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：

学生班级：03 班

学生学号：200110320

学生姓名：柯嘉铭

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）

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,lt,bge,lui,jal

CPU 频率：25MHz

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,lt,bge,lui,jal

CPU 频率：100MHz

设计的主要特色（除基本要求以外的设计）

1. 流水线 CPU 实现了使用数据前递解决数据冒险。
2. 流水线 CPU 实现了静态分支预测的功能，即当有跳转指令时，总是预测分支不跳转，当发生跳转时再进行修改 pc、清空流水线寄存器等操作进而实现跳转。
3. 流水线 CPU 实现了使用暂停以及数据前递解决 load_use 冒险。

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：

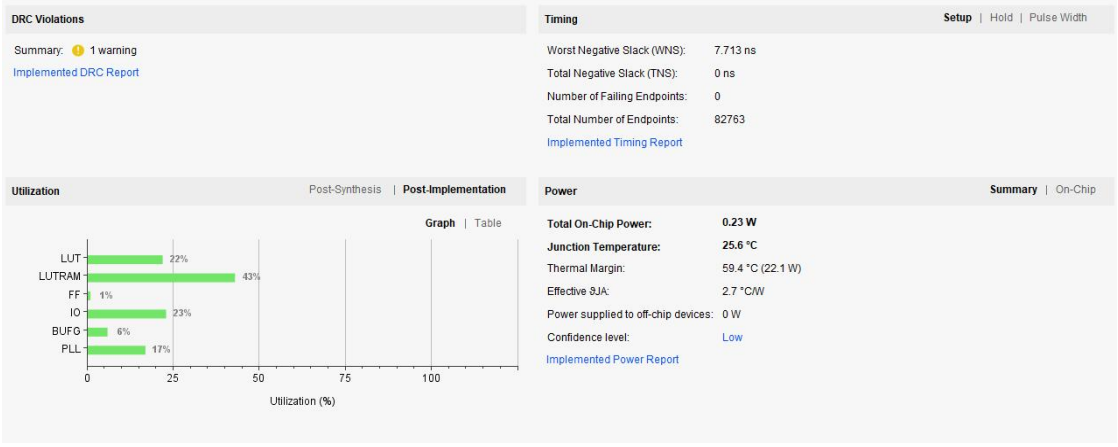

Figure 1 单周期 CPU 资源使用、功耗数据截图

流水线 CPU：

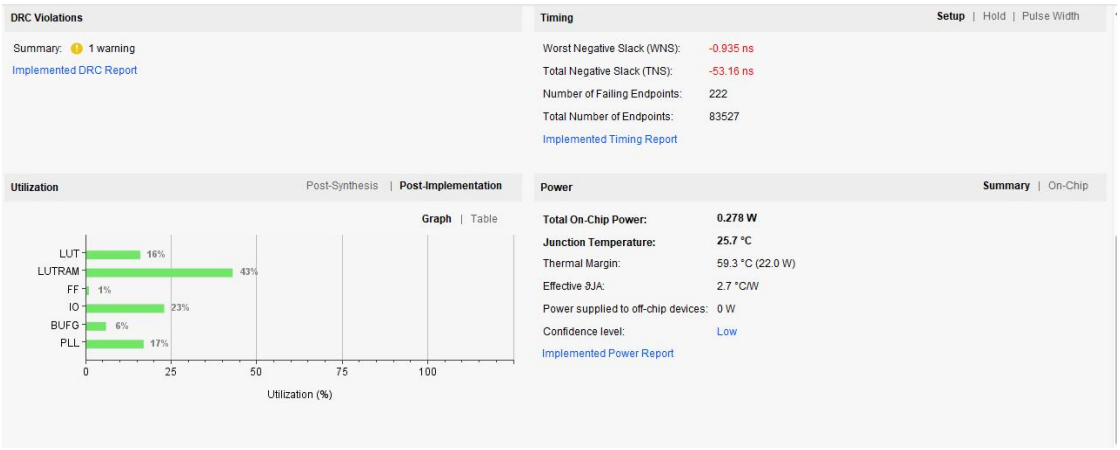

Figure 2 流水线 CPU 资源使用、功耗数据截图

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。

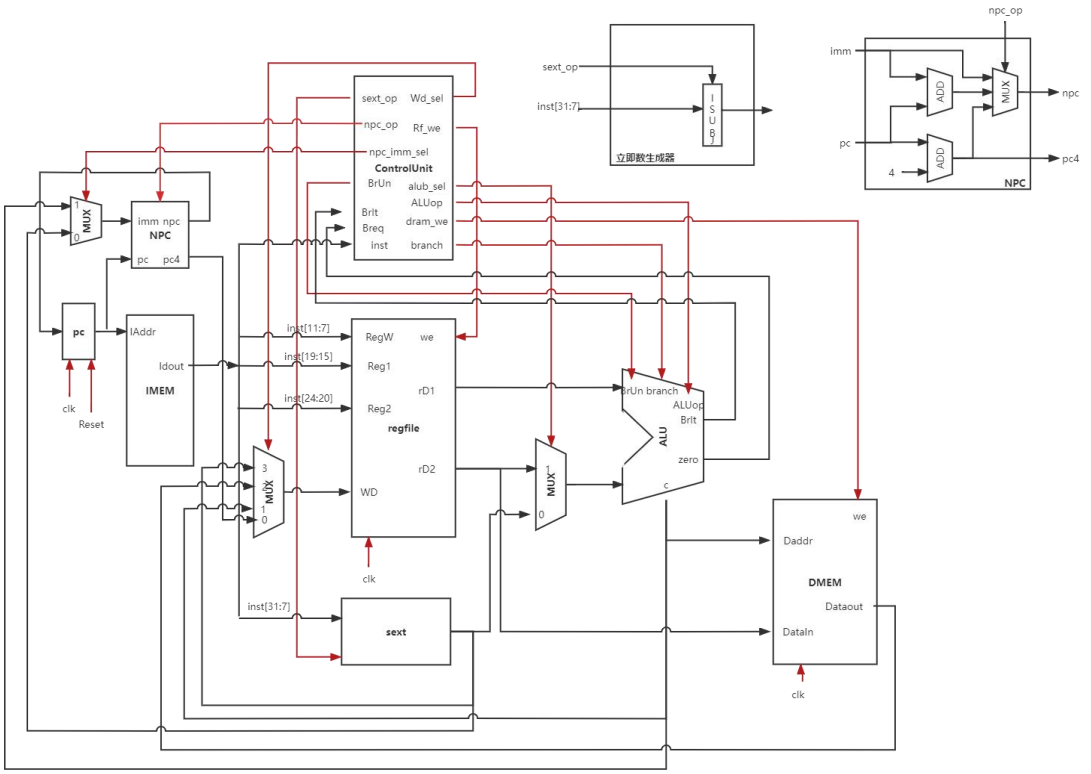

Figure 3 单周期 CPU 整体框图

单周期 CPU 整体框图见 Figure3，其中红线为控制单元 ControlUnit 输出的信号。

各个部件功能含义：

pc：

pc 部件是时序逻辑部件，内部含有一个 32 位的寄存器，用于存储当前正在执行的指令的地址。输入端输入下一个指令地址，当时钟上升沿来到时，寄存器将存储下一个指令的地址并输出。

npc：

npc 为 next pc 的缩写，顾名思义为计算下一条指令地址的组合逻辑部件，内部逻辑见 Figure 3 右上角。输入端 imm 输入来自立即数生成器 sext 生成的立即数或者是由 ALU 计算得出的 rd1 + imm 的结果，由控制单元 ControlUnit 输出的 npc_imm_sel 控制选择；输入端 pc 接受来自 pc 部件的输出；输入端 npc_op 接受来自控制单元 ControlUnit 的输入，选择 npc 生成下一条指令地址。输出端 pc4 输出 pc + 4 的结果，输出端 npc 输出下一条指令的地址。

IMEM：

IMEM 是指令寄存器，用于存储指令。支持数据的异步读取，输入端输入指令的地址，输出端输出相对应的指令。

ControlUnit：

ControlUnit 为 CPU 的控制单元，是组合逻辑部件。输入端输入指令，在内部对指令进行解析，进而得到控制信号并输出。

regfile：

regfile 是 CPU 的寄存器堆，内含 32 个 32 位的寄存器。输入端 RegW 是要写入数据的寄存器编号，Rge1 和 Reg2 是要读出的寄存器编号，读出数据输出分别为 rD1 和 rD2 支持异步读、同步写，其中 we 是控制单元 ControlUnit 输出的寄存器堆写使能信号，wD 接受要写回寄存器 RegW 的数据，该数据可以是 pc + 4、立即数生成器生成的立即数、 ALU 计算的结果或者是从数据存储器读出的数据，由控制单元 ControlUnit 输出的信号 wd_sel 选择。

sext：

立即数生成器，是组合逻辑部件，接受指令的一部分，根据控制单元的 sext_op 信号生成立即数并输出。

ALU：

算术运算单元，是组合逻辑部件，接受两个操作数，根据控制单元的 ALUop 信号判断该执行的运算，此外，控制单元的 brun 信号判断是否进行无符号比较，branch 判断是否进行比较操作。控制单元输出端 ALUc 输出运算结果，zero 当 branch 有效且两个操作数相等时输出 1，否则输出 0；brlt 当 branch 信号有效且操作数 A 小于操作数 B 时，输出为 1，否则为 0（操作数 A 为 ALU 上方的输入，操作数 B 为 ALU 下方的输入）。

DMEM：

数据存储器，是时序逻辑元件，在实验中由 ip 核生成，是一个存储 32 位数据的 64KB 的 RAM。输入端 dram_we 接受来自控制单元的存储器写使能信号， Daddr 接受要读/写存储器的地址， DataIn 接受要写入存储器的数据。 输出端 DataOut 输出从存储器中读出来的数据。

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。

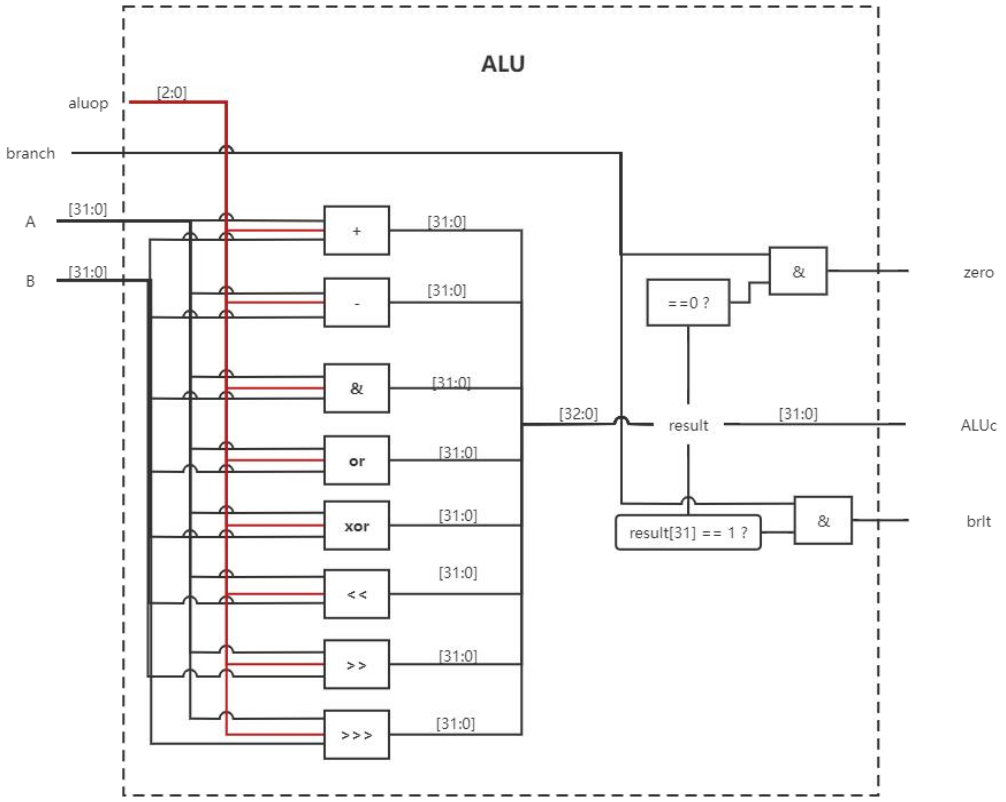

Figure 4 ALU 模块详细设计

ALU 模块详细设计见 Figure4，根据 aluop 对两个 32 位操作数 A 和 B 进行 add，sub，or，and，xor，sll，srl 或 sra 运算。branch 信号有效时，ALU 执行的时 sub 运算，当 result 等于 0 时，zero 为 1，否则为 0；当 result[31] 为 1 时，即得出结果为负数时，brlt 输出为 1，否则为 0。

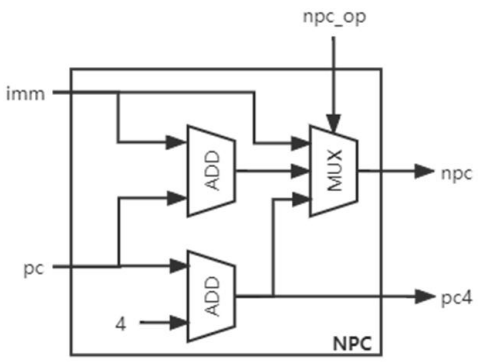

Figure 5 NPC 部件详细设计

NPC 模块详细设计见 Figure 5，由两个加法器和一个多路选择器组成。

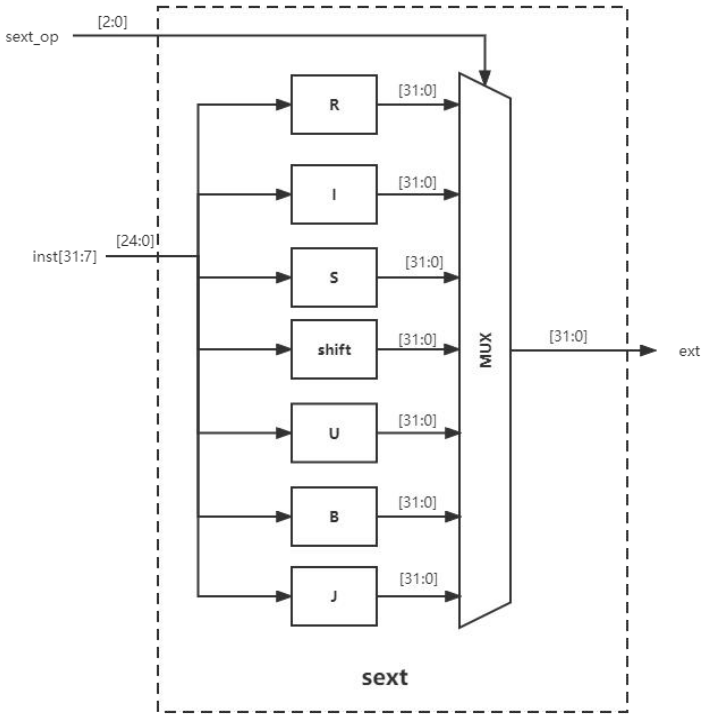

Figure 6 sext 模块详细设计

立即数生成器模块的设计图如 Figure 6 所示，该模块接受指令的 [31:7] 部分，然后根据控制单元的 sext_op 信号选择生成的立即数。

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。

逻辑运算指令：

add 指令：

在 trace 测试中，选取 add 中的指令如下：

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

在前两条指令中，给 x1 和 x2 寄存器均赋值为 1，第三条指令执行后，将 x1 和 x2 寄存器中的值相加，结果存放在 x14 中。

结果波形分析如下：

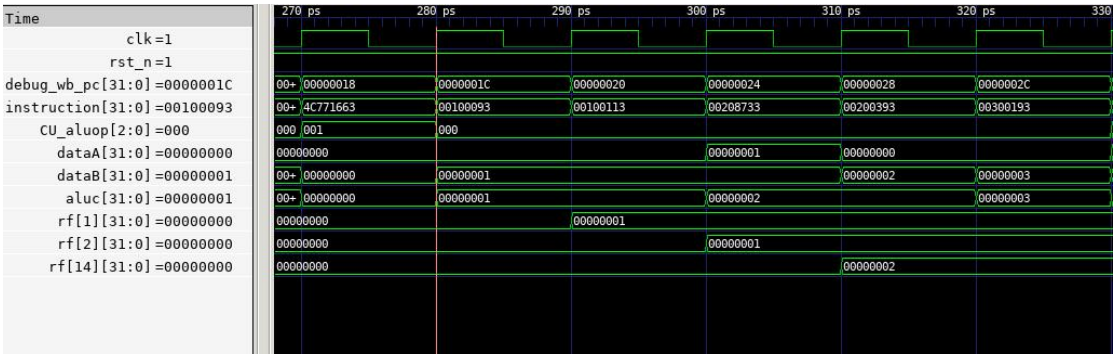

Figure 7 add 指令执行结果

在 Figure7 中，可以看到执行完 pc 地址为 24 的指令后，也就是上面的第 3 条指令后，寄存器 x14 的结果正是寄存器 x1 和 x2 相加的结果。上图中，CU_aluop 表示是来自控制单元的 aluop 信号， aluop 信号为 'b000 时，运算单元执行的是加法操作，在执行 pc 地址为 24 的指令后，alu 的结果 aluc 写回到寄存器堆中编号为 14 的寄存器中。

详细波形分析如下：

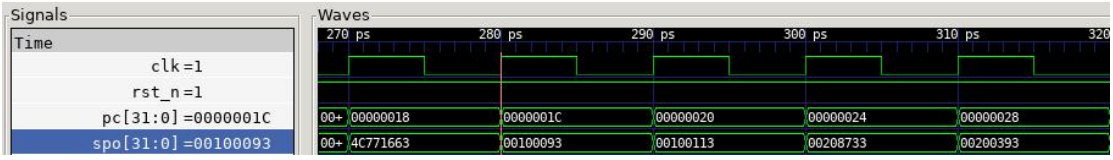

Figure 8 IF 阶段波形

在 Figure 8 中展示的是 IF 阶段的波形，可以看到在接收到 pc 地址后，指令寄存器输出了正确的指令。

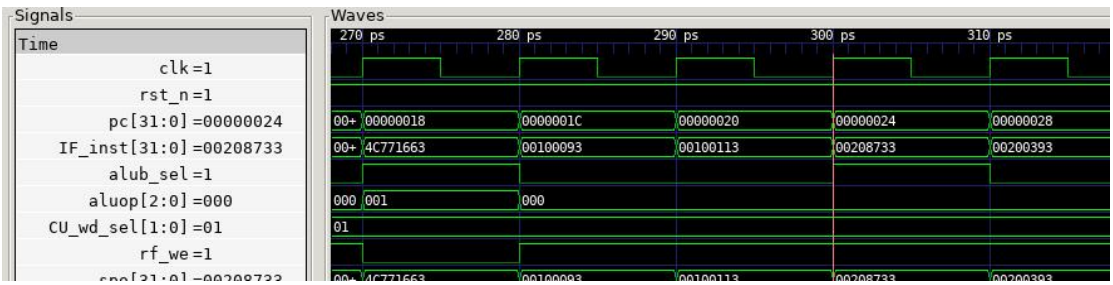

Figure 9 add 指令 CU 模块波形

如上图，控制单元接收到指令后，对指令进行译码，输出的相关信号有：alub_sel 为 1，表示运算单元的 dataB 是来自寄存器堆中读出来的数据；aluop 为 'b000，表示运算单元该执行的是加法操作；CU_wd_sel 为 'b01 表示写回寄存器堆中的数据是运算单元的计算结果； rf_we 为 1，表示寄存器堆中的写使能有效。

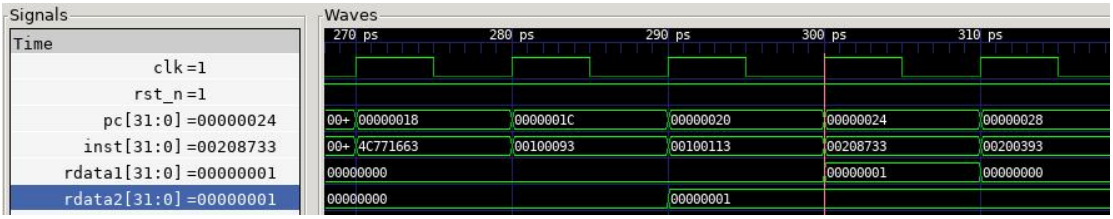

Figure 10 add 指令 ID 阶段波形

ID 阶段接受到指令后，对指令进行译码，读出了寄存器 x1 和 x2 的数据。

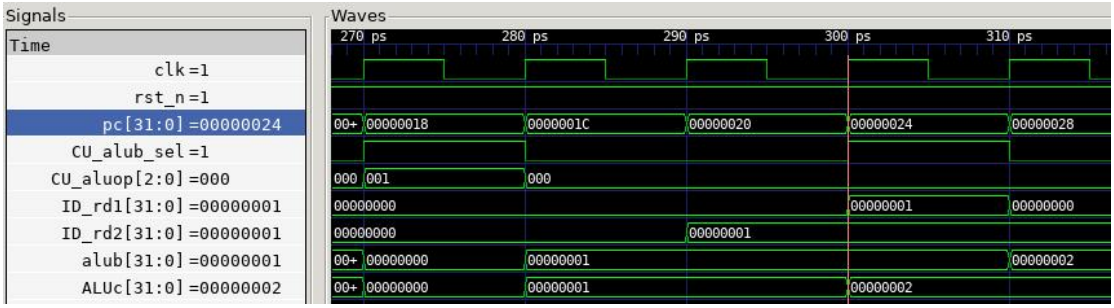

Figure 11 add 指令 EX 阶段波形

在 EX 阶段，接受来自控制单元以及译码阶段的信号后，在 ALU 中进行相关运算，并且将运算的结果输出为 ALUc。

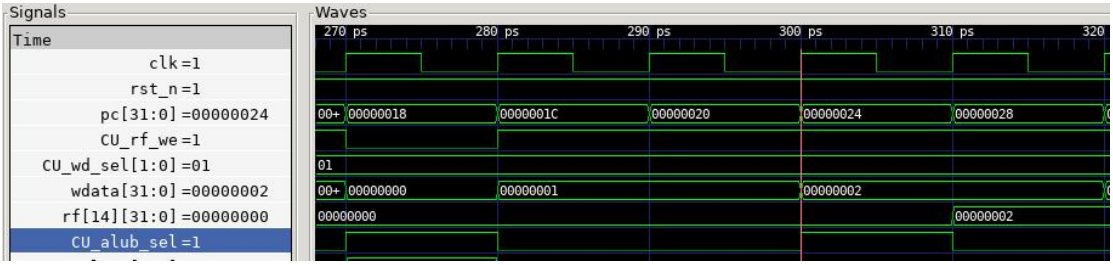

Figure 12 add 指令 WB 阶段波形

由于 add 指令不涉及访问数据存储器，在此只分析 WB 阶段的波形。可以看到在 WB 阶段，CU_rf_we 有效，表明寄存器堆写有效，CU_wd_sel 为 'b01，表明写回寄存器堆的数据是来自 EX 阶段的运算器的结果，可以看到 wdata 与 EX 阶段的 ALUc 相同。由于寄存器堆是时序逻辑元件，因此在下一个时钟上升沿来到时，数据写入了寄存器堆中编号为 14 的寄存器中。

访存指令：

在 trace 测试中，选取 sw 中的指令如下：

```
   4:   000020b7          lui  x1,0x2
   8:   00008093          addi    x1,x1,0 # 2000 <begin_signature>
   c:   00aa0137          lui  x2,0xaa0
  10:   0aa10113          addi    x2,x2,170       #       aa00aa
<_end+0xa9e07a>
  14:   0020a023          sw  x2,0(x1)
  18:   0000a703          lw  x14,0(x1)
```

使用 lui 指令以及接下来的 addi 指令将寄存器堆中编号为 2 的寄存器中的数据设为 0x00aa00aa，然后将 x2 寄存器中的值保存在数据存储器中地址为 0x00002000 中的位置，再使用 lw 指令将此位置中的数据读出来放到寄存器 x14 中。

结果波形分析如下：

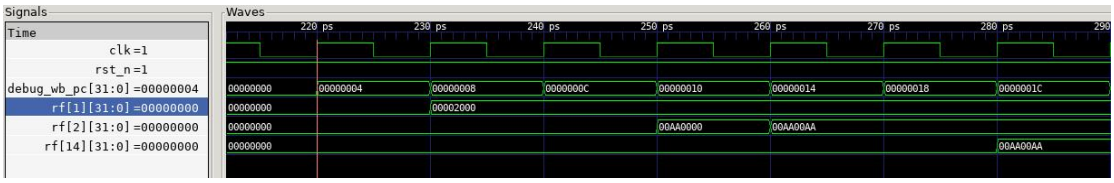

Figure 13 访存结果波形

从波形中可以看到，在执行完地址为 0x18 的指令之后，寄存器堆中的数据就是之前保存在数据存储器中地址为 0x00002000 的数据。

下面对上述指令的最后一条，即 lw x14,0(x1) 进行详细分析：

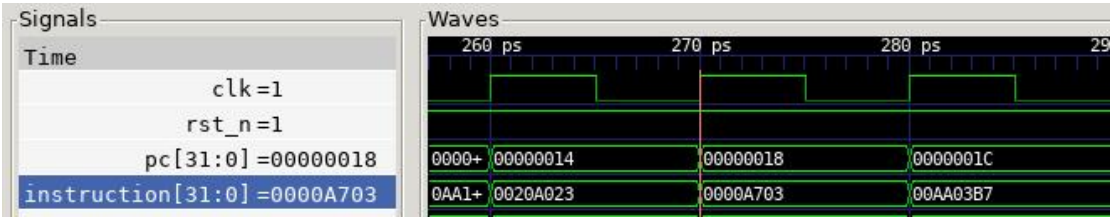

Figure 14 lw 指令 IF 阶段波形

首先在 IF 阶段取指令，根据指令的地址，取出来的指令为 0x0000A703，该指令接下来会被传送到控制单元以及 ID 阶段进行译码。

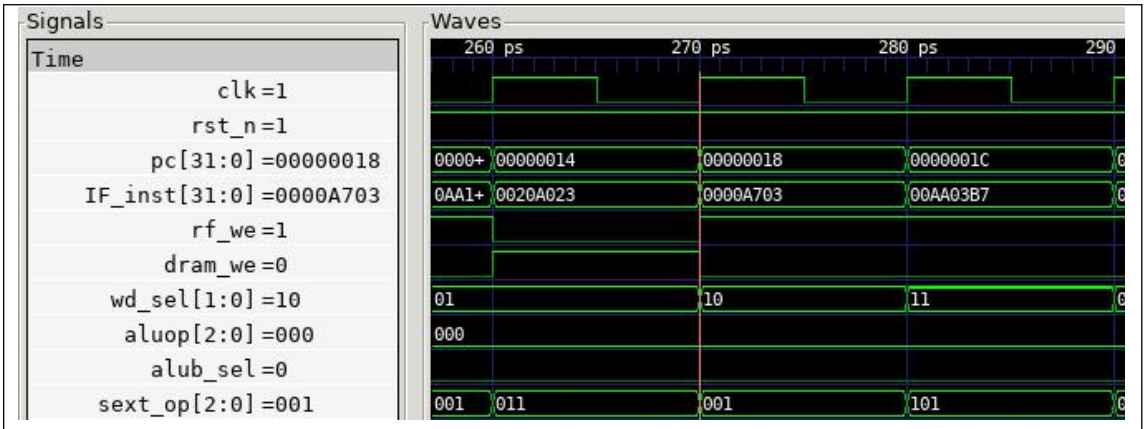

Figure 15 lw 指令 CU 模块波形

接受到 IF 阶段送来的指令后,控制模块会对该指令进行译码:将 rf_we 置为 1,表示写回寄存器堆有效;将 dram_we 置为 0,表示数据存储器写无效,即只能读;将 wd_sel 置为 'b10,表示写回寄存器堆中的数据是从数据寄存器中读出来的数据;将 aluop 置为 'b000,表示此时 alu 需要执行的是加法操作;将 alub_sel 置为 0,表示此时 alu 的第二个操作数是符号扩展的立即数;将 sext_op 置为 'b001,表示立即数生成器对立即数进行 I 型扩展。

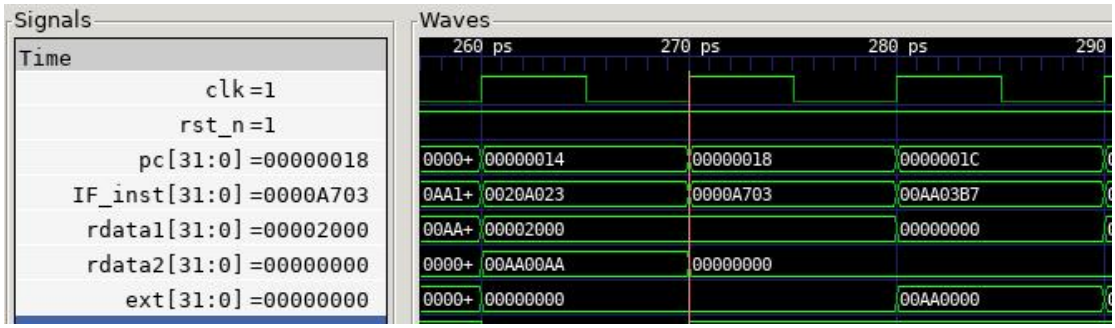

Figure 16 lw 指令 ID 阶段波形

在 ID 阶段,主要的工作是将数据从寄存器堆中读出以及堆立即数进行符号扩展。

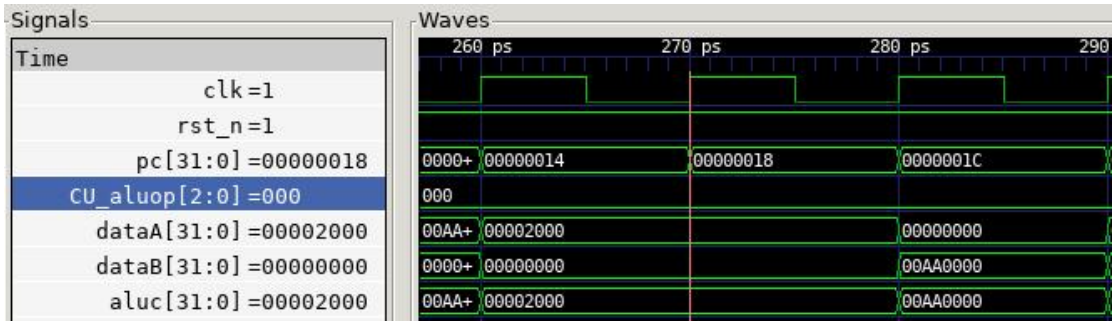

Figure 17 lw 指令 EX 阶段波形

在 EX 阶段,主要是在 alu 中根据 aluop 以及两个操作数进行相关的计算。

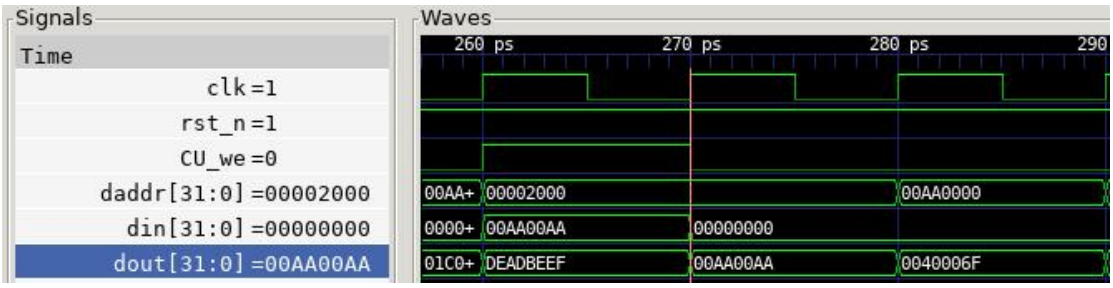

Figure 18 lw 指令 MEM 阶段波形

在 MEM 阶段，上一阶段的运算单元的结果就是要访问的地址，即 daddr，读出的数据为 dout。

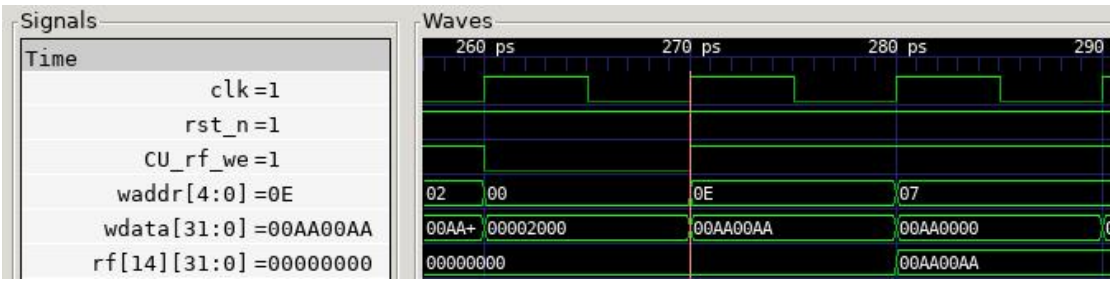

Figure 19 lw 指令 WB 阶段波形

WB 阶段，根据控制控制单元的相关信号、要写回的数据以及要写到的地址将数据写回到寄存器堆中，由于寄存器堆是时序逻辑部件，因此在下一个时钟上升沿到来的时候，数据才能写回到指定的寄存器中。从波形中可以看到在下一个时钟上升沿到来的时候，数据正确地写入了寄存器 x14 中。

分支跳转指令：

在 trace 测试中，选取 beq 指令如下：

```
   8:   00000093          addi    x1,x0,0
   c:   00000113          addi    x2,x0,0
  10:   00208663          beq x1,x2,1c <reset_vector+0x18>
```

上述指令执行后，由于寄存器 x1 中的值与寄存器 x2 中的值相同，因此在执行完 beq 指令后会发生跳转，指令地址会跳转到 0x1c。

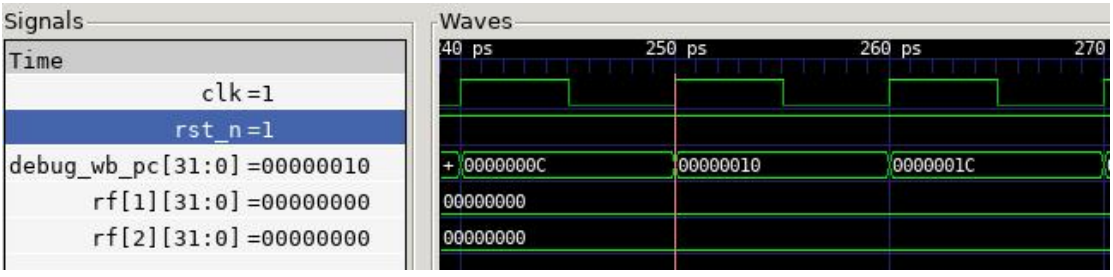

Figure 20 beq 指令结果波形

从结果波形可以看出，beq 指令执行后，pc 发生了跳转。

beq 指令的执行的关键部件是控制单元、alu 以及 npc 模块，下面对这几个模块进行分析。

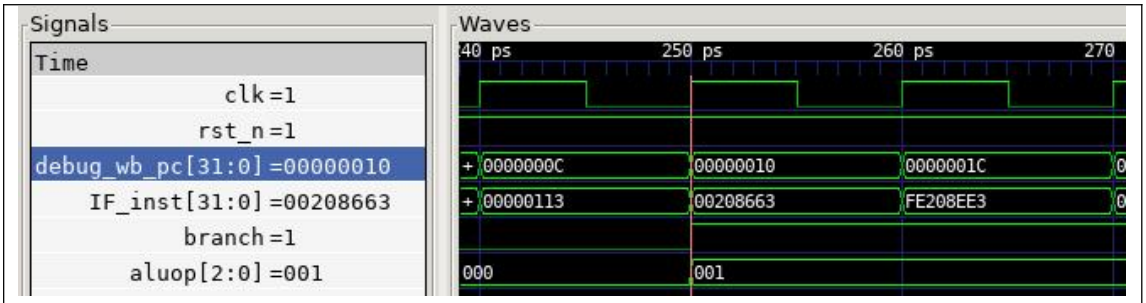

Figure 21 beq 指令 CU 模块波形 1

CU 模块接收到来自 IF 阶段的指令后，对指令进行译码。将 branch 信号置为 1，表示现在执行的是分支跳转指令；将 aluop 置为 'b001，表示 alu 进行的是减法运算。

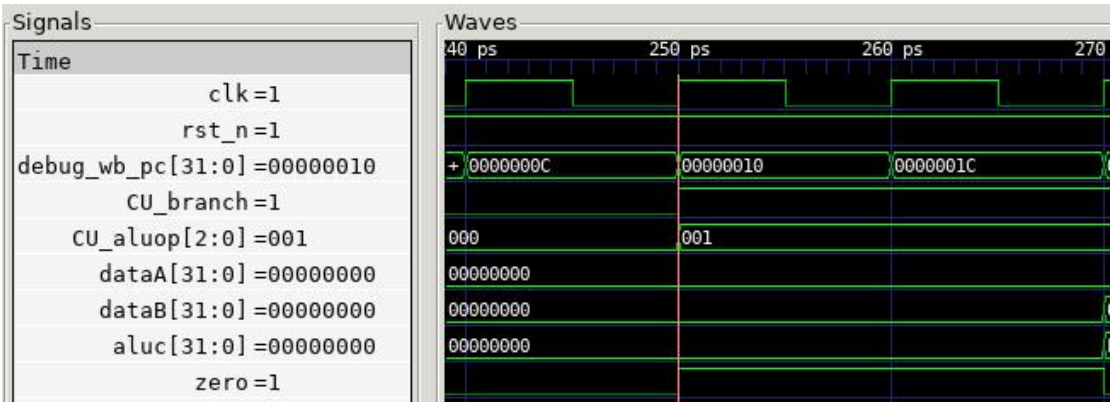

Figure 22 beq 指令 alu 模块波形

alu 模块对两个立即数执行减法操作，得到结果 aulc 为 0。由于 aluc 为 0 且 CU_branch 有效，因此 zero 信号为 1。zero 信号传送到 CU 模块的 breq 输入端上。

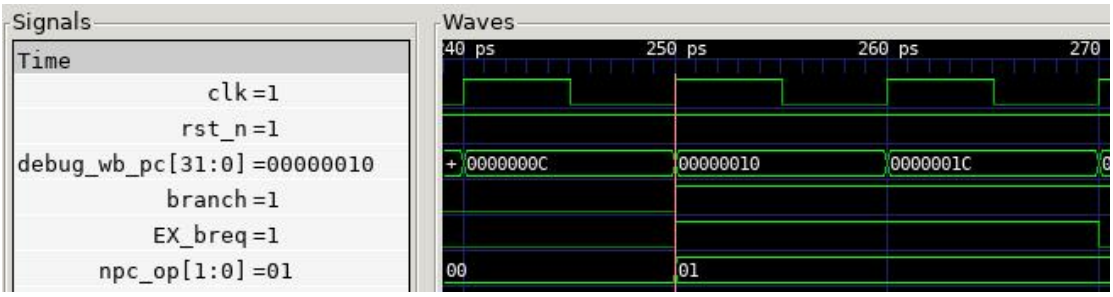

Figure 23beq 指令 CU 模块波形 2

由于 branch 为 1 且 breq 为 1，判断得出指令跳转，将 npc_op 置为 'b01，即跳转到所给 pc + imm。

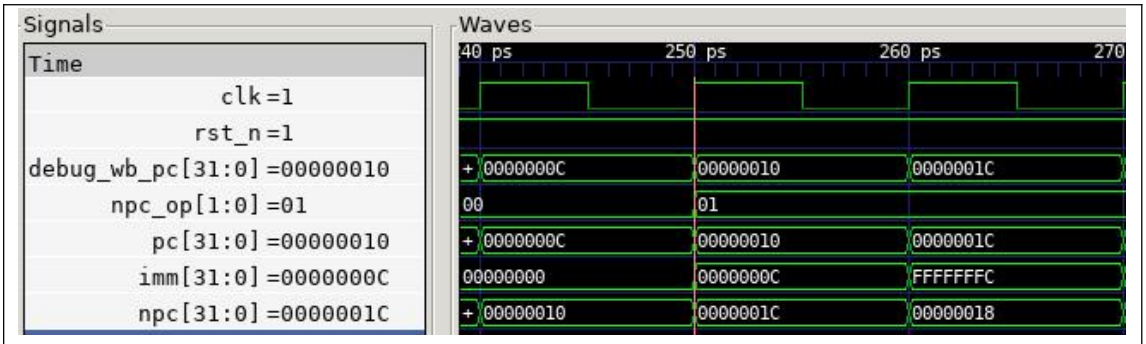

Figure 24 beq 指令 npc 模块波形

从波形中可以看出，输出的 npc 就是应该跳转到的指令地址。从下一条指令的 pc 可以得出指令发生了跳转。

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。

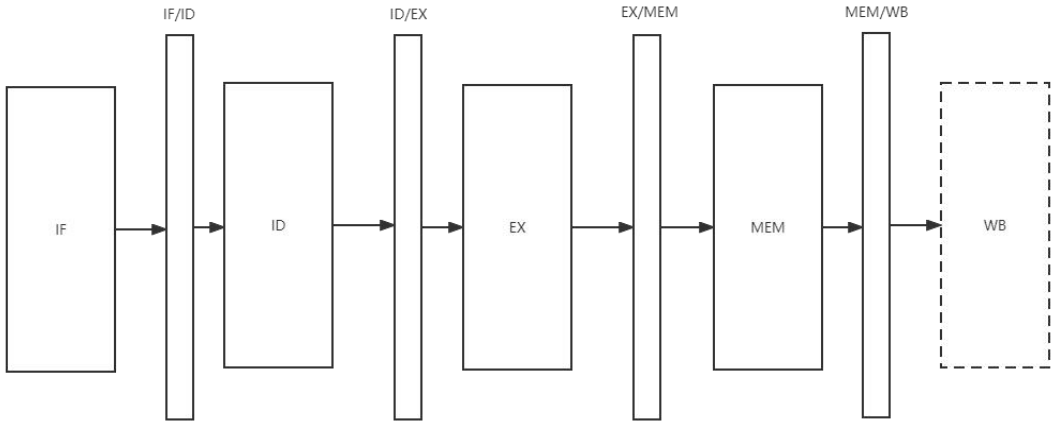

Figure 25 流水线划分

流水线划分如上图。将流水线 CPU 设计为具有 5 级流水，包括 5 个阶段和 4 个流水线寄存器。

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：

该阶段主要完成取地址、取指令的操作。

ID（译码）阶段：

该阶段主要是对指令进行译码、从寄存器堆中取数据以及进行立即数生成。

EX（执行）阶段：

该阶段包含有运算单元 ALU 和计算下一个指令地址的单元 npc 。

MEM（访存）阶段：

该阶段主要是访问数据存储器，进行数据的读写操作。

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。

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。

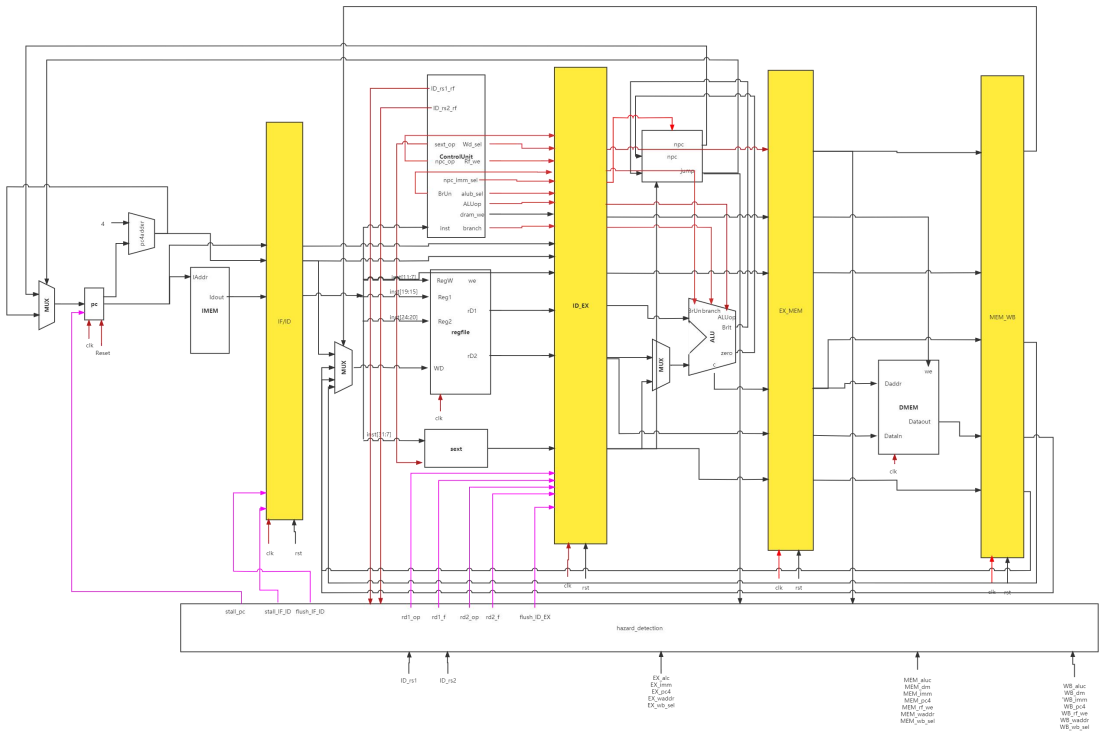

Figure 26 流水线 CPU 整体框图

流水线 CPU 整体框图如上图所示。其中，IF（取指），ID（译码），EX（执行），MEM（访 存），WB（写回）这 5 个阶段内部逻辑已画出。

处理器内部模块功能含义：

IF 阶段：

这一阶段主要进行取指令，放入 IF/ID 寄存器中，同时将 pc 地址和 pc+4 放入 IF/ID 寄存器。

IF/ID：

IF 阶段与 ID 阶段之间的流水线寄存器。

ID 阶段：

译码阶段。这一阶段由三个主要部件组成，分别是根据指令发出控制信号的控制单元 ControlUnit 、寄存器堆 regfile 、立即数生成器 sext 。三个部件主要功能与单周期 cpu 中同名部件主要功能相同。ID 阶段将部分控制信号、在寄存器堆取出的数据以及立即数生成器中生成的立即数送到 ID/EX 寄存器中。

ID/EX：

ID 阶段与 EX 阶段之间的流水线寄存器。

EX 阶段：

执行阶段。这一阶段主要由两个部件组成，分别是计算下一个指令地址的部件 npc 以及进行运算的单元 ALU 。EX 阶段计算出下一条指令的地址后将地址

送到 IF 阶段，将运算单元的结果送到 EX/MEM 寄存器中。

EX/MEM：
EX 阶段与 MEM 阶段的流水线寄存器。

MEM 阶段：
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。

MEM/WB：
MEM 阶段与 WB 阶段的流水线寄存器。

WB 阶段：
在上图中，WB 阶段并没有显式地画出来。在 WB 阶段，主要是将流水线寄存器 MEM/WB 中要写回寄存器堆中的数据传送到 ID 进而写回到寄存器堆中。

Hazard_detection：
冒险检测以及处理模块。检测数据冒险、控制冒险等冒险，控制流水线寄存器进行停顿或者清空，能够使用数据前递的方式解决冒险。

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。

流水线 CPU 部分模块与单周期设计类似。

下图是 hazard_detection 模块的设计图：

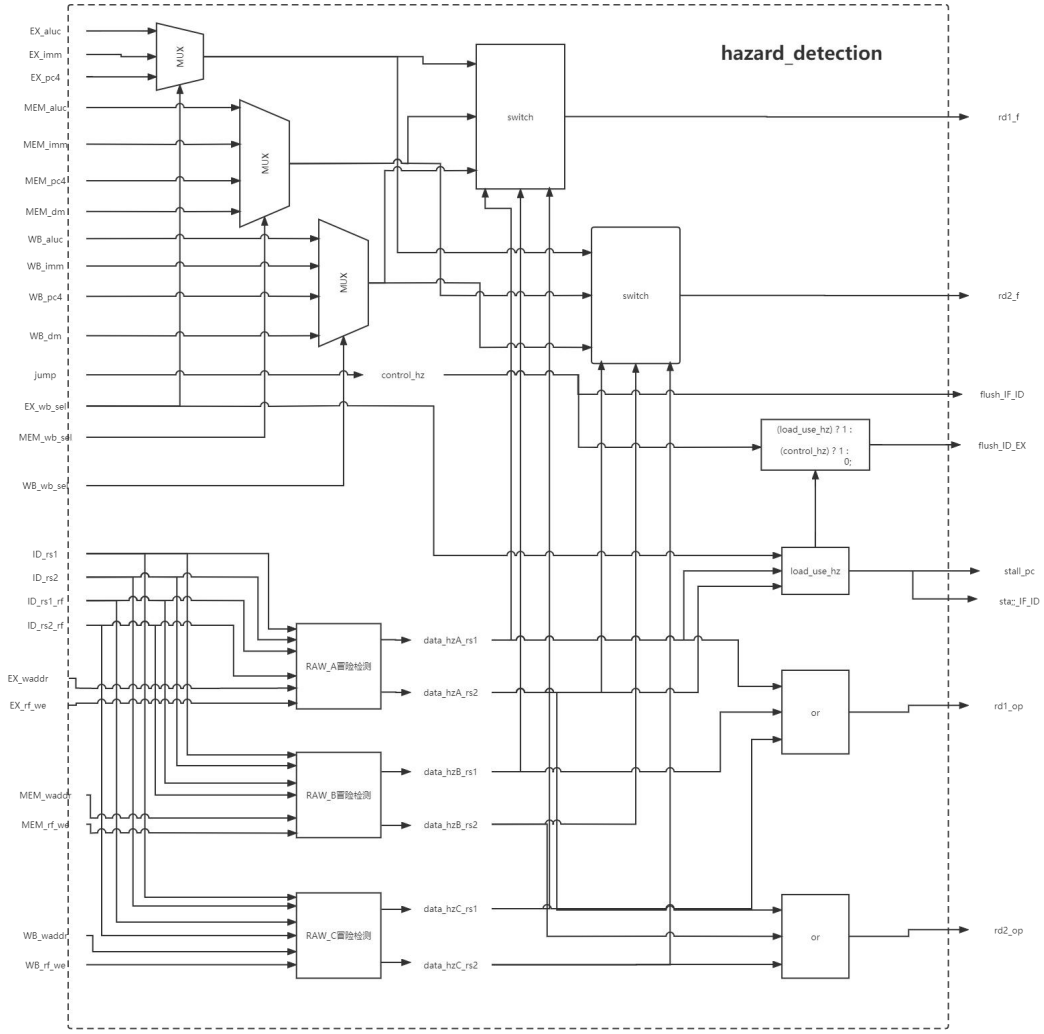

Figure 27 hazard_detection 模块图

数据冒险检测：

RAW 型数据冒险共有三种情形，如下：

相邻指令发生 RAW 冒险 —— Figure28 中，第 2 条指令在译码阶段访问的寄存器与第 1 条指令在执行阶段将要写回的寄存器冲突。

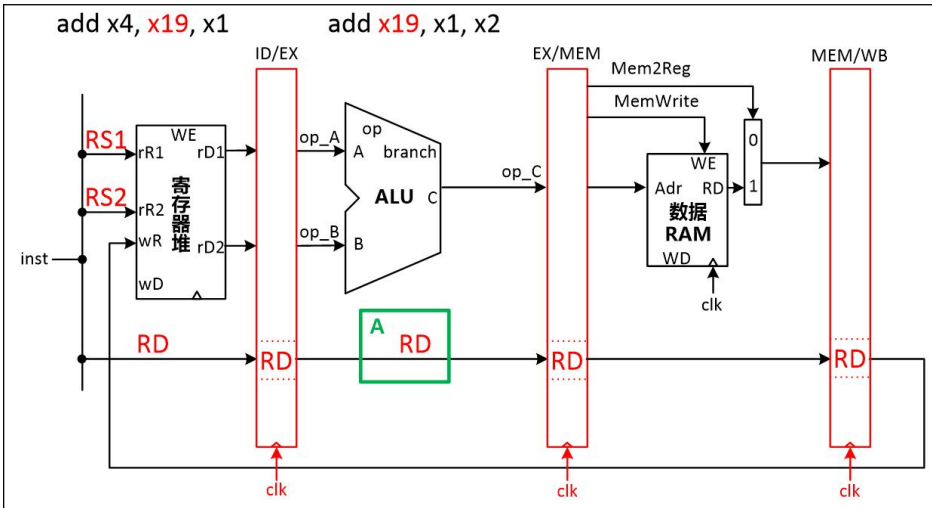

Figure 28 RAW 情形 A

间隔 1 条指令发生 RAW 冒险 —— Figure 29 中，第 3 条指令在译码阶段访问的寄存器与第 1 条指令在访存阶段将要写回的寄存器冲突。

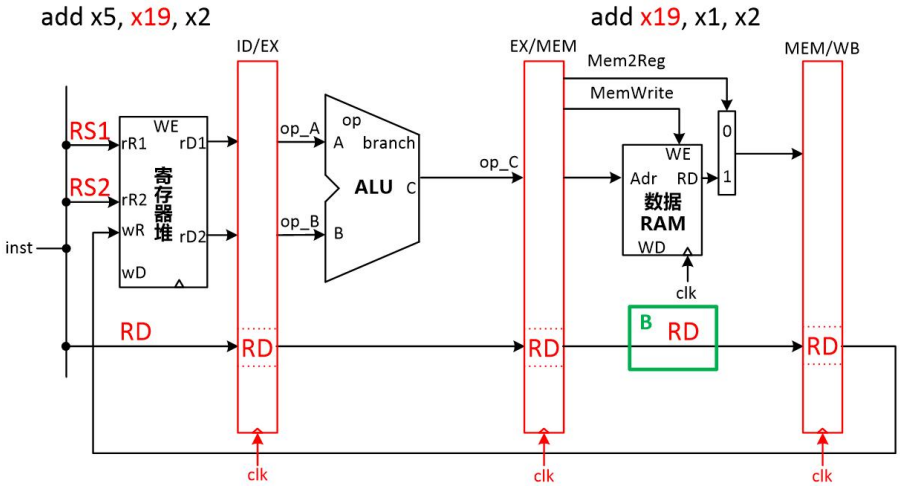

Figure 29 RAW 情形 B

间隔 2 条指令发生 RAW 冒险 —— Figure 30 中，第 4 条指令在译码阶段访问的寄存器与第 1 条指令在写回阶段将要写回的寄存器冲突。

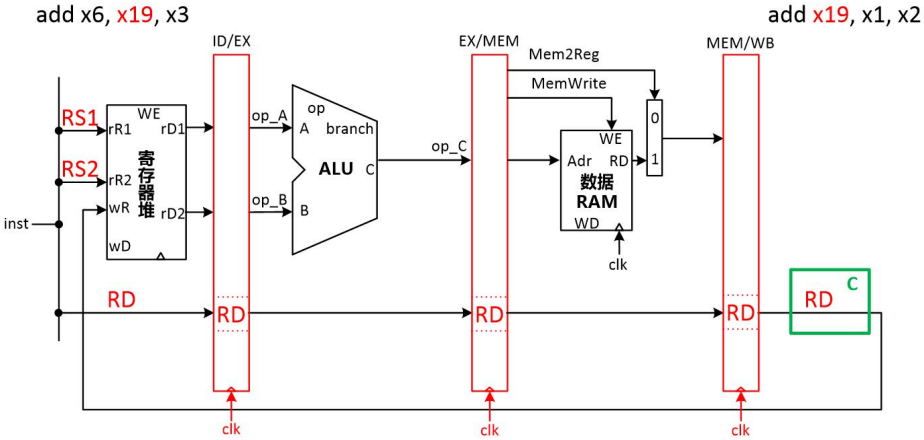

Figure 30 RAW 情形 C

具体的检测逻辑代码如下：

```
// RAW 情形 A
assign data_hzA_rs1 = ((EX_waddr == ID_rs1) & EX_rf_we & ID_rs1_rf & (ID_rs1 != 5'b0));
assign data_hzA_rs2 = ((EX_waddr == ID_rs2) & EX_rf_we & ID_rs2_rf & (ID_rs2 != 5'b0));

// RAW 情形 B
assign data_hzB_rs1 = ((MEM_waddr == ID_rs1) & MEM_rf_we & ID_rs1_rf & (ID_rs1 != 5'b0));
assign data_hzB_rs2 = ((MEM_waddr == ID_rs2) & MEM_rf_we & ID_rs2_rf & (ID_rs2 != 5'b0));

// RAW 情形 C
assign data_hzC_rs1 = ((WB_waddr == ID_rs1) & WB_rf_we & ID_rs1_rf & (ID_rs1 != 5'b0));
assign data_hzC_rs2 = ((WB_waddr == ID_rs2) & WB_rf_we & ID_rs2_rf & (ID_rs2 != 5'b0));
```

其中，ID_rs1_rf 表示是否用到 rs1，EX_waddr 表示在 ID/EX 寄存器中的写回寄存器堆的地址，EX_rf_we 表示在 ID/EX 寄存器中的写寄存器堆使能信号，其他变量类似。

Load_use 冒险检测：

```
wire load_use_hz = (data_hzA_rs1 | data_hzA_rs2) & (EX_wb_sel == `wdDM);
```

即当发生 RAW 情形 A 的冒险且写回的数据是从数据存储器中读出来的数据时，发生 load_use 型冒险。

冒险的解决方法如下：

- 将ID/EX寄存器中控制信号全置为0(RegWrite, MemWrite...)
  - 被停顿的指令在EX, MEM, WB 阶段执行空指令 nop
  - 不会有寄存器或者存储器被写入数据
- 禁止PC寄存器和IF/ID寄存器内容发生改变
  - ID阶段的寄存器会继续使用IF/ID寄存器中相同字段读寄存器
  - 下一条指令会重新取指
  - 1个时钟周期的停顿，能够让ld指令的MEM阶段完成
    - 就可以把取到的数据前递到EX阶段

Figure 31 load_use 冒险解决方法

控制冒险检测：

```
wire control_hz = jump;
```

由于采用静态分支预测，因此在 npc 计算出指令地址要发生跳转时，jump 信号有效，则检测到发生控制冒险。

前递：

```
wire [31:0] EX_f = (EX_wb_sel == `wdALUC) ? EX_aluc :   // EX_f 表示前递的是 EX 阶段的结果
                   (EX_wb_sel == `wdImm) ? EX_imm :
                   (EX_wb_sel == `wdPC4) ? EX_pc4 :
                   0;
```

```
wire [31:0] MEM_f = (MEM_wb_sel == `wdALUC) ? MEM_aluc : // MEM_f 表示前递的是数据存储器
                                                         //读出的数据
                    (MEM_wb_sel == `wdImm) ? MEM_imm :
                    (MEM_wb_sel == `wdPC4) ? MEM_pc4 :
                    MEM_dm;

wire [31:0] WB_f = (WB_wb_sel == `wdALUC) ? WB_aluc : // WB_f 表示前递的是将要写回寄存器
                                                      //堆中的数据
                   (WB_wb_sel == `wdImm) ? WB_imm :
                   (WB_wb_sel == `wdPC4) ? WB_pc4 :
                   WB_dm;


always @ (*) begin
rd1_f = (data_hzA_rs1) ? EX_f :   // 根据数据冒险类型选择前递的数据
                         (data_hzB_rs1) ? MEM_f :
                         (data_hzC_rs1) ? WB_f :
                         32'b0;

rd2_f = (data_hzA_rs2) ? EX_f :
                         (data_hzB_rs2) ? MEM_f :
                         (data_hzC_rs2) ? WB_f :
                         32'b0;
```

流水线的暂停与清空：

在解决冒险时，需要暂停 pc 以及部分流水线寄存器；由于采用静态分支预测，所以当发生控制冒险时，需要将部分流水线寄存器清空：

```
stall_PC = (load_use_hz) ? 1 : 0; // 有效时，暂停 pc 模块
stall_IF_ID = (load_use_hz) ? 1 : 0; // 有效时，暂停 IF/ID 流水线寄存器
flush_IF_ID = (control_hz) ? 1 : 0; // 有效时，清空 IF/ID 流水线寄存器
flush_ID_EX = (load_use_hz) ? 1 :   // 有效时，清空 ID/EX 流水线寄存器
                              (control_hz) ? 1 :
                              0;
```

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。

控制冒险：

在 trace 测试中，选取 beq 指令如下：

```
   8:   00000093            addi    x1,x0,0
   c:   00000113            addi    x2,x0,0
  10:   00208663            beq x1,x2,1c <reset_vector+0x18>
```

波形如图：

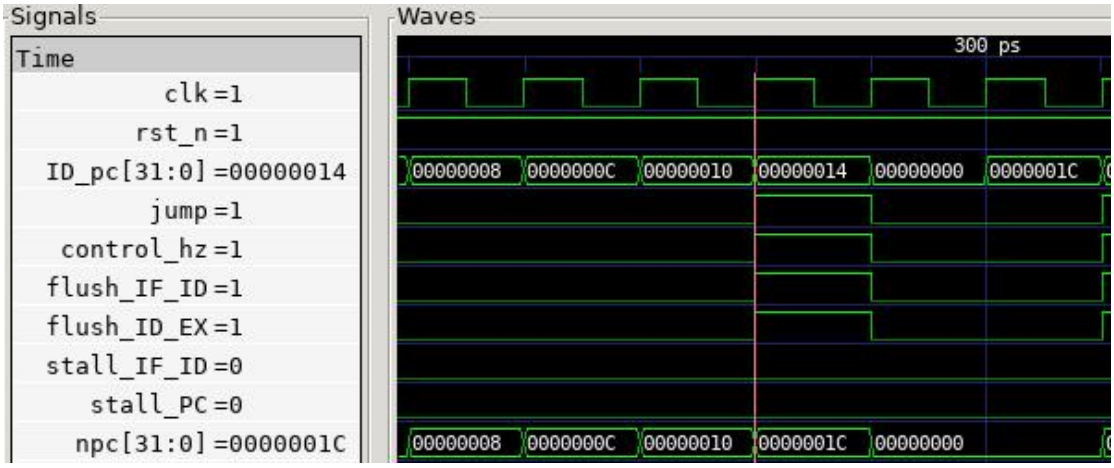

Figure 32 控制冒险波形

在 hazard_detection 中检测到控制冒险，清空流水线寄存器 IF/ID 和流水线寄存器 ID/EX，将 npc 中计算得出的要跳转到的指令地址赋给 pc。

由于 RISC 经典五级流水“IF-ID-EX-MEM-WB”不存在 WAR 和 WAW 的数据冒险，因此下面只分析 RAW 冒险的三种情形。

RAW 情形 A：

在 trace 测试中，选取 add 指令如下：

```
   8:   00000113            addi    x2,x0,0
   c:   00208733            add x14,x1,x2
```

上面第一条指令将数据写回 x2 中，第二条用到了 x2 作为操作数，发生了 RAW 情形 A 数据冒险。

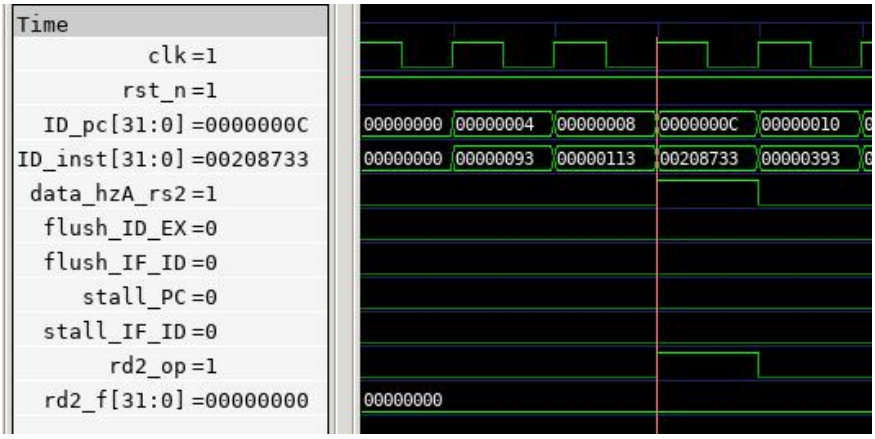

Figure 33 RAW_A

如上图，在 hazard_detection 模块检测出了 RAW_A 数据冒险，并且使用前递解决了数据冒险。

RAW 情形 B：

在 trace 测试中，选取 add 指令如下：

```
   4:   00000093              addi    x1,x0,0
   8:   00000113              addi    x2,x0,0
   c:   00208733              add x14,x1,x2
```

第一条指令将数据写回到寄存器 x1 中，第三条指令使用到 x1 作为操作数，因此产生了 RAW 情形 B 的数据冒险。

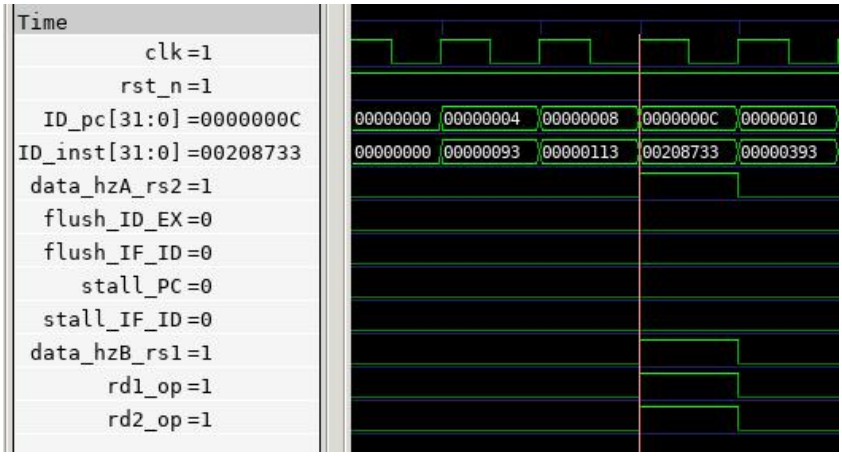

Figure 34 RAW_B

如上图，在 hazard_detection 模块检测出了 RAW_B 数据冒险，并且使用前递解决了数据冒险。

RAW 情形 C：

在 trace 测试中，选取 add 指令如下：

```
  f8:   800000b7              lui  x1,0x80000
  fc:   00008137              lui  x2,0x8
 100:   fff10113          addi    x2,x2,-1 # 7fff <_end+0x5fff>
 104:   00208733              add x14,x1,x2
```

在第一条指令中，要将数据写回到寄存器 x1 中，在第四条指令中使用到了 x1 作为操作数，因此发生了 RAW 情形 C 的数据冒险。

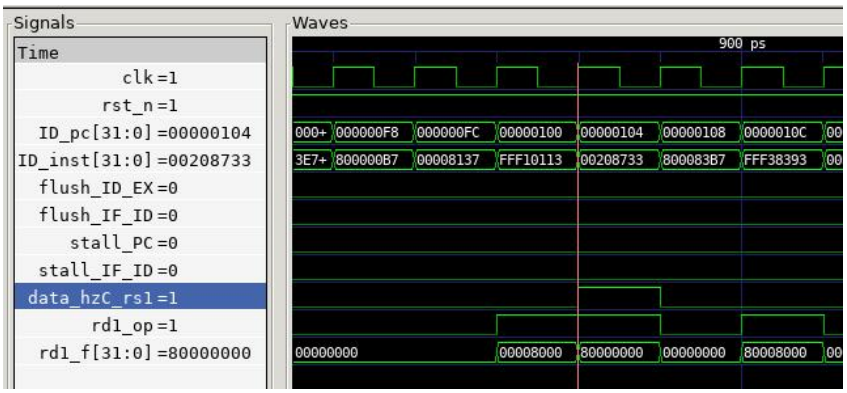

Figure 35 RAW_C

如上图，在 hazard_detection 模块检测出了 RAW_C 数据冒险，并且使用前递解决了数据冒险。

Load_use 冒险：

选取 trace 测试中的 lw 指令：

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

第一条指令使用 lw 指令从数据存储器中载入数据到寄存器 x14 中，第二条指令使用 x14 作为操作数，于是发生了 load_use 冒险。

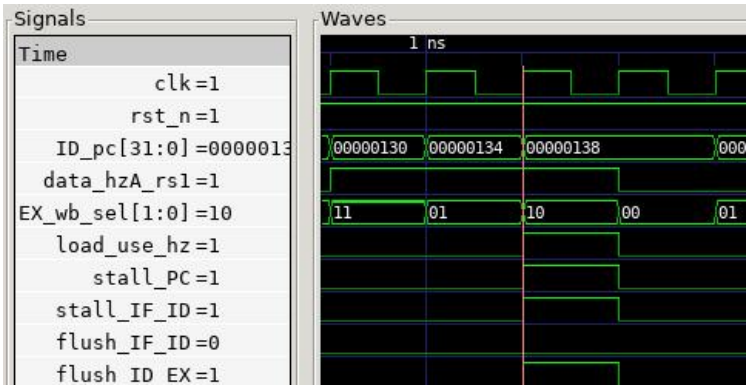

Figure 36 load_use 冒险

如上图，在 hazard_detection 模块中，由于发生了 RAW 情形 A 数据冒险且写回的数据为从寄存器堆中读出来的数据，因此检测出了 load_use 冒险，使用停顿 pc 和 IF/ID 流水线寄存器，并且清空 ID/EX 流水线寄存器中数据来解决冒险。

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。

1. 在设计单周期 CPU 时，没有考虑到现实中数据存储器并不是集成在 CPU 内部的，虽然能过 trace 测试，但是考虑到现实情况，在下板时将这一点纠正了过来。
2. 在实现流水线 CPU 时，由于忘记了 trace 测时的复位信号与下板时的复位信号相反而导致在下板的时候花了较长时间。

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。

个人收获：

在经过了这一次实验之后，我对 CPU 的设计、实现以及工作流程有了更加深入的了解，同时对 verilog 的编写更加熟悉。通过这一次实验，我对计算机系统有了更加深入地理解。

对课程的建议：

建议课程能够开展在考试之后（可能由于开学延期导致夏季学期的开展与考试冲突），这样的话更有利于同学们投入到实验之中。