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

# 实验报告

开课学期：2022 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：T2608

学生班级：2001110

学生学号：200111028

学生姓名：伍文浩

评阅教师：

报告成绩：

实验与创新实践教育中心制

2022 年 7 月

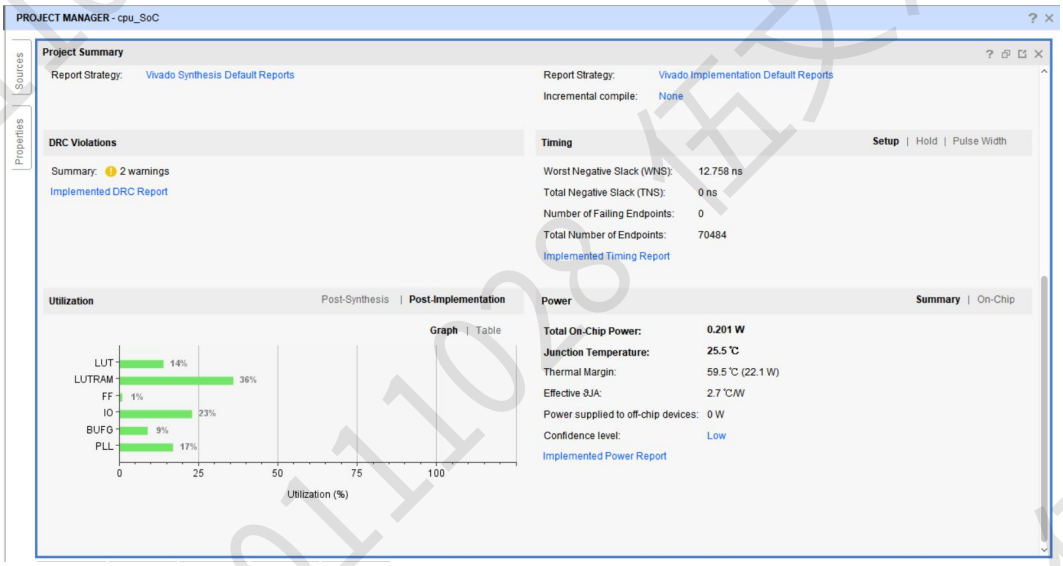
注：本设计报告中各个部分如果页数不够，请同学们自行扩页。原则上一定要把报告写详细，能说明设计的成果、特色和过程。报告应该详细叙述整体设计，以及设计中的每个模块。设计报告将是评定每个人成绩的重要组成部分（**设计内容及报告写作**都作为评分依据）。

| 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） |
| --- |
| 在这次实验中，我设计实现了单周期和流水线 CPU。它们支持运行 24 条基本指令.<br>R 型：and, add, srl, sw, sra, sll, or, sub, xor.<br>I 型：addi, subi, andi, ori, xori, ssli, srli, srai, lw, jarl.<br>B 型：bne, beq, blt, bge.<br>S 型：sw.<br>U 型：lui.<br>J 型：jal.<br>单周期 CPU 运行在频率为 25MHz 的时钟下，流水线运行在频率为 80MHz 的时钟下。<br>流水线 CPU 完成了对数据冒险、加载-使用冒险和控制冒险的处理。 |
| 设计的主要特色（除基本要求以外的设计） |
| 1. IO 总线<br>在单周期 CPU 和流水线 CPU 中使用了 IO 总线连接 CPU 和外设，对 CPU 而言外设和内存统一成了一个整体，可以使用一根地址线、一根数据线、一条使能信号管理。<br>2. 数据前递<br>使用前递机制解决三种数据冒险。并且当这三种数据冒险同时发生时，可以按照正确的优先级处理转发。<br>3. 流水线暂停<br>使用暂停解决加载-使用冒险和分支控制。当我们在执行阶段检测到加载使用冒险时，通过暂停取值和译码阶段，使得译码指令延缓到访存结束后执行，从而解决加载使用冒险。<br>4. 分支处理<br>(1) 对于 B 型指令，始终预测不跳转（也就是选择 PC+4）。直到执行阶段检查到发生跳转时，通过清除访存和译码阶段的指令，并重写 PC 消除影响，如果检测到未跳转则继续执行。<br>(2) 对于跳转指令，我们在执行阶段计算出跳转地址后，清空访存和译码阶段，最后再跳转到新计算出的 PC 值。<br>5. 复杂冒险。<br>当加载-使用冒险和分支跳转指令连续出现时，我们优先处理加载-使用冒险。如，jalr 指令使用的寄存器的值需要通过 lw 指令得到，我们先暂停取值译码阶段，访存获得数据后，配合转发机制，提前得到需要使用的寄存器的值， |

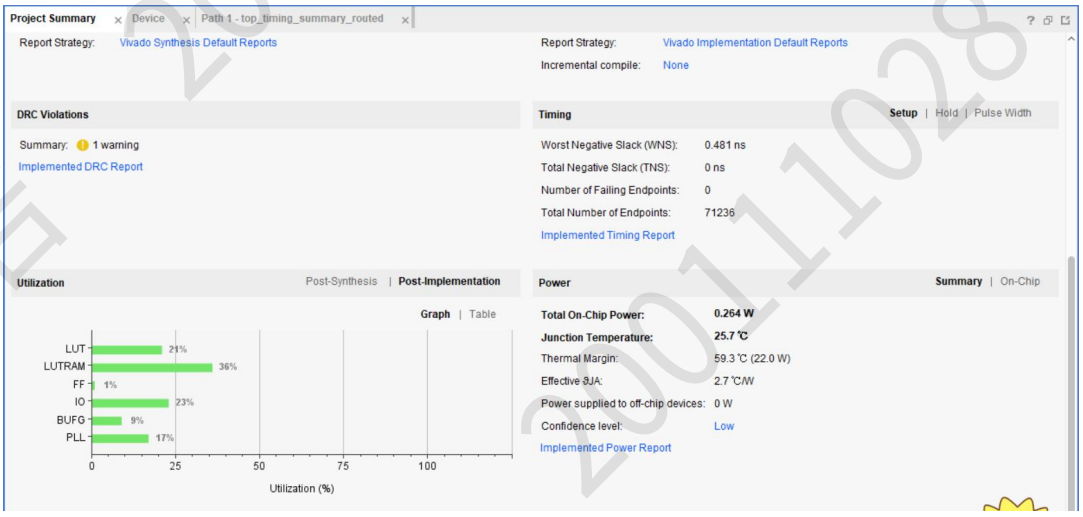
然后在执行阶段计算出跳转地址后，通过清除取值和译码阶段寄存器，重写 PC 完成跳转控制。

资源使用、功耗数据截图（Post Implementation；含单周期、流水线 2 个截图）

单周期 CPU 资源使用、功耗数据截图:




流水线 CPU 资源使用、功耗数据截图：

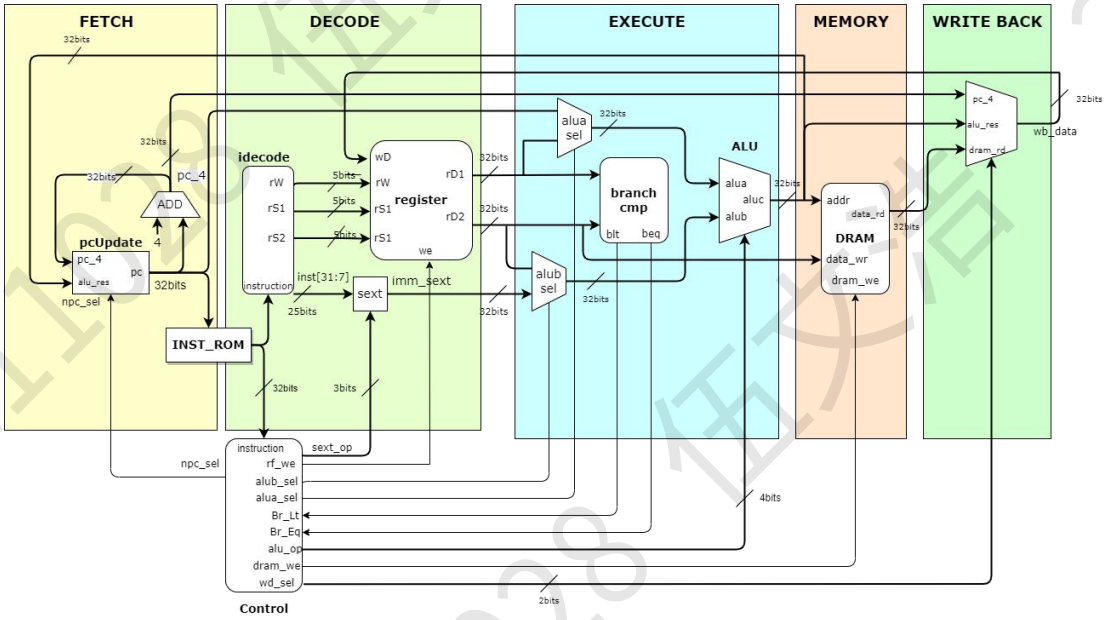


# 1 单周期 CPU 设计与实现

## 1.1 单周期 CPU 整体框图

要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、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。

1. 单周期 CPU 整体框图




2. 概述

此次实验的单周期 CPU 整体框图如上所示。从左到右按照取值–译码–执行–访存–写回的顺序排列。在 CPU 中我们一共实例化了七个模块（访问数据内存在 IO 总线中实现，而 IO 总线虽然也实例化成了模块，但其中主要实现线路的连接，逻辑性不强，这里就不赘述了）。各模块功能概述如下

**pcUpdata:** 更新 PC。

**idecode:** 译码。

**sext:** 立即数拓展。

**register:** 寄存器文件。

**branchcmp:** 分支比较。

**ALU:** ALU 执行单元。

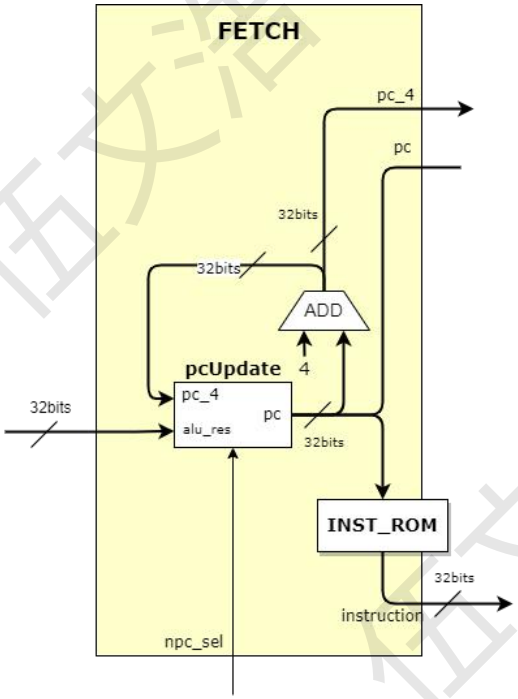
**control:** 控制控制信号，控制其他模块的行为。

## 1.2 单周期 CPU 模块详细设计

要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明。

1. 取值模块

这一部分主要完成：PC、PC+4 的计算、指令获取和更新 PC 的工作。



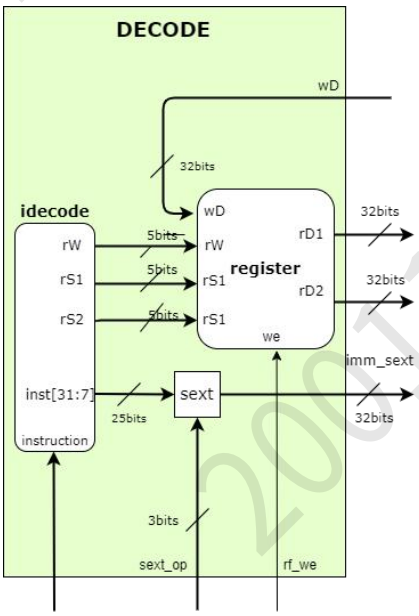

(1) 模块 pcUpdate 接收来自控制模块的 npc_sel 信号，该信号为 0 时，下一条指令地址为 PC+4；该信号为 1 时，下一条指令地址为 ALU 的计算结果。

(2) 使用加法器计算出 PC+4 的值，和 PC 一起供后续模块使用。

(3) 将 PC 传递给指令寄存器 INST_ROM，从中读取出 PC 对应的指令。

2. 译码模块

这一部分主要完成：指令分解、立即数扩展和寄存器文件的读、



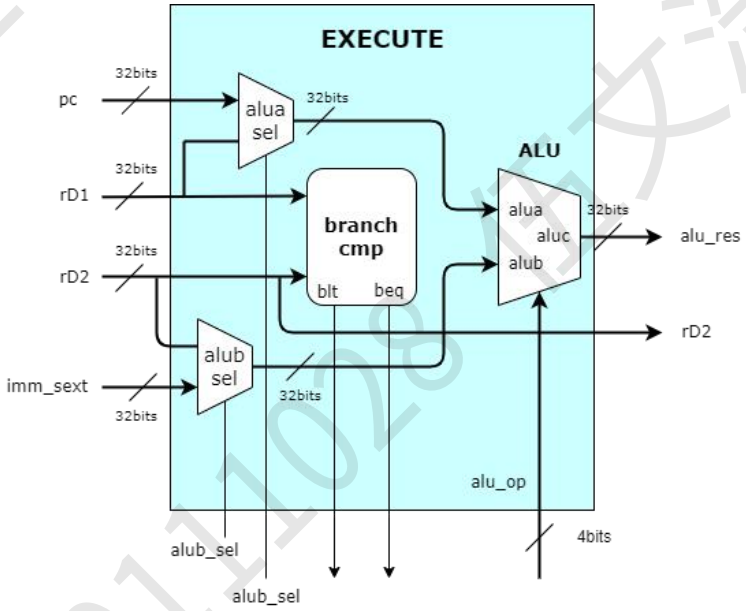
(1) 解析指令。IDECODE 模块接收来自 INST_ROM 的指令信号，从指令中解析出源寄存器一、源寄存器二、目的寄存器和立即数符号扩展需要使用的所有位。

(2) 立即数扩展。sext 模块完成立即数扩展。它接收来自 control 模块的控制信号 sext_op，根据 sext_op 对立即数进行扩展。复活拓展使用 verilog 的拼接指令完成。

(3) 寄存器访问。在我的设计中，读寄存器的操作是使用组合逻辑实现的。传入源寄存器编号后，直接读出相应的数据。

3. 执行模块

此阶段主要完成：分支比较操作、ALU 源操作数的选择和进行 ALU 运算。




(1) 分支比较。branchcmp 模块根据输入寄存器数的大小关系输出相应的指令信号。如果 rD1 < rD2，blt 为高电平；如果 rD1 == rD2，beq 为高电平。

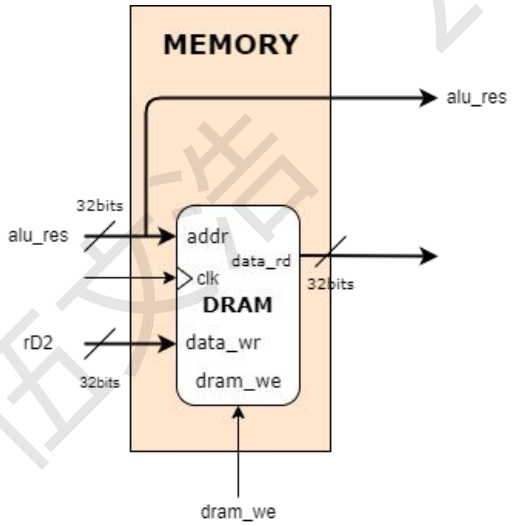
(2) 运算操作数选择。这里使用了两个三态门，分别从 rD1 和 pc 中选出 alua 和从 rD2 和 imm_sext 中选出 alub。选择信号由控制模块 control 提供。

(3) 运算。ALU 运算模块接收来自 control 模块的信号 alu_op，在内部完成相应的逻辑运算，结果从 alu_res 输出。

(4) 将 rD2 向后传递给访存模块使用。

4. 访存模块

这一阶段我们主要完成对数据内存的读写操作。

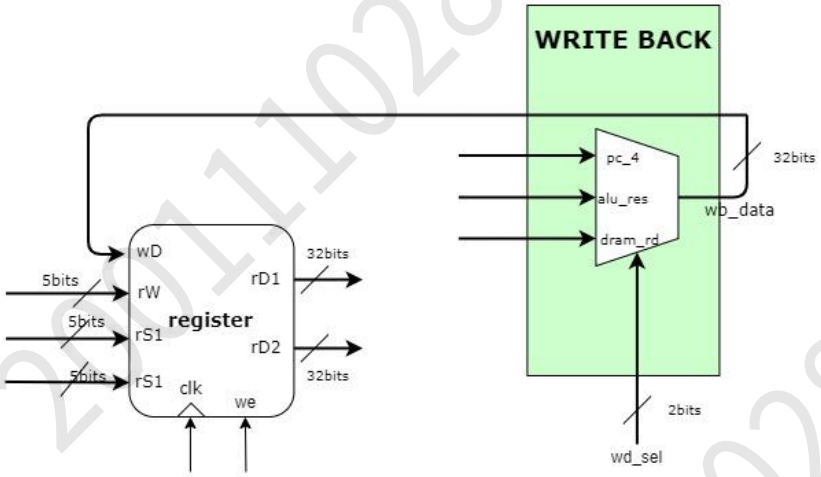



(1) 数据存储器 DRAM 使用 IP 核实现，读操作为组合逻辑，写操作为时序逻辑。访存地址由 alu_res 提供，写回数据由 rD2 提供，写使能信号来自 control 单元，数据从 data_rd 读出。

(2) 执行模块的运算结果 alu_res 被发送到下一阶段。

5. 写回模块

在此阶段主要完成写回寄存器操作数的选择。




我们使用了一个多路选择器，使用来自 control 的控制信号 wb_sel 选择写回数据。接收三个数据：PC+4（32bits）、alu_res（32bits）、data_rd（32bits）。

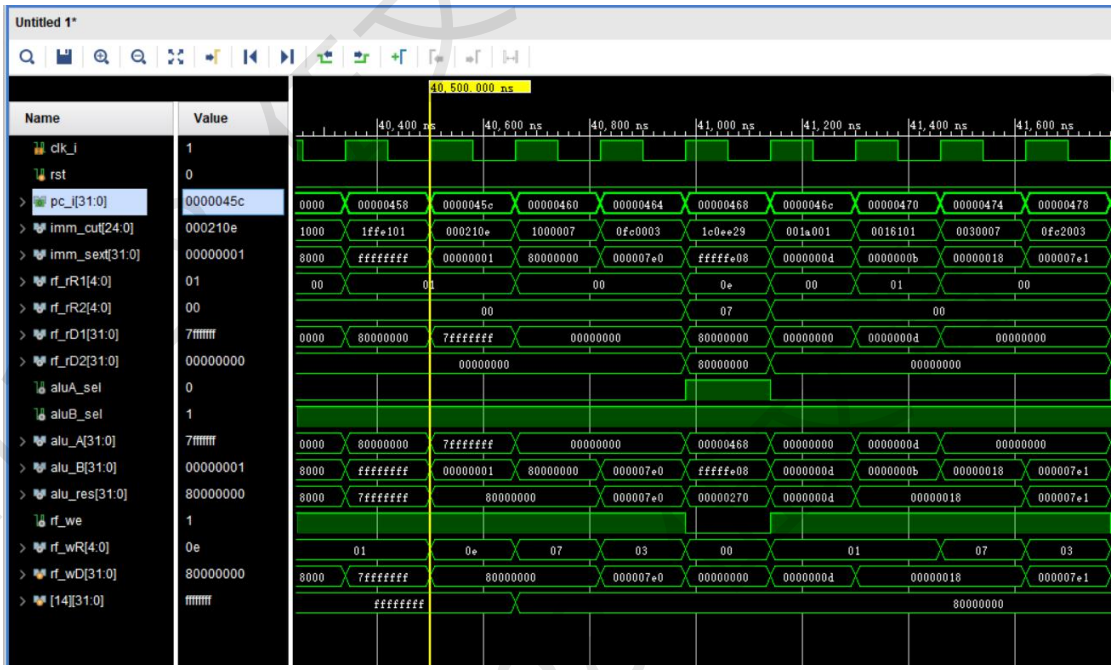
写回到寄存器文件采用时序逻辑。写回之前，我们会先判断写入的寄存器编号是否为 0，如果编号为 0，则跳过这次写入操作。

## 1.3 单周期 CPU 仿真及结果分析

要求：包含逻辑运算、访存、分支跳转三类指令的仿真截图以及波形分析；每类指令的截图和分析中，至少包含 1 条具体指令；截图需包含信号名和关键信号。

1. 逻辑运算指令

以 addi 为例，波形如下所示：




(1) 黄线标出时刻，时钟上升沿到来，pc 更新为 0x0000045c，指令更新为 0x00108713。这条机器码对应的汇编程序为 addi　x14, x1, 1，汇编上下文如下：

```
454: 800000b7          lui x1,0x80000
458: fff08093          addi  x1,x1,-1 # 7fffffff <_end+0x
45c: 00108713          addi  x14,x1,1
```

(2) 译码阶段。经过前两条指令，寄存器 x1 的值被设置为 0x7fffffff。观察波形可以看到，源寄存器一编号 rf_rS1 的值为 01，说明是从寄存器 x1 取数，取出的数据为 0x7fffffff，这与前两步汇编的运行结果一致。再看立即数。imm_cut 是未经重组的立即数，而 imm_sext 是经过扩展重组后的立即数，其值为 0x00000001，与指令表述一致。

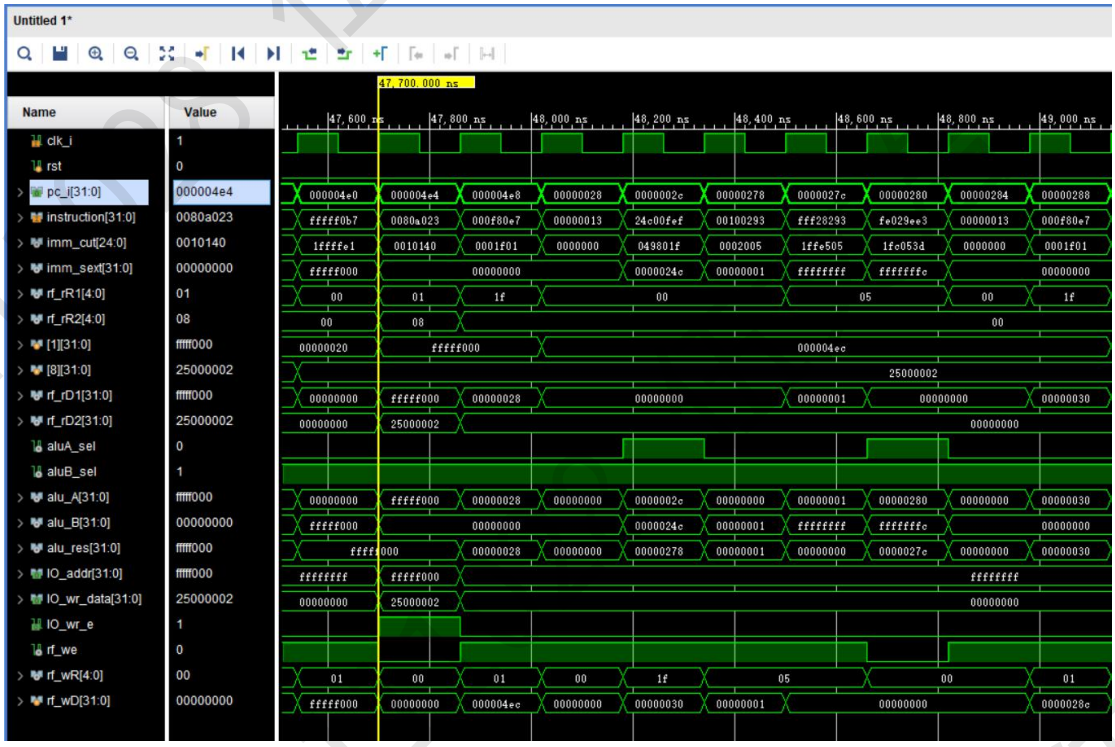
(3) 进入执行模块。alua_sel 为 0，表示选择 rf_rD1 作为第一个操作数；alub_sel 为 1，表示选择立即数作为第二个操作数；可以看到，经过选择后，操作数一 alu_A 的值为 0x7fffffff；操作数二 alu_B 的值为 0x00000001，符合预期。alu_op 为 0，表示进行加法运算。运算结果 alu_res 为 0x80000000。

(4) 该指令没有访存操作。

(5) 在写回阶段，注意到，写回使能信号 rf_we 为高电平 1，说明要进行一次写操作，写寄存器编号 rf_wR 为 0e 表示结果将写入寄存器 x14，而需要写入的数据 rf_wD 为 0x8000000，与计算结果一致。因为写寄存器使用时序电路实现，所以在下一个时钟上升沿到来后完成对寄存器文件的写入。至此指令执行完毕。

2. 访存指令

以 sw 指令为例，仿真波形如下：




考察黄线指示的时钟上升沿。我们将要执行一条 sw 指令，该指令的汇编上下文如下图所示：

```
4dc: 00140413          addi  x8,x8,1
4e0: fffff0b7          lui x1,0xfffff
4e4: 0080a023          sw  x8,0(x1) # fffff000 <_end+0xffffaf90>
4e8: 000f80e7          jalr  x1,0(x31)
```

(1) 取值阶段，pc 更新为 0x000004e4，从 INST_ROM 中取出指令 0080a023，这是一条 sw 指令，其含义是将寄存器 x8 的值写入到内存地址为 R[x1]+0x0（实际上是外设数码管，但是对 CPU 而言，外设和内存由总线桥统一为巨大的内存）的位置。

(2) 译码阶段，指令中的立即数被解析为 imm_sext=0x00000000，这条指令访问的源寄存器一为 rf_rR1=01，源寄存器二为 rf_rR2=08。我们查看寄存器文件可以得知这两个寄存器内储存的值，R[x1]=0xfffff000，R[x8]=0x25000002。读出值 rf_rD1 和 rf_rD2 与寄存器内的储存的数据

是一致的。

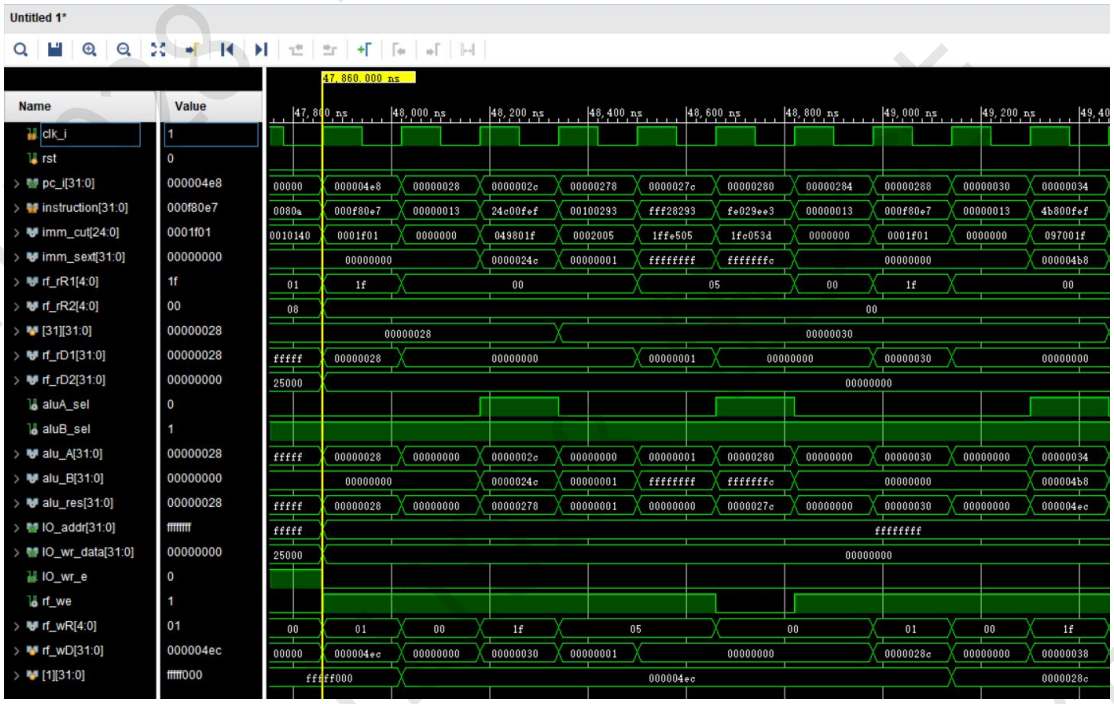
(3) 执行阶段，aluA_sel=0，表示选择 rf_rD1 作为操作数一，aluB_sel=1，表示选择立即数，作为操作数二。计算出结果 alu_res=0xfffff000。

(4) 访存阶段，我们需要写外设。地址 IO_addr 来自 alu_res=0xfffff000，写使能信号 IO_wr_e=1，写数据 IO_wr_data 来自 rf_rD2=0x25000002。

(5) 写回阶段无操作，寄存器写使能 rf_we=0。

3. 分支跳转指令

这里以 jalr 指令为例，仿真波形如下图所示：




考察黄线指示的时钟上升沿到来时，执行一条新的指令该指令上下文为：

```
4dc: 00140413          addi  x8,x8,1
4e0: fffff0b7          lui x1,0xfffff
4e4: 0080a023          sw  x8,0(x1) # fffff000 <_end+0xffffaf90>
4e8: 000f80e7          jalr  x1,0(x31)
```

(1) 取值阶段，pc 更新为 0x000004e8，从指令寄存器 INST_ROM 取出指令 instruction=0x000f80e7，这是一条 jalr 指令，指令含义为：跳转到 R[x31]+0 地址处，并且将 pc+4 的值储存到寄存器 x1 内。

(2) 译码阶段，解析出立即数的值为 imm_sext=0x00000000。源寄存器一 rf_rR1=1f，源寄存器二不使用。从寄存器一中读出的数据 rf_rD1=0x00000028，这与寄存器文件中显示的数值一致。

(3) 执行阶段。选择 rf_rD1 和立即数进行加法运算，运算结果为 alu_res=0x00000028。这个运算值实际上也是下一条指令的地址，也

就是 pc 的下一步值。查看下一时钟上升沿到来时，发现 pc 被更新为 0x00000028。

(4) 本指令不涉及访问内存。

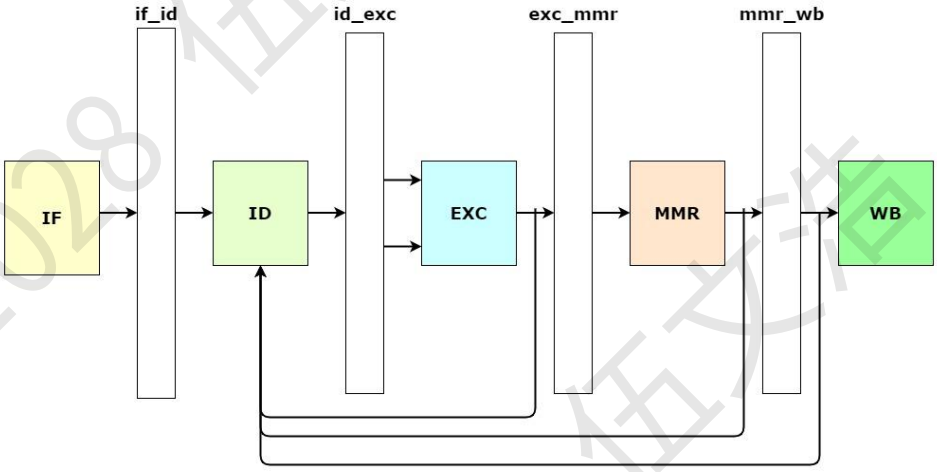
(5) 写回阶段，要将 pc+4=0x000004ec 的值写入寄存器 x1。寄存器写使能 rf_we=1，rf_wR=01，rf_rD=0x000004ec。可以看到下一时钟上升沿到来后，寄存器 x1 的值被更新为 0x00004ec。至此，指令执行完成。

# 2 流水线 CPU 设计与实现

## 2.1 流水线的划分

要求：画出流水线如何划分，说明每个流水级具备什么功能、需要完成哪些操作。

1、流水线划分




如图所示，采用常用的五阶段流水线，中间使用四个寄存器分隔，加上用于数据前递的线路。

2、流水线各阶段功能：

（1） IF：CPU 取值阶段。在这一阶段，保存 PC 的值，并且在每个时钟上升沿到来时更新 PC。通过 OUTPUT 线路将 PC 传递给 IROM，获得指令，并分析指令类型，将指令划分成各个部分传递给译码阶段。

（2） ID：CPU 译码阶段。在这一阶段，我们一方面通过分析指令重写访问和写回寄存器的编号、产生各种控制（使能）信号，另一方面读取寄存器的值。在这一阶段，我们还能检测数据冒险，并且通过数据转发的方式处理冒险。

（3） EXC:CPU 执行阶段。这一阶段主要有两个模块：分支选择模块和 ALU 运算模块。分支选择模块比较从寄存器中读出的数据（经过转发修正），根据指令类型计算出分支选择信号。而 ALU 运算模块则根据 alu_op 对输入的操作数进行相应的运算，并得出运算结果。当然，传入 ALU 运算单元的操作数时经过多路选择器选择后的操作数。

（4） MMR:CPU 访存阶段。这一阶段是我们 CPU 访问数据内存和外设的交流平台。访存阶段的功能并没有模块化实现，其主要逻辑在于将 CPU 内部的线路和外部线路连接起来。

（5） WB:CPU 写回阶段。顾名思义，我们需要将执行阶段或者访存阶段获得的值写回寄存器文件。此时，我们可以保证已经获得了所有需要的数据，我们通过 wd_sel 选择写回寄存器的数据。

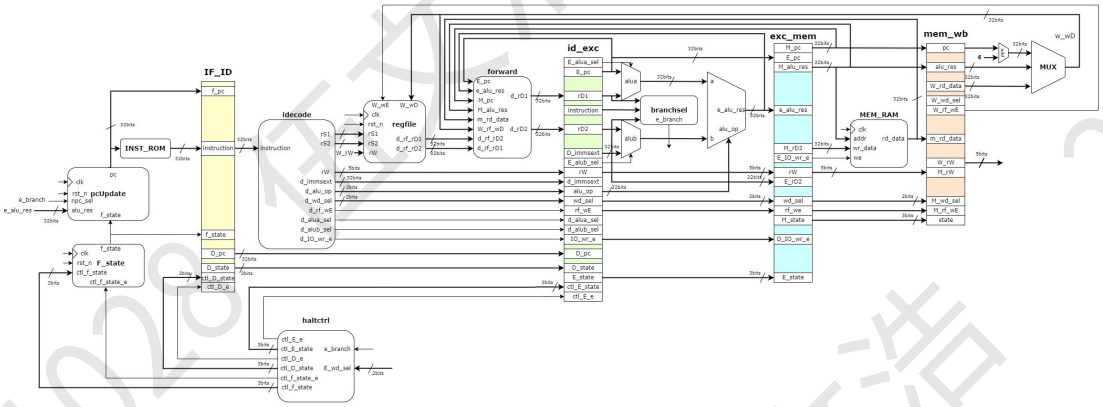
此次实验中，我们设计了标准五级流水线 CPU，它们通过四个寄存器分隔开。这四个寄存器在时钟上升沿到来时一齐更新内容，从而实现各个阶段的数据能够井然有序地流动。

通过在流水线中增加前递线路和控制模块，我们的 CPU 可以高效而正确地处理数据冒险和控制冒险。

## 2.2 流水线 CPU 整体框图

要求：无需画出模块内的具体逻辑，但要标出模块的接口信号名、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。

1. 流水线整体框图




如上图所示，我的流水线 CPU 被四个寄存器划分成五个部分，分别代表指令执行的取值、译码、执行、访存和写回阶段。位于译码阶段的 forward 模块负责完成数据前递，总览全局的 haltctl 模块负责控制各个寄存器停顿或者清除寄存器内容。

2. 各模块功能简述

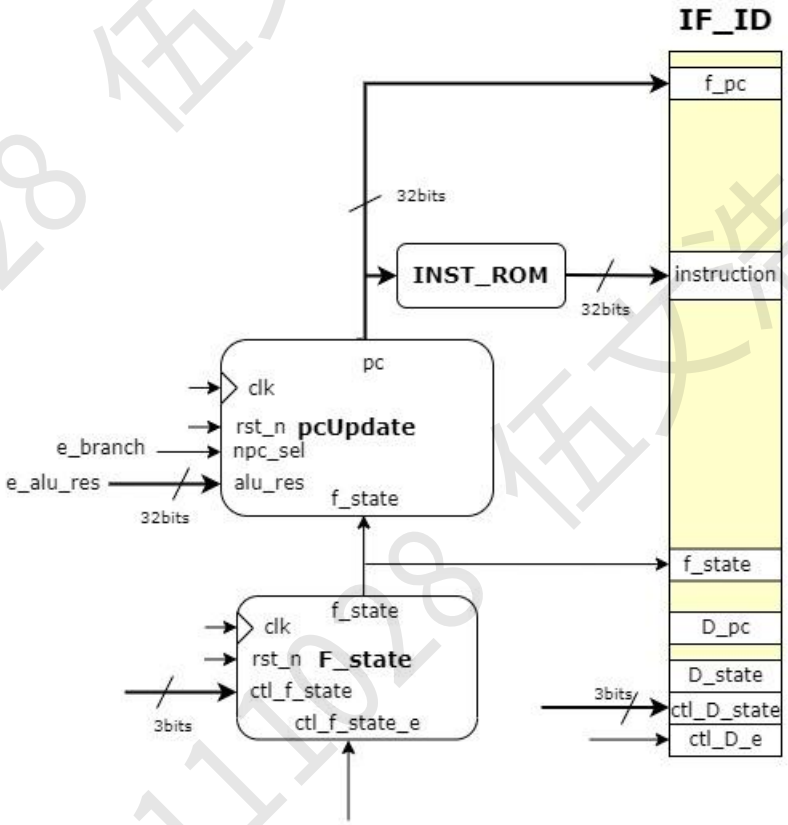
(1) pcUpdate：取值阶段模块。负责保存取值阶段 pc 值，以及更新 pc 操作。
(2) F_state：　取值阶段模块。负责计算该阶段运行状态。
(3) idecode：　译码阶段模块。“翻译”指令，产生各种控制信号。
(4) regfile：　译码阶段模块。寄存器文件。
(5) forward:　在译码阶段实现。负责处理数据前递（该模块线路较多，部分为在图中体现）。
(6) branchsel: 比较操作数大小关系，产生分支跳转信号。
(7) alu:　　进行算数逻辑运算。
(8) INST_ROM：指令内存，由 IP 核实现
(9) MEM_RAM：数据内存，由 IP 核实现。

## 2.3 流水线 CPU 模块详细设计

要求：画出各个模块的详细设计图，包含内部的子模块，以及关键性逻辑；标出子模块接口信号名、各信号线的信号名和位宽，并有详细的解释说明；此外，必须结合模块图，详细说明数据冒险、控制冒险的解决方法。

### 1. 取值阶段

(1) 详细设计图




(2) 实现功能实现解析

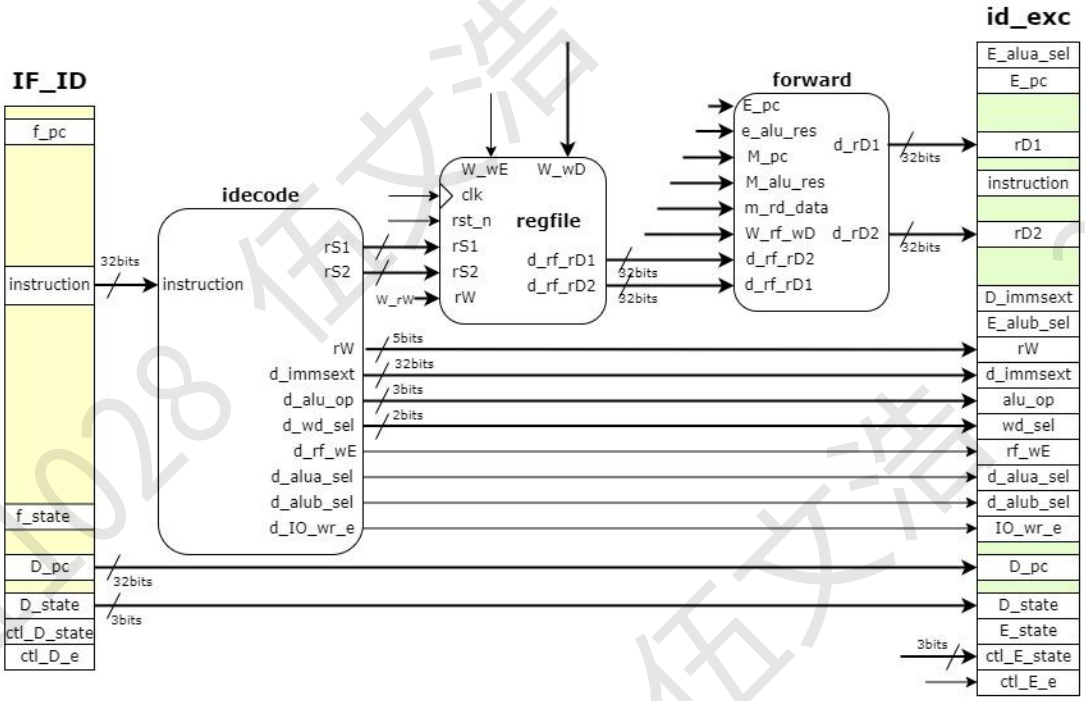
这一阶段主要完成更新 PC，取值的任务。真正与这两个任务有关的模块只有 pcUpdate 模块和指令 IP 核 INST_ROM。F_state 与流水线控制有关，我将在控制冒险部分讲述其功能。

pcUpdate 模块接收外部时钟信号 clk。每当时钟上升沿到来时，检查 rst_n 复位信号和 f_state 状态信号，当且仅当 rst_n=1 且 f_state= STATE_WORK 时更新 pc。

下一步 pc 值由分支选择信号 e_branch 决定。e_branch 为 0 时选择 pc+4，e_branch 为 1 时选择 alu_res。

## 2. 译码阶段

(1) 详细设计图




(2) 实现功能实现解析

这一阶段主要完成指令的解析和寄存器文件的读取。

模块 idecode 从中继寄存器中获取指令信号，并且将其解析为其他信号，包括 r1，r2，rw，扩展后的立即数和其他控制信号等。有些信号会在本阶段被使用，如寄存器序号；有些信号会向后传递，直到其需要使用的阶段。

模块 regfile 是寄存器文件，它接收源寄存器编号，从中读出相应的数据。另外，每当时钟上升沿到来时，如果写使能为 1，并且写寄存器不为 0 时，regfile 会使用 W_wD 更新相应寄存器。（注，W_wE 和 W_wD 都是来自写回阶段的信号）

模块 forward 是处理数据前递的模块。该模块输入较多，图中展示了所有参与前递结果选择的数据线路。模块将写使能为 1 的阶段的写寄存器与当前译码的寄存器进行比较，以判断是否存在数据前递的需要。发现数据冒险时，模块内按照“执行”>“访存”>“写回”的优先级次序处理转发数据。下面以处理读寄存器一为例，展示实现代码。

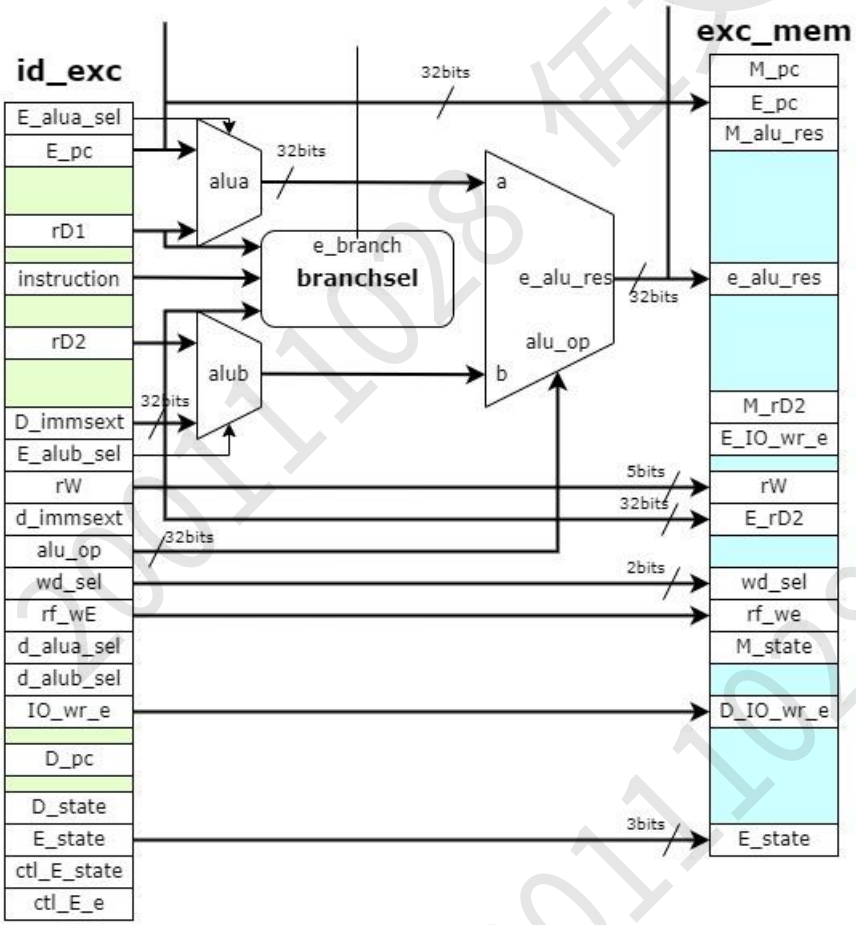
```
// 考察各个阶段是否存在对源寄存器一的转发
wire fw_E_1 = (E_wd_sel == `WB_PC4 || E_wd_sel == `WB_ALU) && E_rW != 5'b0 && E_rW == d_rS1;
wire fw_M_1 = M_wd_sel != `WB_NONE && M_rW != 5'b0 && M_rW == d_rS1;
wire fw_W_1 = W_wd_sel != `WB_NONE && W_rW != 5'b0 && W_rW == d_rS1;

// 实现对寄存器一的转发
always @(*) begin
    if (fw_E_1)         d_rD1   = E_wd_sel == `WB_PC4 ? E_pc + 3'b100 : e_alu_res;
    else if (fw_M_1)    d_rD1   = M_wd_sel == `WB_ALU ? M_alu_res
                                : M_wd_sel == `WB_PC4 ? M_pc + 3'b100
                                : m_rd_data;
    else if (fw_W_1)    d_rD1   = w_rf_wD;
    else                d_rD1   = d_rf_rD1;
end
```

信号 fw_E，fw_M，fw_W 分别表示执行、访存和写回阶段是否存在数据前递需要。检测到前递需要后，按照前面提到的优先级次序处理数据前递。如果不需要前递，则使用从寄存器文件中读取出的数据作为最终使用的数据。

### 3. 执行阶段

(1) 详细设计图




(2) 实现功能实现解析

这一阶段主要完成分支选择的判断和 ALU 逻辑运算。

branchsel 模块检测到指令为 jal，jalr 跳转指令或者 B 型指令并且 rD1 和 rD2 的大小关系与跳转要求的一致时，发出跳转信号。位于取值阶段的 pcUpdate 模

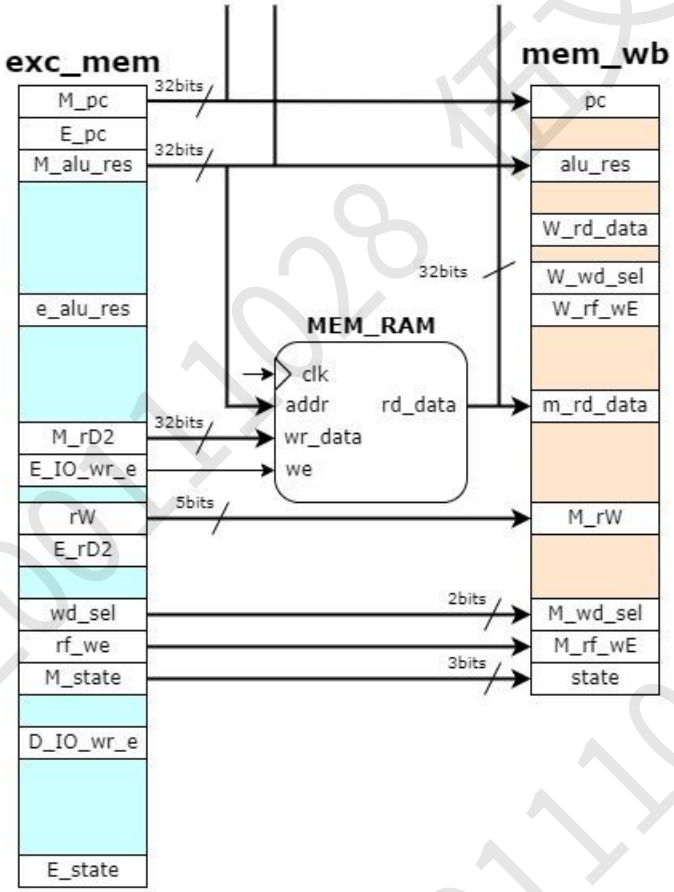
块接受到分支跳转信号后，将选择 alu 运算结果作为下一周期的 pc。

在进行 ALU 运算之前，先使用三态门器件选择参与运算的操作数。当 alua_sel 为 0 时，选择 rD1 为第一操作数，否则选择 pc 为第一操作数；当 alub_sel 为 0 时，选择 rD2 为第二操作数，否则选择 immsext 为第二操作数。在 ALU 运算单元中，我们可以执行九种运算操作——add，sub，and，or，xor，sll，slr，sar，直接使用 alub。实现时，同步执行这些运算，然后使用多路选择器，以 alu_op 为选择信号，选择出正确结果。

另外，这里将 pc 值和 alu 运算结果前递给了译码阶段的 forward 模块。因为，sw 指令在访存阶段需要使用 rD2 的值，所以 rD2 被传递给了下一阶段，但是 rD1 没有。

**4. 访存阶段**

(1) 详细设计图




(2) 实现功能实现解析

此阶段主要实现读写数据内存或者与外设进行数据交换。实际实现时，访问外设和访问内存都通过总线桥实现。这里图中仅展示了读写内存的情况。

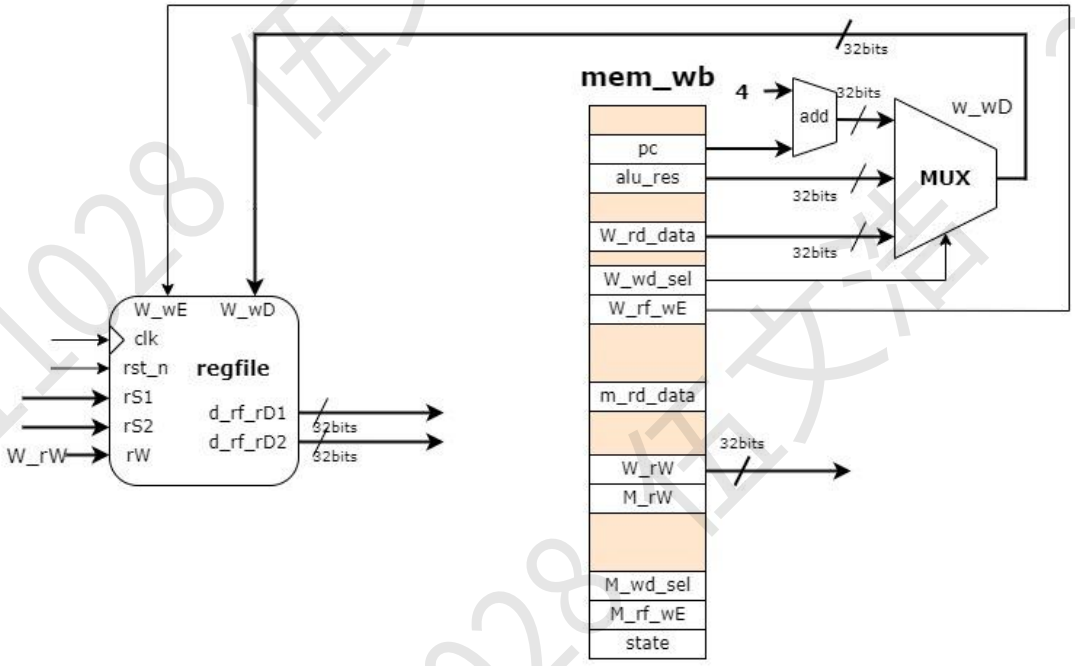
读写内存时，地址由 alu_res 提供。当时钟上升沿到来时，如果写信号 we 为高电平时，将 rD2 的内容写入相应内存地址（或外设中）；而 rd_data 采用组

合逻辑直接从相应内存地址（或外设）读出。

为了处理这一阶段的数据冒险，我们将 alu_res，pc 和从内存（外设）读取的数据向前传送给 forward 模块。

## 5. 写回阶段

(1) 详细设计图



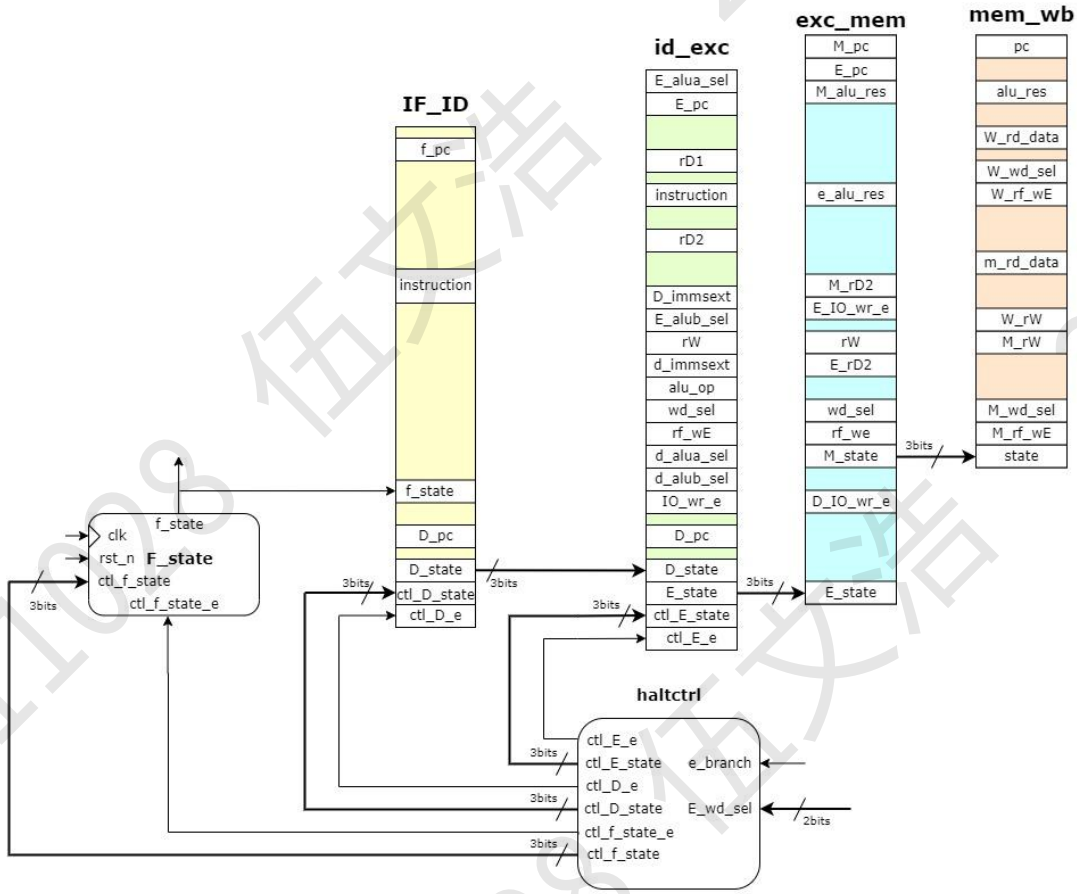

(2) 实现功能实现解析

写回阶段逻辑相对简单。只需要使用一个多路选择器就可以从 pc+4，alu_res 和访存读取结果中选择合适的结果写回到寄存器 W_rW 中。唯一要注意的一点是，当写回的是零号寄存器时，需要保持寄存器 x0 的值不变。

## 6. 处理控制冒险

(1)模块图

如图所示，在我设计的流水线 CPU 中，主要依赖一个暂停控制模块和各级中继寄存器处理解决控制冒险。控制冒险又可以具体细分为两类——加载-使用冒险和由分支跳转指令产生的冒险。




(2) 解决控制冒险的思路

控制信号 state 一共有三种：work、pause 和 idle。当时钟上升沿到来时，如果某一中继寄存器接受到了 work 信号，它将使用前一阶段的值更新寄存器；如果其接收的是 pause 信号，将保持值不变；如果它接受到的是 idle 信号，寄存器将复位为初始状态。

解决加载-使用冒险：当执行阶段的写回选择信号 wd_sel 为 WB_IO（写回 IO 读取的结果），并且写回寄存器非零，而且将在下一指令的译码阶段被使用时，haltctl 向 F_state 和寄存器 IF_ID 发送 pause 暂停信号；向寄存器 id_exc 发送清除 idle 信号，在下一周期，“加载”指令可以从访存阶段读取到数据，配合数据前递机制，给译码阶段使用。

解决分支跳转指令冒险：当检测到执行阶段 e_branch 信号为 1 时，说明该指令应当选择跳转到分支地址执行。因为我们的 pcUpdate 模块会根据 e_branch 信号选择下一阶段的 pc，所以我们无需对该阶段进行操作。我们只需在下一时钟到来时清除寄存器 IF_ID 和寄存器 id_exc 的内容就可以了（也就是，清除了两个阶段的执行结果）。

## 2.4 流水线 CPU 仿真及结果分析

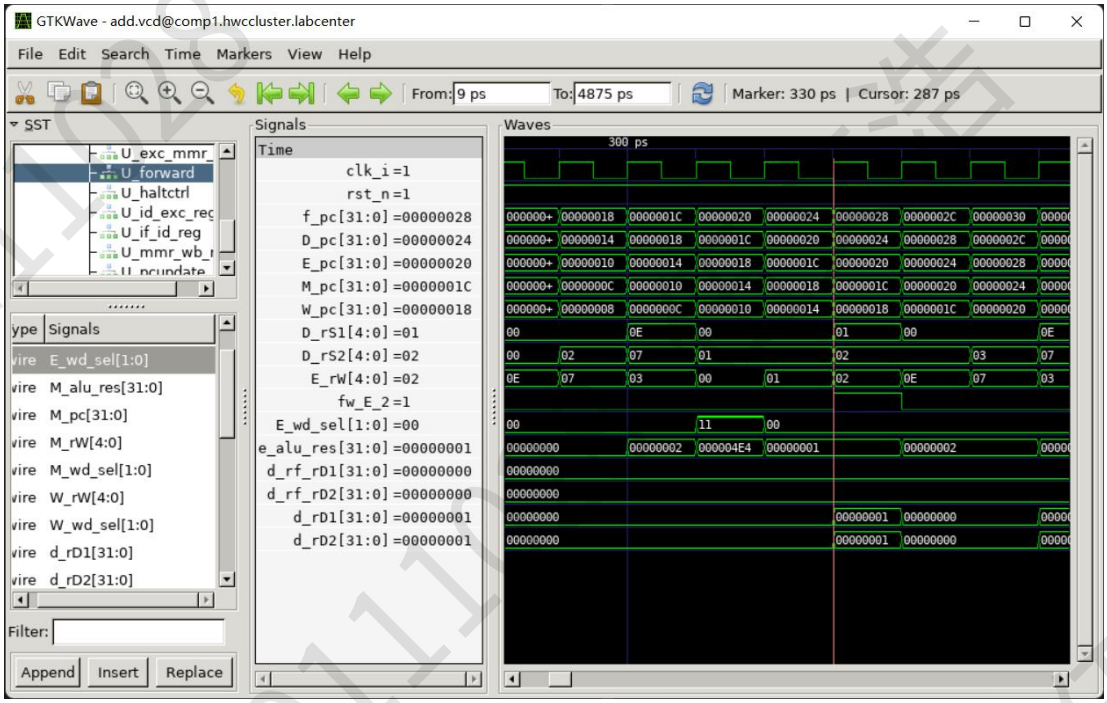
要求：包含控制冒险和数据冒险三种情形的仿真截图，以及波形分析。

1. 流水线数据冒险

(1) 相邻指令使用同一寄存器

如图所示，地址位于 0x24 的指令使用到了前一条指令在执行阶段计算出的寄存器 x2 的值。我将结合波形分析一下我设计的流水线 CPU 是如何处理这种数据冒险的。

```
20: 00100113          addi  x2,x0,1
24: 00208733          add x14,x1,x2
```




如上图所示，在红线标出的时钟上升沿到来后，位于地址 0x24 的指令执行到译码阶段；而位于地址 0x20 的指令执行到执行阶段。我们可以看到，执行阶段的写回寄存器 E_rW 与译码阶段使用的寄存器二 D_rS2 一致，打开处理信号前递的模块 forward 发现它检测到了这一现象，所以前递信号 fw_E_2 产生高电平。

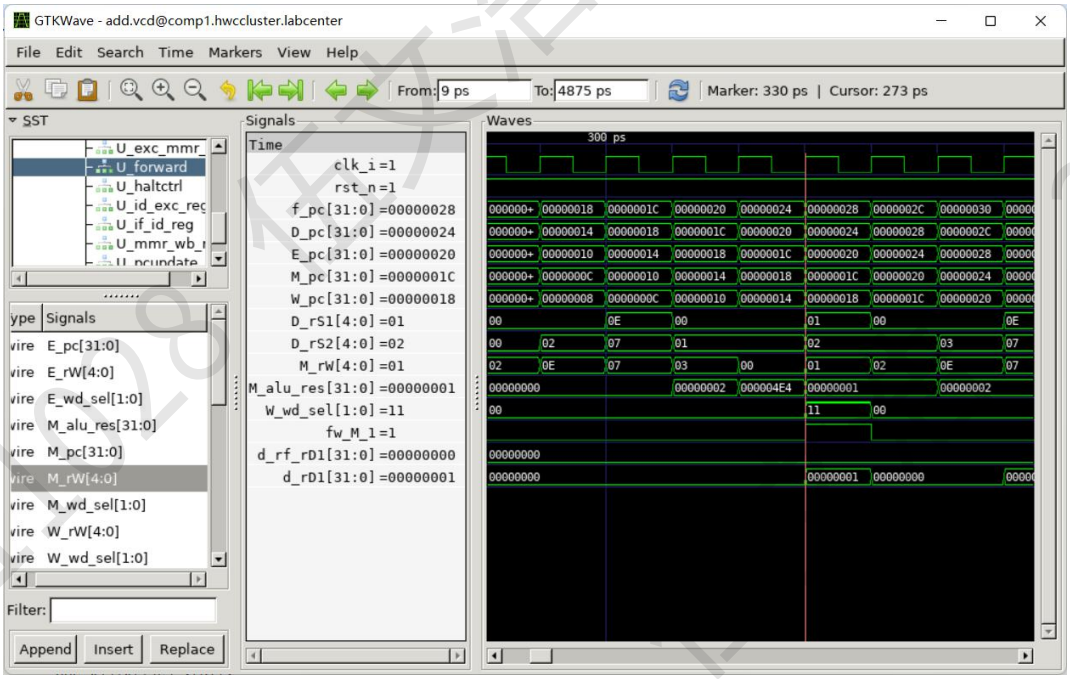
这时候，直接从寄存器内读出的数据 d_rf_rD2 是不准确的，需要使用在执行阶段新计算出的值 e_alu_res 替换，可以看到替换后 d_rD2=e_alu_res=0x01。之后，使用 d_rD2 的值进行计算即可得到正确的结果。

(2) 间隔为一的指令使用同一寄存器

如图所示，地址位于 0x24 的指令在译码时需要使用寄存器 x1 的值，而寄存器 x1 的被位于 0x1c 的指令修改过，此时，修改后的值仍在访存阶段还未写回寄存器，直接使用从寄存器内读出的 x1 的值将产生错误计算结果。我将结合

波形分析一下我设计的流水线 CPU 是如何处理这种数据冒险的。

```
1c: 00100093          addi  x1,x0,1
20: 00100113          addi  x2,x0,1
24: 00208733          add x14,x1,x2
```



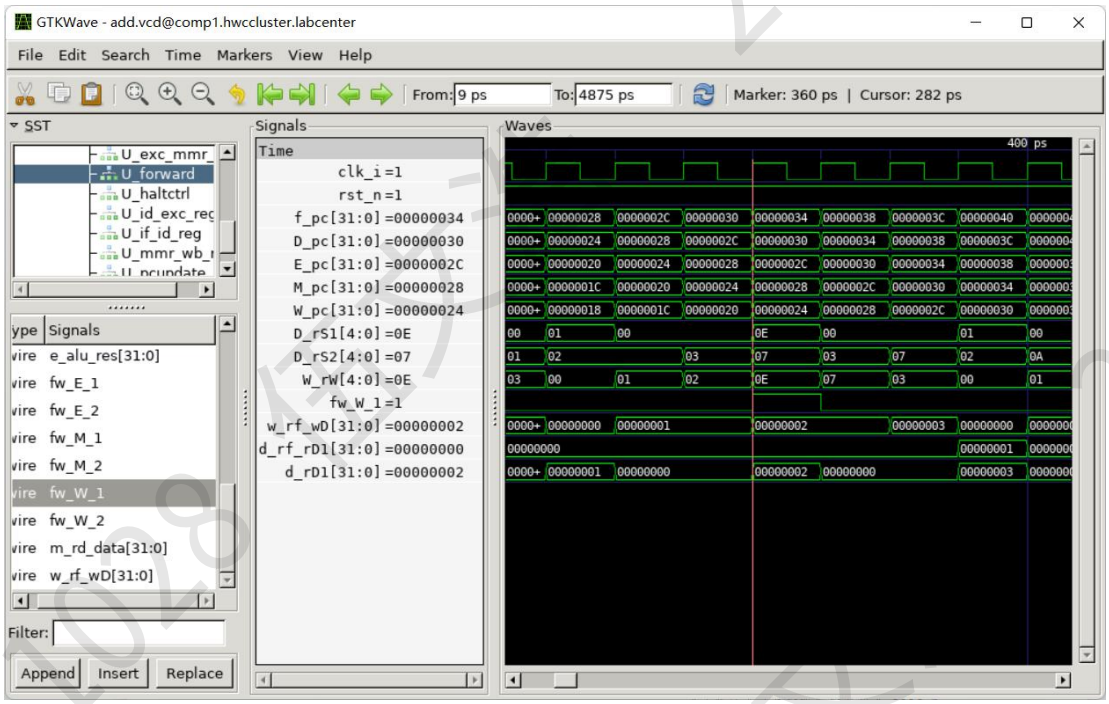

如上图所示，在红线标志的时钟上升沿啊到来时，位于译码阶段的指令是地址 0x24 处的指令 add x14, x1, x2。其译码需要使用寄存器 D_rS1=01。但是注意到，位于地址 0x1c 的指令对寄存器 x1 的写还未完成。

打开处理数据冒险的模块 forward，因为它检测到了这一冒险，所以信号 fw_M_1 为高电平，又因为 M_wd_sel=00，这指示我们使用 ALU 运算结果作为前递值。前递完成后，d_rD1=M_alu_res=0x01。前递完成。

(3) 间隔为二的指令使用同一寄存器

如图所示，地址位于 0x30 的指令在译码时需要使用寄存器 x14 的值，而寄存器 x1 的被位于 0x24 的指令修改过。在指令 0x30 的译码阶段，修改后的值虽然已经运行到了写回阶段，但是写寄存器要到下一个时钟周期到来时才会进行。这是如果直接使用从寄存器内 x14 的值将产生错误计算结果。我将结合波形分析一下我设计的流水线 CPU 是如何处理这种数据冒险的。

```
24: 00208733          add x14,x1,x2
28: 00200393          addi  x7,x0,2
2c: 00300193          addi  x3,x0,3
30: 4a771a63          bne x14,x7,4e4 <fail>
```




如上图所示，当红线指示的时钟上升沿到来时，位于地址 0x30 的指令运行到译码阶段，它需要读取 D_rS1=x14 的值。

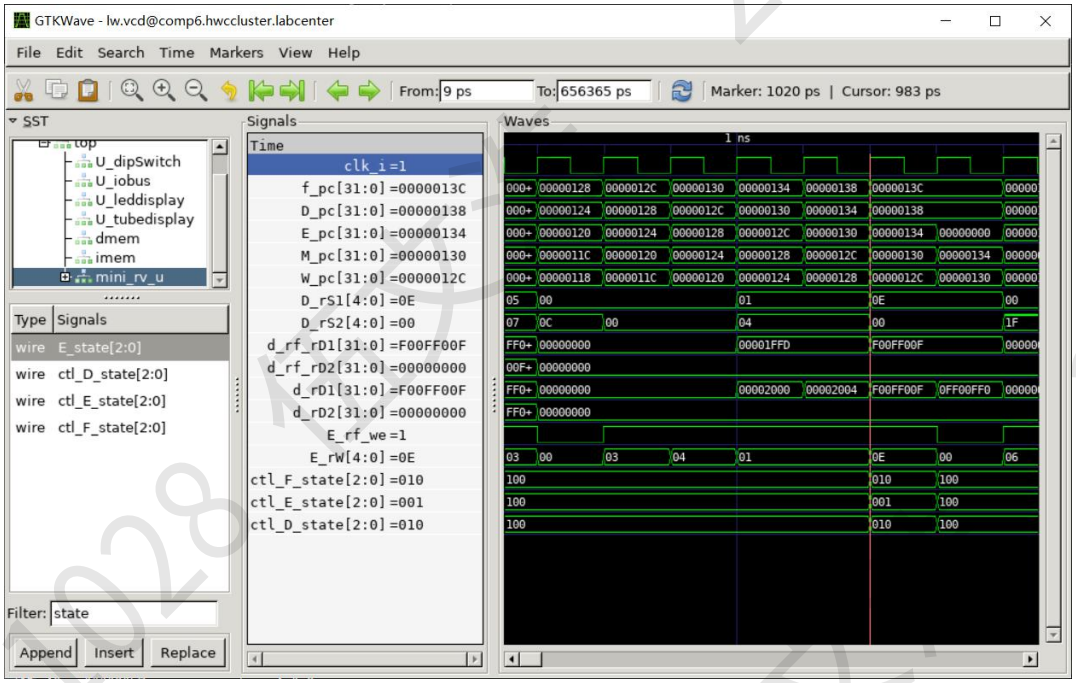
在 forward 模块中检测到写回阶段的写寄存器 W_rW=x14 和译码阶段需要使用的寄存器时一致的，于是产生前递信号 fw_W_1，表明对寄存器一有写回阶段的数据前递。由于在写回阶段已经产生了写回的数据 w_rf_wD，所以我们直接将 w_rf_wD 的值赋值给 d_rD1 即可。

2. 控制冒险

(1) 加载-使用冒险

如下图所示，lw 指令从内存加载数据存入寄存器 x14，而下一条指令 addi 使用寄存器 x14 的值作为源操作数。当指令 addi 运行到译码阶段时，指令 lw 才运行到执行阶段，也未获得寄存器 x14 的值，所以，我们单纯地无法使用转发机制解决这种冒险。这时候就需要使用控制流水线运行的手段。

```
134: 0040a703          lw  x14,4(x1)
138: 00070313          addi  x6,x14,0
```




如上图所示，在红线表明的时钟上升沿到来时，addi 指令运行到译码阶段，而 lw 指令才运行到执行阶段。因为 CPU 检测到执行阶段的写寄存器编号 E_rW 与访存阶段中的一个读寄存器编号 D_rS1 一致，并且当前指令是 lw 指令，触发控制暂停机制。

可以看到，在该时钟周期，三个控制信号都发生了变化，ctl_E_state=001，表示执行阶段清空结果（即插入空泡），ctl_F_state=ctl_D_state=010 表示取值阶段和译码阶段暂停一个周期，这一变化可以从各个阶段的 pc 值体现——执行阶段清零，取值和译码阶段维持上一个周期的值。

在下一个周期，由于访存完毕后获取了相应内存储存数值，这是所有阶段恢复正常运行，状态码为 100。然后通过指令前递机制完成对 x14 的使用，前递前，d_rf_rD1=0xF00FF00F，前递后 d_rD1=0x0FF00FF0。至此，加载使用冒险处理完毕。

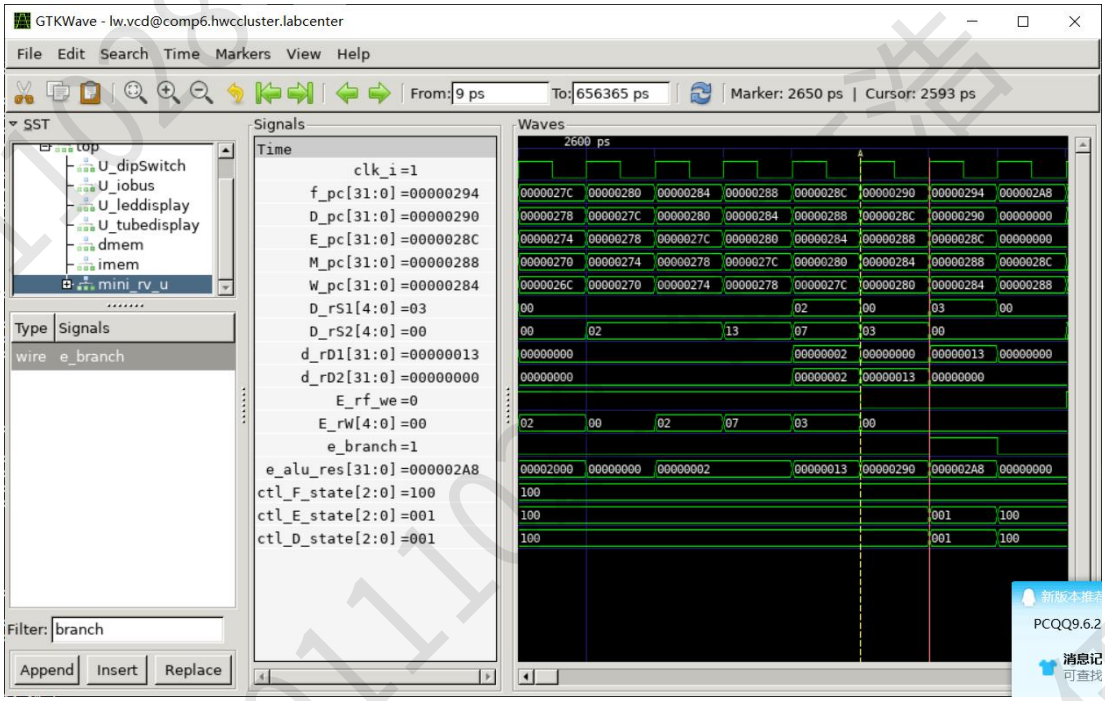
(2) 分支跳转指令

如下图所示，当执行到位于地址 288 的指令时，因为运行结果正确，所以不应当跳转到分支，而当执行到位于地址 28c 的指令时，由于运行结果正确，应当跳转到 pass 分支。我将根据波形图分析 CPU 是如何处理这两种情况的。

```
 288: 00711463          bne x2,x7,290 <fail>
 28c: 00301e63          bne x0,x3,2a8 <pass>

00000290 <fail>:
 290: 00018063          beq x3,x0,290 <fail>
 294: 00119193          slli  x3,x3,0x1
 298: 0011e193          ori x3,x3,1
 29c: 05d00893          addi  x17,x0,93
 2a0: 00018513          addi  x10,x3,0
 2a4: 00000073          ecall

000002a8 <pass>:
 2a8: 00100193          addi  x3,x0,1
```




流水线 CPU 运行波形如上图所示。在我设计的流水线 CPU 中，需要到执行阶段才能判断是否需要进行分支跳转，所以，在选择下一条指令时全部采用 pc+4。

在绿色虚线指示的时钟上升沿到来时，第一条 bne 指令运行到执行阶段，此时计算出的结果是不选择分支（e_branch=0），所以各个阶段正常运行（即维持之前 pc+4 的选择）。

在红色实线指示的时钟上升沿到来时，第二条 bne 指令执行到执行阶段，它发现，应到选择分支（e_branch=1），也就是说，之前选择 pc+4 的执行结果应当作废。所幸，作废的两条指令最多值运行到译码阶段，没有对寄存器文件和内存做任何修改，我们通过传递 ctl_D_state=ctl_E_state=001 清除信号，来消除这两条指令的影响（因为这两条指令在下一时钟周期将分别进入 译码和执行 阶段）。

再下一个周期，可以看到 f_pc 的值已经变为上一时钟周期在执行阶段计算出的下一 pc 值 e_alu_res=0x2AB。至此，分支跳转指令的控制冒险解决。简单的跳转指令 jal 和 jalr 指令同理，这里不再赘述。

# 3 设计过程中遇到的问题及解决方法

要求：包括设计过程中遇到的有价值的错误，或测试过程中遇到的有价值的问题。所谓有价值，指的是解决该错误或问题后，能够学到新的知识和技巧，或加深对已有知识的理解和运用。

在设计实现单周期和流水线 CPU 时，我遇到并解决了下列问题：

1. 外设的写实现。

在写外设时，由于 sw 的指令只存在一个时钟周期，其信号也只会存在一个时钟周期，在之前的设计中，数码管和 led 的数据都只显示一个周期，这对于人眼而言过快了，所以看上去数码款一直显示全零。

后来的解决方案是把写外设改为使用时序逻辑实现，外部使用几个寄存器储存需要显示的值，在每个时钟上升沿到来时，通过检查写使能和写地址对显示数据进行更新。**也就是，外设读取用组合逻辑实现，写外设用时序逻辑实现。**

2. 无法满足时钟约束

初次实现流水线 CPU 时曾出现过无法满足时钟的情况，这是因为 CPU 的逻辑回路太长，延迟太大。检查后发现，是因为我的译码模块的逻辑回路过长。发现问题后，我修改了部分逻辑，**将有些逻辑电路提前到取值阶段实现，将有些逻辑电路延后到执行阶段实现，从而均匀了各个阶段的线路延迟**，最终通过了时钟要求。后续，通过进一步简化逻辑，将时钟频率提高到了 80MHz（优化修改的部分未在整体框图中体现）。

# 4 总结

要求：谈谈学完本课程后的个人收获以及对本课程的建议和意见。请在认真总结和思考后填写总结。

1. 课程学习内容

经过一个月的学习实验，我在老师的指导下完成了一下学习任务：

(1) 编写可以进行八种运算，运行在 RISC-V 指令集 24 条基本下的计算器汇编程序，并且在给给定的 logisim CPU 上成功运行该汇编程序。

(2) 设计实现单周期 CPU，完成单周期 CPU 远程平台 trace 调试和下板验证。

(3) 设计实现流水线 CPU，解决数据冒险和功能冒险问题，完成远程平台 trace 调试和下板验证，通过重新组织简化代码，提高 CPU 运行频率。

2. 收获总结

(1) 这次课程 CPU 设计实验时我们开展的第二次硬件实验课，CPU 设计是我们在老师指导下完成的最大最复杂的硬件实验项目。这次实验，强化了我的硬件程序编程能力，强化了我的项目开发能力，强化了我的耐挫能力。

(2) 这次实验为我积累了宝贵的项目开发经验。在实验开始前，仇老师指导我们先完成 CPU 的整体框架设计——填写数据通路表和控制信号表、绘制 CPU 整体线路图等。这种开发思路令我十分受用，它帮助我们有效地保持开发的连续性和一致性。

(3) 本次项目实验我采用自底向上的开发顺序，先完成底层小木块的功能实现，然后再在顶层模块中将各种小器件组合起来。这种顺序允许我每次专注解决一个局部的问题，同时，通过在代码中添加适量注释增加代码的可读性，我也不会“做了后面忘了前面”。

(4) 学习到了远程平台的登录与使用，温习了常用的 linux 指令。

3. 意见

(1) 这次实验只有一位老师管理我们的班级，感觉老师非常辛苦，经常需要加班为同学们验收，或者指导进度稍微落后的同学。希望之后开课的时候老师资源可以更丰富一些，希望能够招收到助教。

(2) 个人认为，仇老师可以设置各个任务的验收截止时间，其实这样是有助于同学们提高代码完成效率的。我也和其他班的同学进行了一些交流，他们有设置每个任务的验收截止时间，但是其实他们整体完成度也更高，做到流水线的同学也更多。