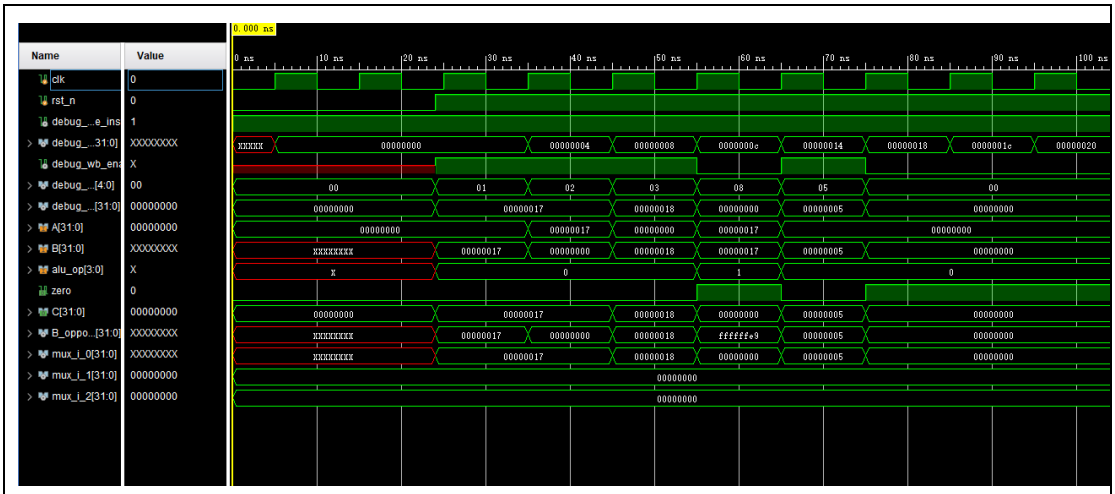
此时 x1 和 x2 是相同的，因此第 4 条 beq 指令会跳转，也就意味着执行结束之后的 x4 的值是 0，x5 的值是 5：



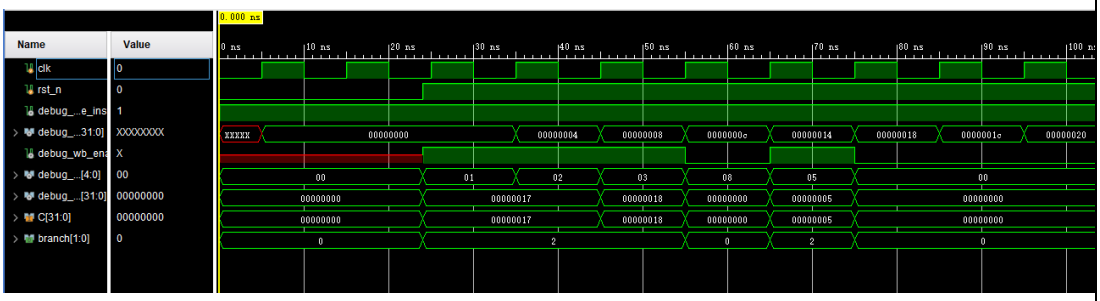
执行 beq 指令的时候首先也是取值，然后译码：



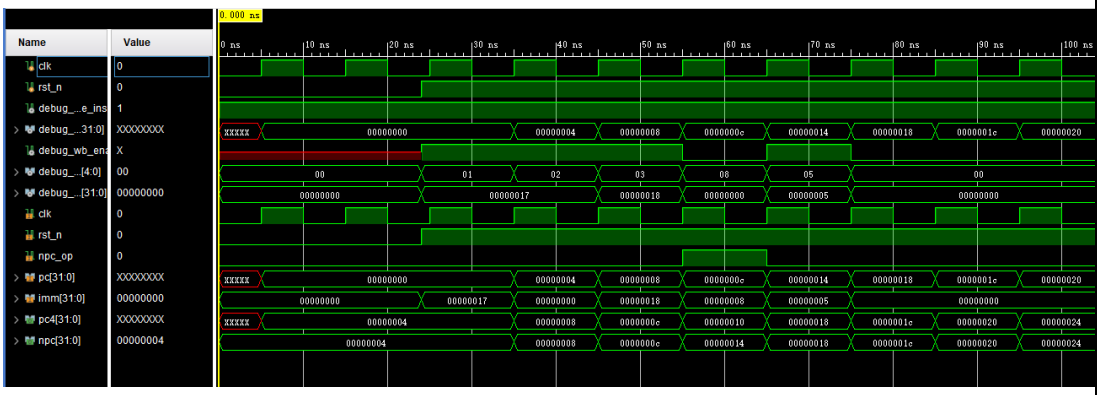
此时 cu 发现这是一条 B 型指令 beq，那么就会把 need_branch 置为 1，表示此时是跳转指令。同时 alu_op 会被置为 1，表示 alu 需要执行 sub：



Alu 执行 sub 操作之后会得到结果 C 是 0，那么这个 0 会被传入 branch_cmp 进行判断：



Branch_cmp 发现 sub 得到的结果是 0 的时候，就会把输出 branch 置为 0，表示此时比较的两个寄存器的值是一样的。那么这个 branch 会输出到 cu 中，cu 得到了这个 branch 之后，发现此时的 need_branch 是 1，同时又发现此时的 branch 表示两个寄存器的值相同，再结合 inst 的格式发现这是一个 beq 指令，于是就知道下一条指令需要跳转，于是把 npc_op 置为 1，表示下一条指令不再是顺序执行了，而是需要进行跳转的：



那么此时的 npc 输出的 npc 就不会是 0x10，而是 0x14。

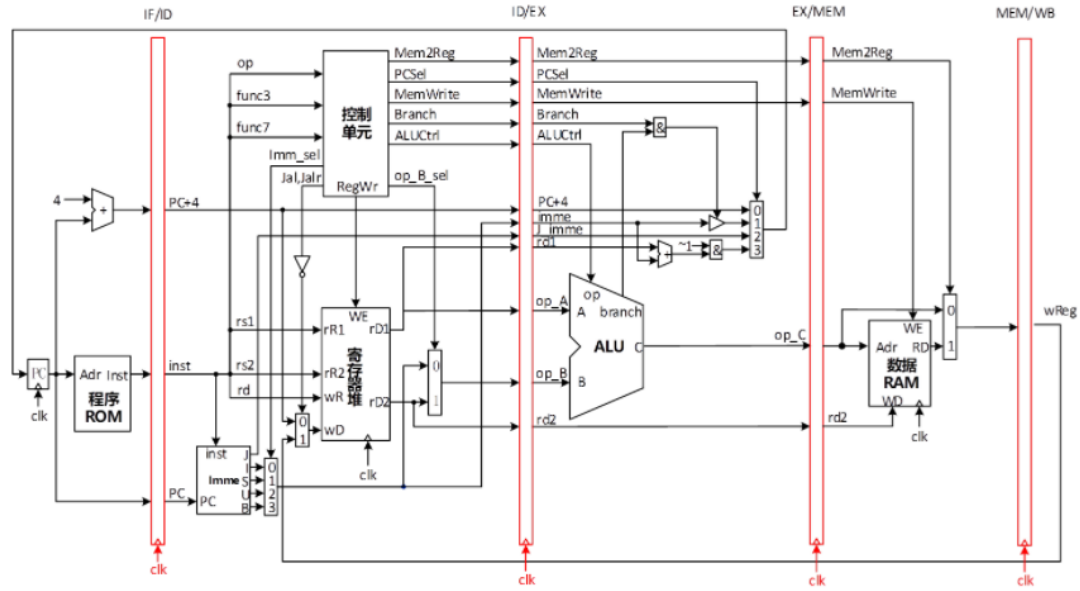
至此，一条 beq 指令执行结束。

2 流水线 CPU 设计与实现

2.1 流水线的划分

（要求：画出流水线的划分，并标明每个阶段 CPU 完成的功能）

流水线主要划分为五个阶段：IF（取指），ID（译码），EX（执行），MEM（访存），WB（写回）。



IF（取指）阶段：
取指阶段做的主要工作是在时钟上升沿的时候根据 pc 的值作为地址去指令存储器 inst_mem 中去取出相应的指令，送入 ID 阶段

ID（译码）阶段：
译码阶段做的主要工作有三：一，控制单元 cu 解析 IF 阶段传递过来的指令，然后赋值各个控制信号，把传给后面的各个阶段；二，立即数扩展单元 sext 获取 IF 阶段传递过来的指令，把指令中的立即数扩展成 32 位 ext，然后传递给 EX 阶段；三：解析 IF 阶段传递过来的指令，然后从里面解析出 rR1 和 rR2，根据这两个值从寄存器堆 Reg File 取出两个源寄存器的值传给 EX 阶段

EX（执行）阶段：
执行阶段做的主要工作是根据 ID 阶段传递过来的 rD1 和 rD2 还有 ext，然后根据传递过来的控制信号，控制 ALU 来对相应的操作数进行相应的运算

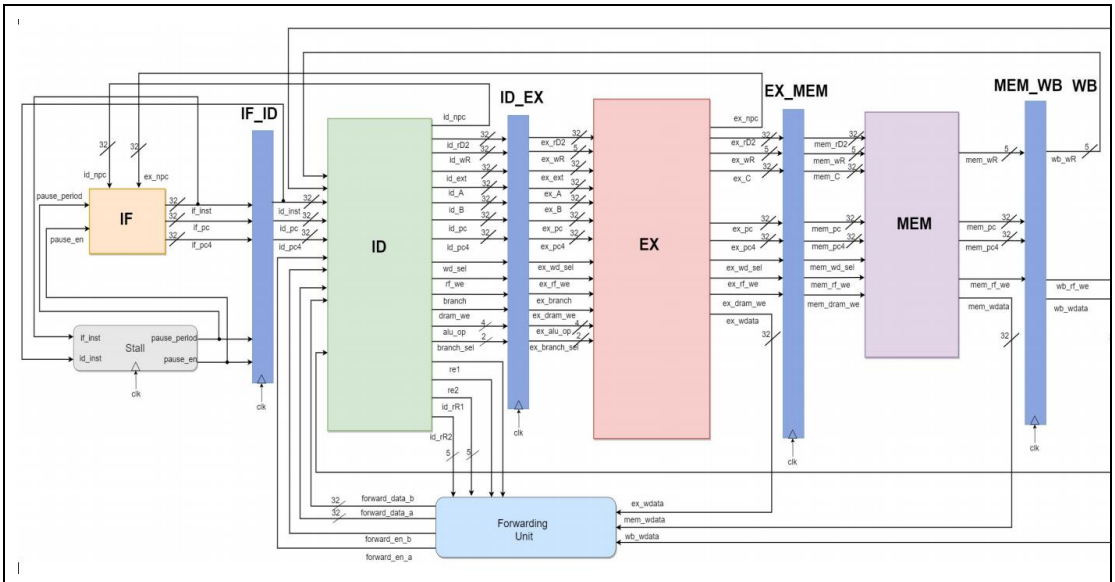
MEM（访存）阶段：
访存阶段做的主要工作是根据 EX 阶段传过来的数据和 ID 阶段穿过来的控制信号来判断是否需要进行访存，根据什么地址进行访存，是写入还是读出，读出到哪一个寄存器，写入的寄存器是哪一个等等

WB（写回）阶段：

写回阶段进行的主要工作是根据 ID 阶段传过来的控制信号判断是否需要写回，根据 EX 阶段传过来的寄存器号和需要写回的数据来判断应该写回到哪里，写回是写回到 Reg File

2.2 流水线 CPU 整体框图

（要求：无需画出模块内的逻辑，但要标出模块之间信号线的信号名和位宽，以及说明每个模块的功能含义）



IF：取指模块，从指令寄存器中取出指令 inst，放入 IF/ID 流水线寄存器。
ID：译码模块，根据 IF/ID 流水线寄存器中取出的指令进行解析，同时根据指令中的寄存器编号在寄存器堆中寻找相应的寄存器并取指，同时进行符号扩展，然后把所有这些信号送入 ID/EX 流水线寄存器。
EX：执行模块，根据 ID/EX 流水线寄存器传过来的寄存器值和立即数，根据 alu_op 进行相应的计算，把计算结果放入 EX/MEM 流水线寄存器。
MEM：访存模块，根据 EX/MEM 流水线寄存器传过来的数据和控制信号进行访存，并且把数据值传入到 MEM/WB 流水线寄存器。
IF/ID, ID/EX, EX/MEM, MEM/WB：流水线寄存器，用于存储各个流水线阶段运行的值。
Forwarding Unit 由 HU 和 FU 两部分组成。
HU：控制冒险检测单元，控制冒险单元总是会进行静态分支预测，每一次都预测跳转，它会根据控制信号 pc_sel 和 npc_op 的取指来判断静态分支预测是否成功，如果失败了，就将输出信号 flush 赋值为 1，将此时 IF 和 ID 阶段正在执行的指令给清空。
FU：前递单元，通过 ID，EX，MEM 阶段传过来的数据以及控制信号，判断是否需要进行数据前递。
STALL：停顿单元，如果出现了 load use 型数据冒险，那么就必须要使用停顿，停顿单元接受 FU 输出的 load use 信号，如果该信号为 1，那么就说明需要停顿。

2.3 流水线 CPU 模块详细设计

（要求：各个模块的详细设计图，要包含内部的子模块，以及关键性逻辑，标

出信号名和位宽，并有详细说明；数据冒险与控制冒险的解决方法必须要详细说明）

Forwarding_unit:

forwarding_unit 是前递检测单元，它接受当前 ID 阶段执行的指令的 rR1 和 rR2，同时也接受 EX 和 MEM 阶段执行的指令的 wR 和 rf_we。这个前递检测单元会进行判断，如果 EX 和 MEM 阶段的 rf_we 有效的话，就把 ID 阶段执行的指令的 rR1 和 rR2 和 EX 和 MEM 阶段执行的指令的 wR 进行比较，如果相同的话，那么就说明出现了 ID/EX 冒险或者 EX/MEM 冒险，把 rD1_sel 和 rD2_sel 置成相应的值。如果 ID/EX 冒险和 EX/MEM 冒险同时出现了的话，那么就选择执行 EX/MEM 冒险。

同时，这个 forwarding_unit 也会进行 load_use 型数据冒险的判断。Forwarding_unit 会接受 ex 阶段执行的执行的 opcode，如果此时 ex 阶段正在执行的指令的 rf_we 是有效，同时 wR 和 rR1 或者 rR2 相同，而且此时如果 opcode 显示该指令是 lw，那么就说明此时发生了 load use 型数据冒险，那么此时 forwarding _unit 就会把 load use 型数据冒险信号赋值并且输出到 stop_pipelined_unit，同时把 rD1_sel 或者 rD2_sel 赋值成 ID/EX 冒险。

Stop_pipelined_unit:
Stop_pipelined_unit 用于暂停流水线，该部件主要是在 load use 型数据冒险发生的时候和前递一起解决冒险的。它接受 forwarding_unit 输出的 load use 信号，如果这个信号有效，那么 stop_pipelined_unit 就会把输出信号 stop 置为有效，这个信号会传递给 IF 模块，让 IF 里面的 pc 的值不变，同时传递给 ID/EX 流水线寄存器，不让流水线寄存器的值改变，以此来达到暂停流水线一次的目的

hazard_detection_unit:
hazard_detection_unit 是控制冒险检测单元，由于我做的流水线 CPU 实现了静态分支预测，每一次都是预测分支不跳转，因此我们需要 hazard_detection_unit 来检测预测是否成功，这个 hazard_detection_unit 会接受两个控制信号 pc_sel 和 npc_op，如果这两个控制信号有任意一个不是 0 的话，那么就说明此时需要执行跳转，那么 hazard_detection_unit 就会输出 flush 到 IF/ID 和 ID/EX 流水线寄存器，把 IF 和 ID 阶段正在执行的两条指令给清空

流水线寄存器：
4 个流水线寄存器 IF/ID, ID/EX, EX/MEM, MEM/WB 都是时序逻辑部件，而且都是下降沿触发，当下降沿来临的时候，流水线寄存器就会把输入端的值存储到内部的寄存器中，并且进行输出。同时部分流水线寄存器还会连接 flush 控制信号和 stop 控制信号，如果 flush 或者 stop 信号有效，那么流水线寄存器就会把控制信号的所有使能信号置为无效，同时把流水线寄存器中的指令置为 add x0, x0, x0，这就就能够完成流水线的停顿和指令的清除

2.4 流水线 CPU 仿真及结果分析

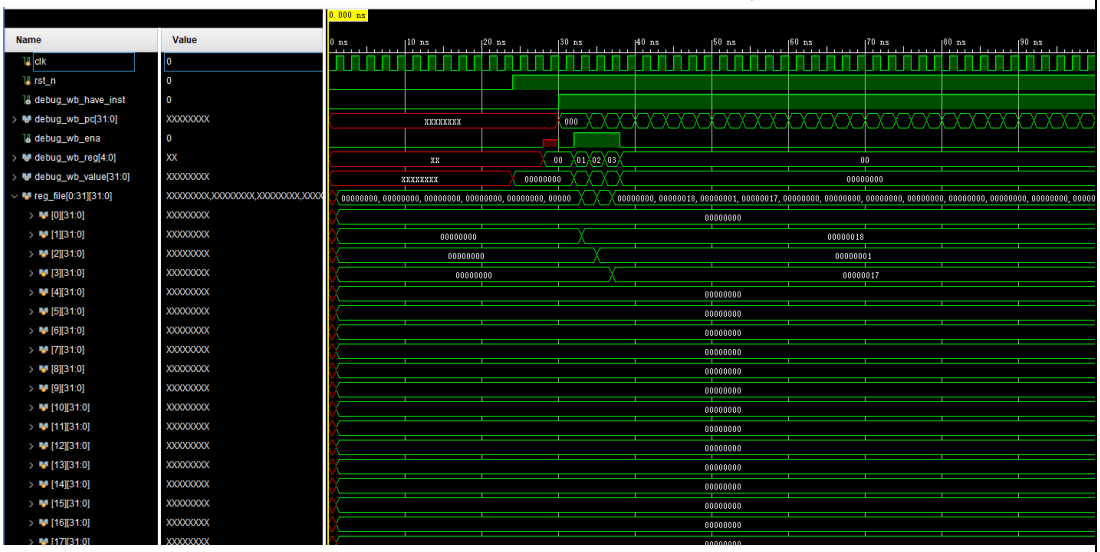
（要求：包含数据冒险、控制冒险的仿真截图，以及结果分析）

数据冒险：

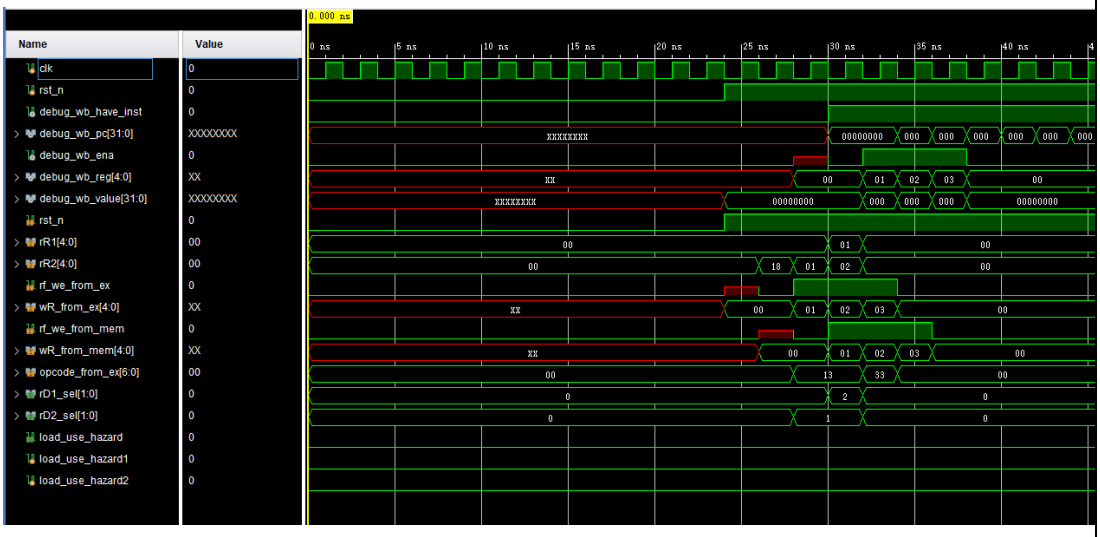
执行的指令如下：

```
addi    x1, x0, 24
addi    x2, x0, 1
sub     x3, x1, x2
```

可以看到，上述的代码中的第 1 行的目的寄存器是 x1，而第 3 行的第一个源寄存器就是 x1，因此会发生 mem/wb 数据冒险，而第 2 行的目的寄存器是 x2，第 3 行的第二个源寄存器是 x2，会发生 ex/mem 寄存器。而如果没有对数据冒险进行处理的话，由于复位的时候我们把所有的寄存器的值都设置为 0，那么结果就会是 x3 的值是 0。而我们处理了数据冒险之后，结果就是：

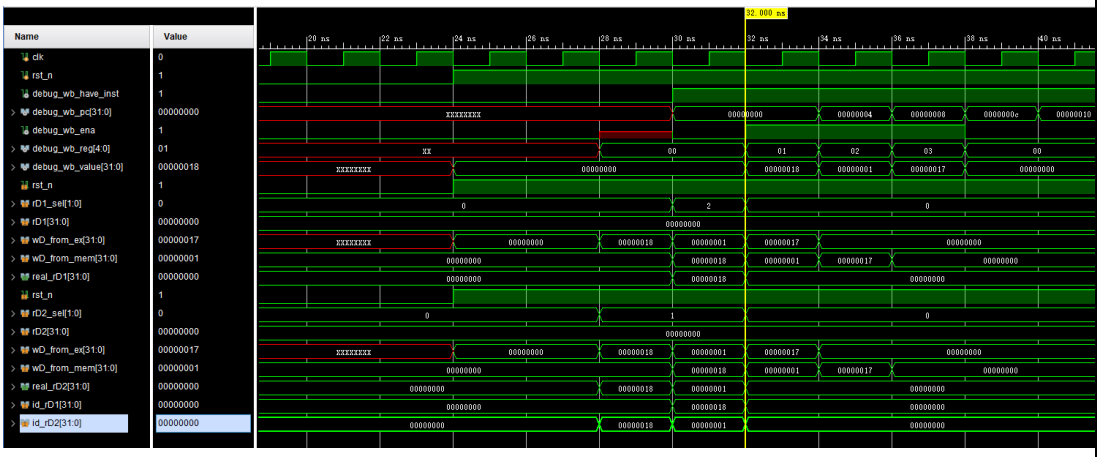


可以看到：x3 的值是正确的，也就意味着我们对数据冒险的处理是成功的。处理数据冒险的核心部件是位于 id 阶段的前递单元 forwarding_unit，它的仿真信号如下：



Forwardind_unit 接受 id 阶段正在执行的指令的 rR1 和 rR2，id/ex 流水线寄存器和 ex/mem 流水线寄存器输出的 rf_we（也就是寄存器堆的写使能信号）和 wR（也就是要写回的目的寄存器编号）作为输入，输出 rD1_sel 和 rD2_sel 信号来判断此时 id 阶段正在执行的指令的 rR1 和 rR2 有没有发生数据冒险，如果发生了，那么发生的是什么数据冒险。这两个输出会接到 id/ex 流水线寄

存器的 rR1 和 rR2 的输入的多路选择器 mux 中作为选择信号。此时执行前两条指令的时候，由于没有发生数据冒险，因此 rR1_sel 和 rR2_sel 都是 0，表示没有发生数据冒险，而到了第 3 条指令的时候，由于发生了两个数据冒险，因此 rR1_sel 的值是 10，表示发生了 mem/wb 数据冒险，而 rR2_sel 的值是 01，表示发生了 ex/mem 数据冒险，那么此时这两个输出就会传到 id/ex 流水线寄存器的 rR1 和 rR2 的输入的多路选择器 mux 中作为选择信号：



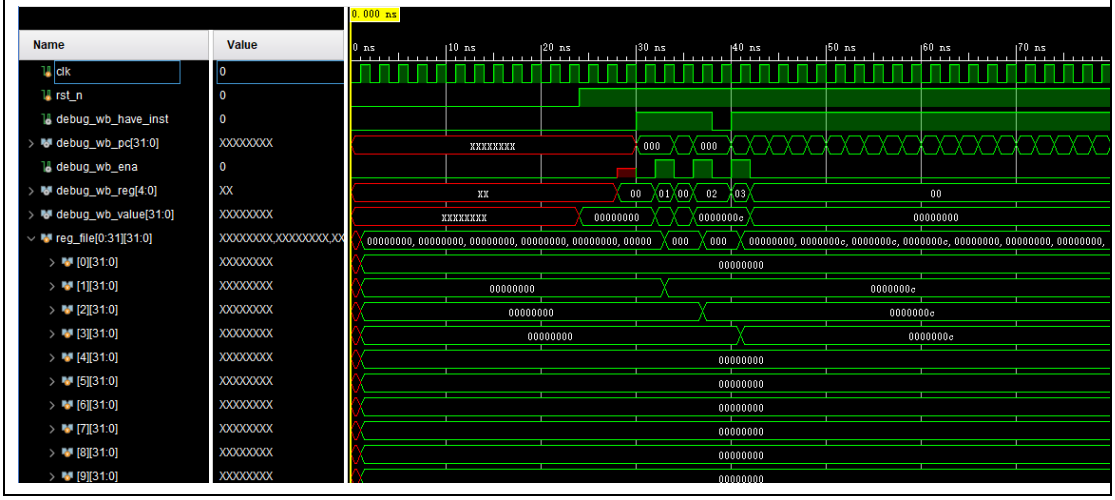
可以看到，rD1_mux 和 rD2_mux 的选择信号都被正确赋值了，所以 id/ex 流水线寄存器的输入 id_rD1 和 id_rD2 的值也都分别变成了 id/ex 流水线寄存器的 wD（就是写入目的寄存器的值）和 ex/mem 流水线寄存器的 wD 了，至此，前递很好地解决了 ex/mem 和 mem/wb 数据冒险

Load use 型数据冒险：

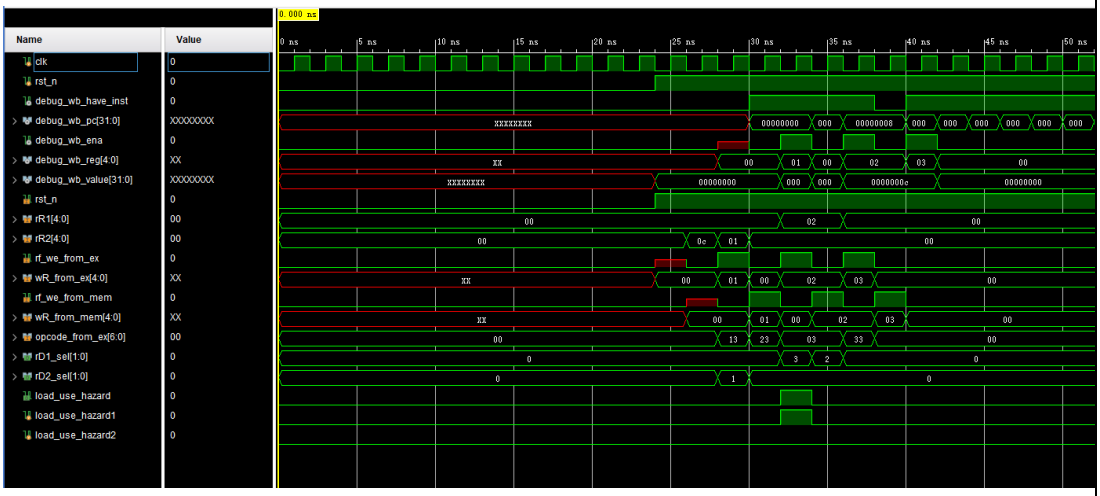
执行的指令如下：

```
addi    x1, x0, 12
sw      x1, 0(x0)
lw      x2, 0(x0)
add     x3, x2, x0
```

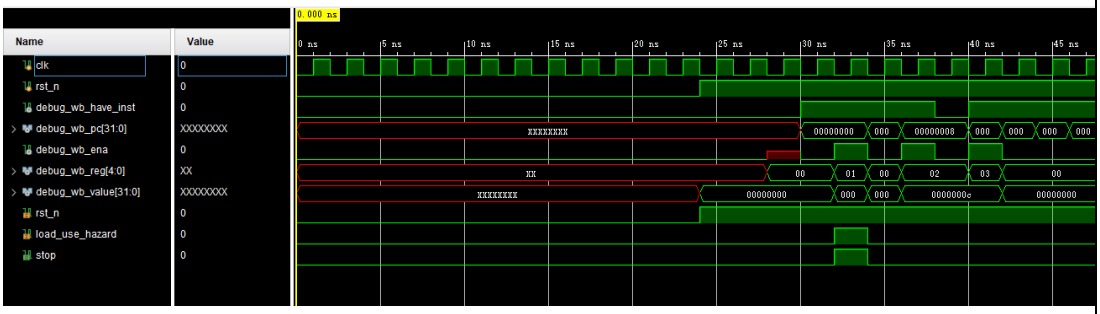
此时第 3 句指令是 lw 指令，目的寄存器是 x2，而下一句第 4 句指令就用到了 x2 作为源寄存器，此时就会发生 load use 型的数据冒险，如果不进行处理的话，由于数据存储器的初始值都是 0，那么最终的 x3 的值会是 0，但是我们此时进行了 load use 型数据冒险的处理之后 x3 的值就是：



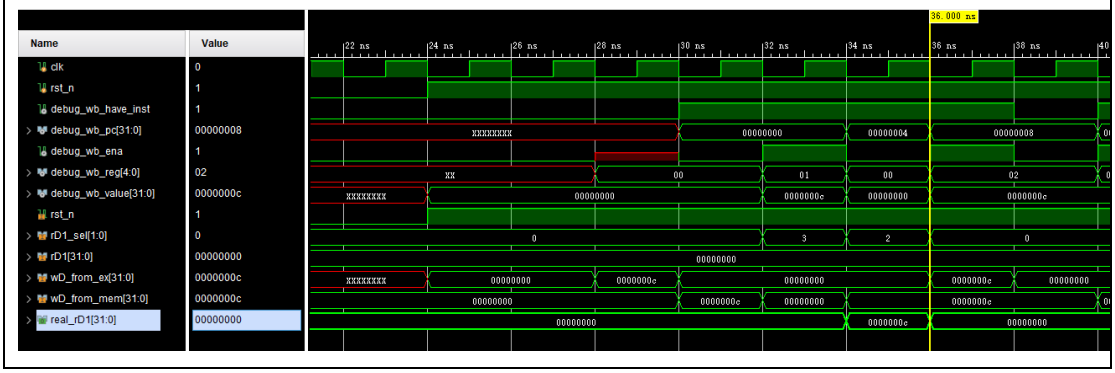
处理 load use 型的数据冒险需要用到前递单元 forwarding_unit 和流水线暂停单元 stop_pipelined_unit 的配合。我们先看 forwarding_unit 的仿真波形：



可以看到，forwarding_unit 通过输入 opcode_from_ex（就是 id/ex 流水线寄存器输出的 opcode）得到此时 ex 中执行的是一条 lw 指令，同时根据输入 wR_from_ex 得到了这条 lw 指令的目的寄存器的编号是 x2，同时输入 rf_we_from_ex 又是有效的，而且发现此时正在 id 阶段执行的指令（也就是地 4 条指令）的第 1 个源寄存器 rR1 是 x2，那么此时就会发生数据冒险，此时的 forwarding_unit 的输出 load_use_hazard 就是 1，这个信号会连接到 stop_pipelined_unit，让流水线暂停一个周期：



（这里的 stop 信号会传入控制单元 control_unit，让流水线暂停一个周期）同时此时的 rR1_sel 就是 11，表示发生了 load use 型数据冒险，需要前递，这个信号会连接到 rD1_mux 作为选择信号：

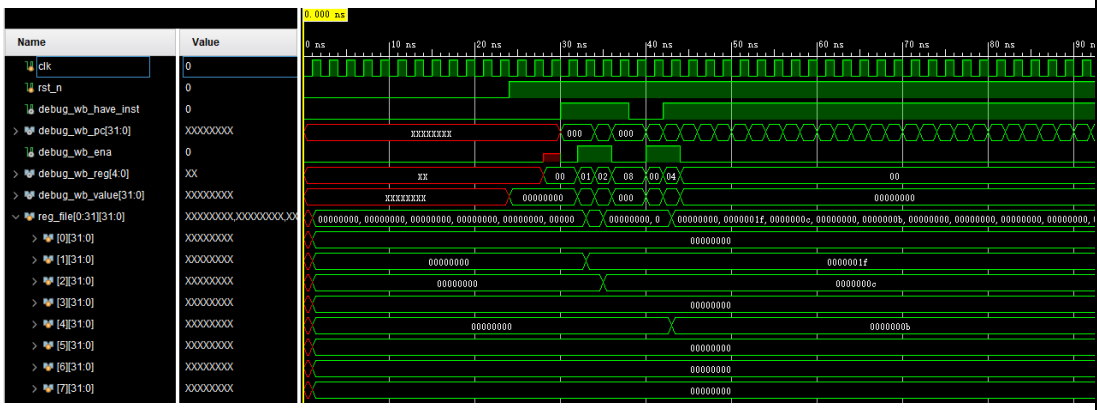


此时 rD1_mux 的选择信号获取了 11，进行了前递，把 mem 阶段输出的值赋值给了真正的 rD1。
这样，前递和流水线暂停就共同完成了对 load use 型数据冒险的处理

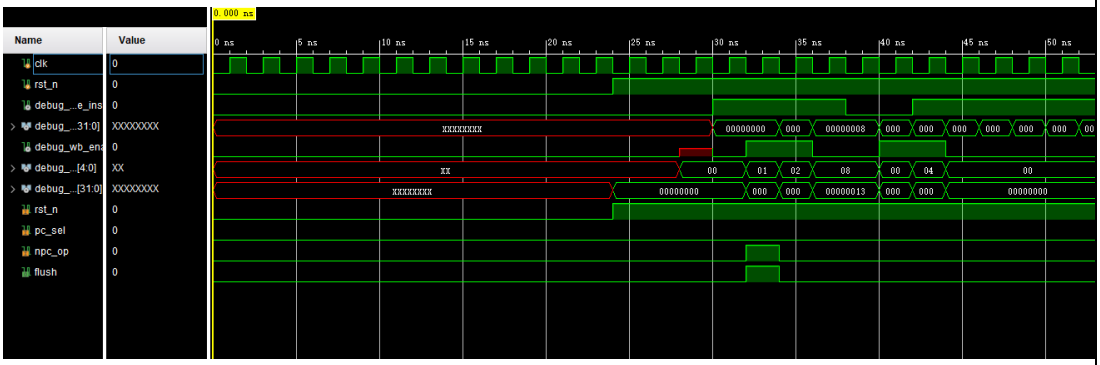
控制冒险：
执行的指令如下：

```
addi    x1, x0, 31
addi    x2, x0, 12
bne      x1, x2, J
addi    x3, x0, 11
J:
addi    x4, x0, 11
```

执行到第 3 条指令的时候遇到了分支指令，此时就会发生数据冒险，二如果你不进行处理的话，那么最终的结果就是 x3 和 x4 的值都是 11。正确的结果应该如下：



执行到第 3 条指令的时候 x1 是 31，x2 是 12，两者的值是不同的，因此此时的 bne 指令应该是要跳转的，但是由于我实现了静态分支预测，总是预测不进行跳转，而此时的预测失败了，需要进行指令的清空。而检测分支预测是否成功和负责输出清空信号的部件是 hazard_detection_unit：



Hazard_detection_unit 接受 pc_sel 和 npc_op 作为输入，当这两个输入都是 0 的时候表示此时的下一条指令的 pc 值是当前指令的 pc 值加上 4，也就表示此时的程序并没有发生跳转，是顺序执行的，因此此时不用进行指令的清空；而一旦这两个输入有任意一个是 1 的话就说明此时的下一条指令的 pc 值不再是当前指令的 pc 值加上 4，也就意味着程序进行了跳转，那么也就意味着此时的静态分支预测失效了，那么此时 flush 信号就会输出为 1，这个 flush 信

号最终会连接到 id/ex 和 if/id 流水线寄存器中：

一旦这两个流水线寄存器接收到了 flush 是 1 的话，那么一方面就会把流水线寄存器中所有的使能信号置为无效，另一方面会把当前的流水线寄存器中的指令设置为 add x0, x0, x0，达到清空指令的效果。

这样就很好地解决了控制冒险的问题

3 设计过程中遇到的问题及解决方法

（包括设计过程中的错误及测试过程中遇到的问题）

1、 在写单周期的时候，疏忽大意未声明信号、位宽写错，在 CPU 顶层文件编写时，因实例化多个模块，变量繁杂，一时疏忽导致在模块连线过程中出现了未声明信号、位宽本为 5 位最终写为 1 位，导致出现错误结果。解决方法：利用给出的错误信息，仿真波形，以及测试代码进行追溯，按照指令逐条检查、修改错误代码。

2、 刚开始进行流水线 CPU 的代码编写工作的时候，所有的 debug 都是自己手写汇编代码，然后生产机器码之后导入 vivado 工程里面的指令存储器 IP 核进行调试。等我把所有的指令调试通过，进行 trace 也是没有问题的，所有选做的指令也全都通过了，接下来我又按照实验指导书的指导进行了上板验证，也是没有问题。但是之后当我想要再次跑 trace 的时候却发现 trace 的 lw 指令突然就过不了了。我刚开始以为是 IP 核的原因，可是 IP 核是虚拟机自带的，应该不会出问题。后来我才发现原来是我上板的时候在数据存储器那里加了一句

```
wire [31:0] waddr_tmp = waddr - 16'h4000;
```

而当时之所以需要加这句话是因为下板测试使用的汇编程序采用的是 IROM 和 DRAM 统一编址，因此需要对原本的 DRAM 访存地址进行修改。而 trace 的时候是不需要的。

4 总结

（要求：个人收获以及对课程的建议）

个人收获：

在近三周的设计、代码编写以及调试后，终于成功完成了单周期、流水线 CPU 的设计，成功通过了 trace 比对与上板验收，夏季学期的计算机设计与实践课也算告一段落了。回望三周以来，实验内容其实是充满挑战性的。一开始回顾计算机组成原理中单周期知识，温习每条指令的内容，从而设计了数据通路与控制信号，这对后面编写代码是至关重要的基础。同时，我也进一步体会到“先框图后代码”的重要性，只有写代码前理清信号之间的关系，理清设计逻辑，才能编写起来又快又好。否则，只会陷入在编写时出现信号输入接错，想到一个时序逻辑再补输入信号的情况，这都是不严谨不规范的。我发现，自己对 verilog 代码的 debug 能力还是相当有欠缺的，三周时间对 verilog 编写能力增进不少，不过以后可能也很少写 verilog 了吧。

总之还是非常非常感谢各位老师的认真负责、悉心指导的。