Lab3 简易单周期 CPU 实验

黄奔皓*

(上海交通大学 IEEE)

目录

1 **实验目的** 2

2 **设计思路概述** 2

3 **文件树** 2

3.1 结构框架 2

3.1.1 lw 指令 3

3.1.2 sw 指令 3

3.1.3 R-Type 和 I-Type 指令 4

3.1.4 Branch 指令 5

3.1.5 jal 和 jalr 指令 5

3.1.6 lui 指令 7

3.2 设计过程中的重点问题 7

4 **实验结果演示与总结** 9

5 **拓展思考部分** 11

5.1 要求 11

5.2 数据通路修改 11

*感谢陈老师的悉心辅导

1 实验目的

本次实验我们将在 lab1 实现的 datapath 和 lab2 实现的 control unit 的基础上，引入数字逻辑课程中所实现的多路选择器、加法器等门级组件。经过 IP 例化指令，实现单周期 CPU 的功能仿真。我们需要：

1. 掌握不同类型指令在数据通路中的执行路径。
2. 掌握 Vivado 仿真方式。

2 设计思路概述

3 文件树

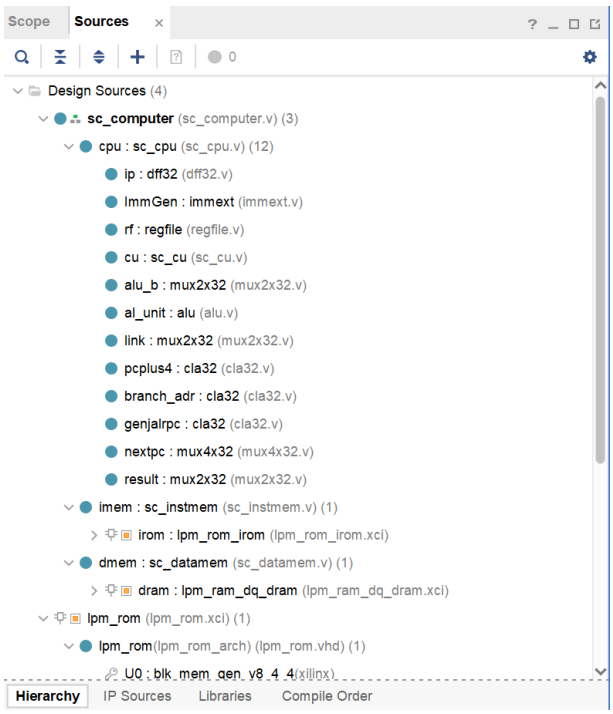

Figure 1: 项目文件树

3.1 结构框架

为实现单周期 cpu，并成功运行 lab2.1 中给出的 RISC-V 代码，我们需要实现以下 6 个指令的数据通路的正确连接。此外，我们还要格外注意 PC 的更新方式。

3.1.1 lw 指令

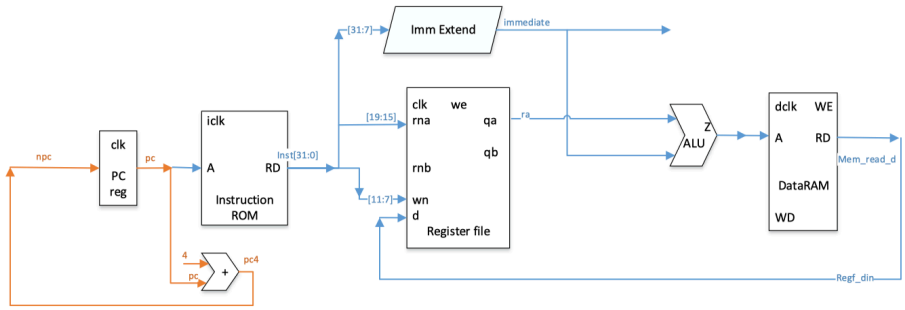

Figure 2: lw 指令的数据通路

第一步 取指，将 PC(即指令地址) 输出至 Instruction Memory(指令存储器) 的 A 端口。lw 指令的汇编格式为：lw rd,imm(rs1)，其中 rs1(base 基地址) 为指令 [19:15] 所指向的寄存器值，imm(是地址偏移) 为指令 [31:20]。

第二步 将 rs1 寄存器的值加上符号扩展后的地址偏移立即数 immediate 得到访存的地址，根据地址从存储器中读取 1 个字 (连续 4 个字节) 的值写入到 rd 寄存器中。

第三步 根据 lw 指令的定义，将指令存储器读出的指令 ([31:0]) 中 [19:15] 连接至寄存器堆的第一个输入地址 rna。

3.1.2 sw 指令

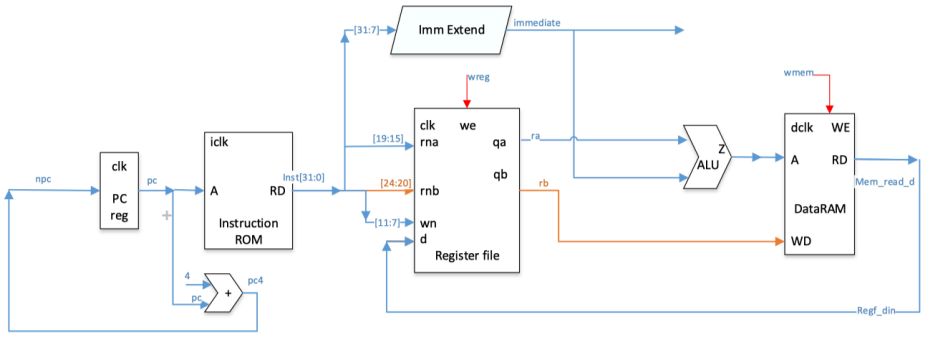

Figure 3: sw 指令的数据通路

sw 指令的定义：sw rs2,imm(rs1)。从指令格式和译码的数据域来看，与 lw 无异。其区别在于，lw 需要从数据存储器读取数据并写入 regfile ，而 sw 仅需要将数据写入数

据存储器。计算地址的 Addr = GPR[rs1]+ sign_extend(imm) 不变，但 rs2 变为读取寄存器，因而需要将其连接至 rnb，读出的数据 RD2 信号连接至数据存储器的 WD 端口。

注意 关于 wmem 和 wreg 控制信号的值，值为 1 时使能，为 0 时不使能。如图 3 所示，在进行存储器读写时，wmem 需要使能，置 1。此外，在进行 sw 写操作，ReadData 的值并不会消失，而是取决于使用的 Memory 类型（可以假设会有随机值读出）。倘若 wn 同时将指令的 [11:7] 写入，这时将会有错误的值写到寄存器堆中。为了避免这种情况，需要将 wreg 置为 0，此时无法对寄存器堆进行写操作。

推论 所有不对寄存器堆进行写操作的指令都应该将 regfile 的写使能关闭；所有不对 Mem 进行读写的指令，也应当关闭其使能端口。

3.1.3 R-Type 和 I-Type 指令

R-type 指令 (XXX rd,rs1,rs2) 将 rs1,rs2 所在寄存器的值进行相应运算后，存入 rd 中。I-type 指令 (XXX rd,rs1,imm) 与 R-type 的差异只在于后者将 rs2 换成了一个立即数。

加入多路选择器 为了区分两种 type 的 ALU 操作，我们需要加入多路选择器 Mux。对于 R-type 指令，ALU 输入为 ra,rb,(指令的 rs1,rs2)，分别对应 alua，alub。对于 I-type 指令，ALU 输入为 ra,immediate,(指令的 rs1 和扩展后 imm)，因此需要在 alub 处加入多路选择器，选择来源为寄存器堆 registerfile 的 qb 输出口信号 rb 还是符号扩展后的立即数 immediate，控制信号为 aluimm。

注意 为了区分寄存器写入 data 的来源，我们还需要加入一个 mux 来控制 regf_din 来源为 ALU 或数据存储器，控制信号为 m2reg。

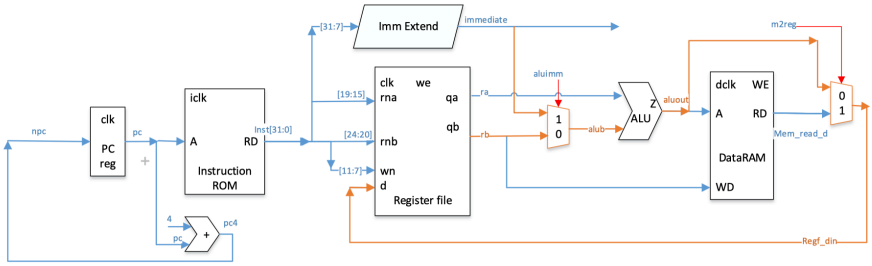

Figure 4: R-type 和 I-type 指令的数据通路

3.1.4 Branch 指令

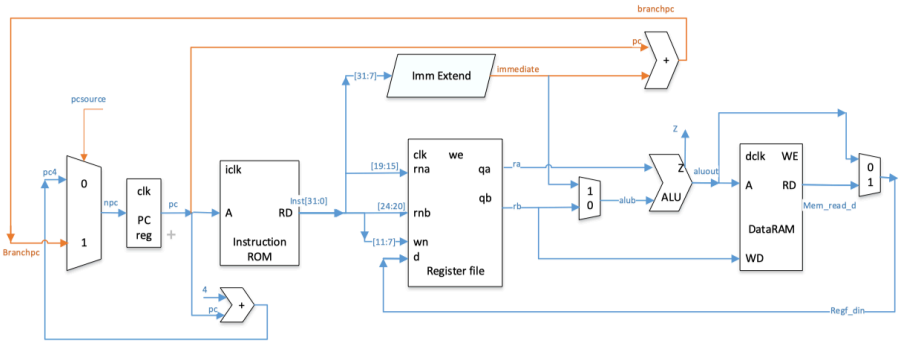

Figure 5: Branch 指令的数据通路

Branch 指令的执行需要三步操作，分别为条件判断，偏移计算，PC 转移。

第一步 条件判断：需要判断 rs1、rs2 所在的寄存器值是否相等，这个功能已在 lab2 的 sc_cu 中实现。

第二步 偏移量的计算：target_off set = sign_extend(imm) << 1 (left shift 1 digit)

第三步 PC 的转移：$PC \leftarrow PC +$ target_off set 根据图 6 中的橙色数据通路进行连接和 pc 的更新即可。

3.1.5 jal 和 jalr 指令

jal **跳转指令** jal 跳转指令实际为无条件跳转，其格式为 jal rd，imm, 包含两个步骤：

第一步: 将当前指令的下一条指令之地址指针 PC+4 存入 rd 寄存器（该值即为函数调用时的返回地址指针）。

第二步: 计算 jalpc 地址，更改 PC。其跳转目标 jalpc 可以沿用上一节的 branchpc 信号，由该指令对应指令 PC 与 immediate 相加得到。

jalr **指令** 是跳转后返回指令，其格式为 jalr rd, imm(rs1)，包含两个步骤：

第一步: 将当前指令的下一条指令之地址指针 PC+4 存入 rd 寄存器

第二步: 将 PC 更改为新的跳转地址，其值来自于 rs1 寄存器的值 ra（基地址）和立即数 immediate（偏移量）的和。

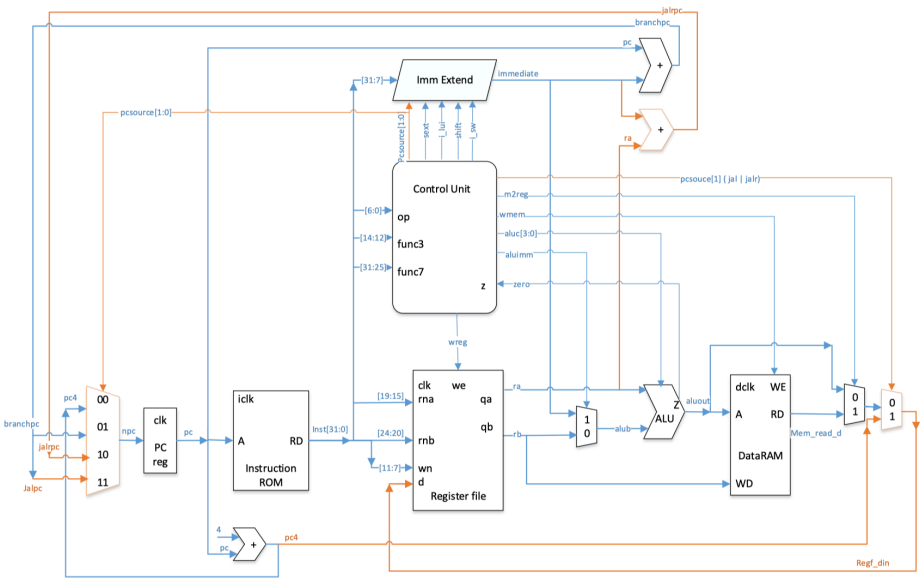

Figure 6: jal 和 jalr 指令的数据通路

注意 随着 PC 的可能来源变为 4 个，原来的二选一多路选择器需要改为 4 选 1，下一条 PC 的值 npc 由 pcsource 根据指令类型选定。具体可以定义如下：

Table 1: nextpc 与 pcsource 和指令类的关系

pcsource[0:1]	instruction	npc(next pc)
0 0	other instruction	pc4
0 1	beq, bne	branchpc
1 0	jalr	jalrpc
1 1	jal	jalpc

相应的代码片段如下：

```
// 产生下一个pc
  cla32 pcplus4(pc, 32'h4, 32'b0, p4); // pc = pc + 4
  cla32 branch_adr (
      .pc(pc),
      .x1(immediate),
      .x2(32'b0),
      .p4(branchpc)
  );
```

```
    cla32 genjalrpc (ra,immediate,32'b0,jalrpc);

    //根据pcsource决定下一个pc的来源
    mux4x32 nextpc(p4,branchpc, jalrpc, branchpc, pcsource[1:0]
    ,npc);
```

3.1.6 lui 指令

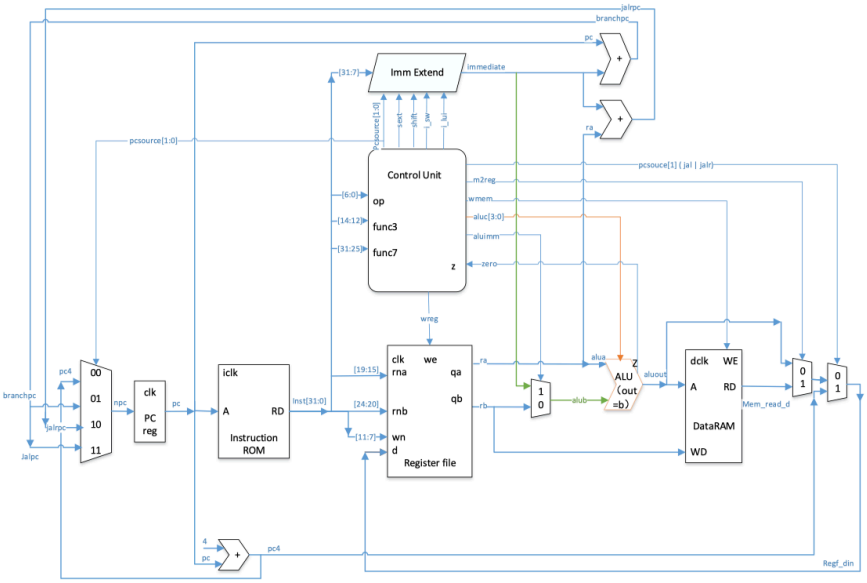

Figure 7: lui 指令的数据通路

lui 指令格式为 lui rd,imm, 它将 20 位 imm 左移 12 位后写入目标寄存器 rd。立即数的移位已在 lab2 的 immext 文件中实现。如图 7 中所示，绿线部分将产生的立即数传入 alu mux，由 aluimm=1 我们设置 alub=imm。lui 的 aluc 是 0001，通过 aluc 的控制，aluout=rb=imm，再通过与 R-type 和 I-type 一样的数据通路写入 rd。

由此看来，lui 的数据通路已经全部实现，所以无需进一步更改前述指令已经完成的数据通路。

3.2 设计过程中的重点问题

问题 1　在进行 IP 例化的时候要注意，无论是 ROM 还是 RAM，都要取消 “Primitives Output Register” 的勾选，否则在模拟仿真的过程中 memout 和 pc 会出现错误对齐的情况，导致异常。

问题 2　data memory 的 data 来源于寄存器的 rb 输出，我们需要在代码中加入”assign data = rb”, 否则 mem 的读写就会出现问题。

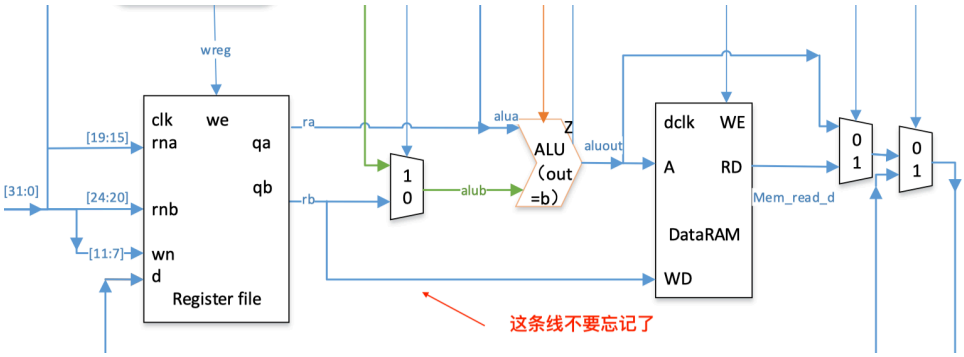

Figure 8: data 的来源

问题 3　在计算 pc 时我们用到了 cla32 模块，模板中的 x2 变量实际上没有参与函数过程。我们在例化该模块时，为了保证程序不出错，在例化时最好统一将 x2 初始化为 32’b0, 或将 cla32 模块中的变量删去。

4 实验结果演示与总结

图 9 为仿真结果波形总图，程序执行到最后在 1515.1ns，停在最后一条死循环跳转指令处，此时寄存器 Reg18 的值为 0001ffff，指令 inst=0000006f，PC 指针 pc=0000007c。

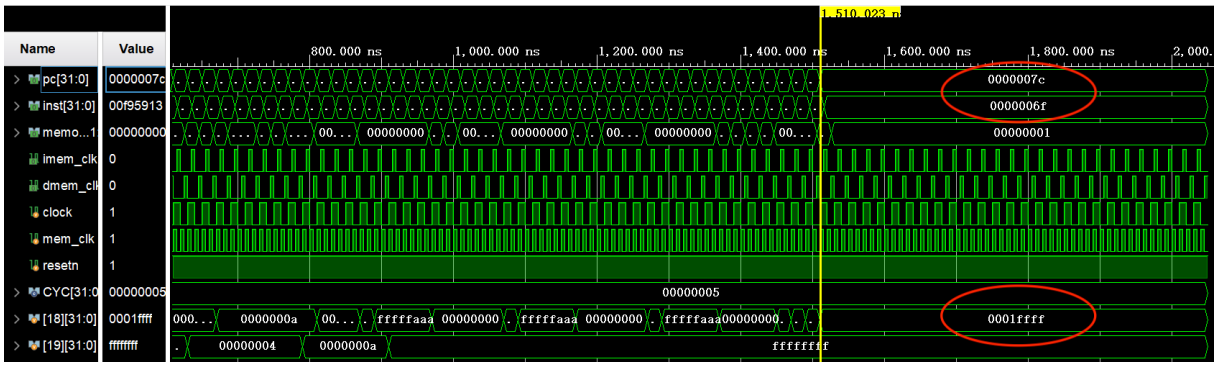

Figure 9: 仿真波形总图

图 10 中标注了程序执行到 730ns 处的情形。这时可以看到寄存器 reg12 的值为 0000000a。此刻执行的指令是 slli x12, x18, 0，reg18 的值被正确赋值给 reg12，reg12 的值改变为 0000000a。说明前面的调用子程序循环计算的一系列指令得到正确执行，且正常退出循环，说明分支指令运行正常，slli 位移指令运行正常。

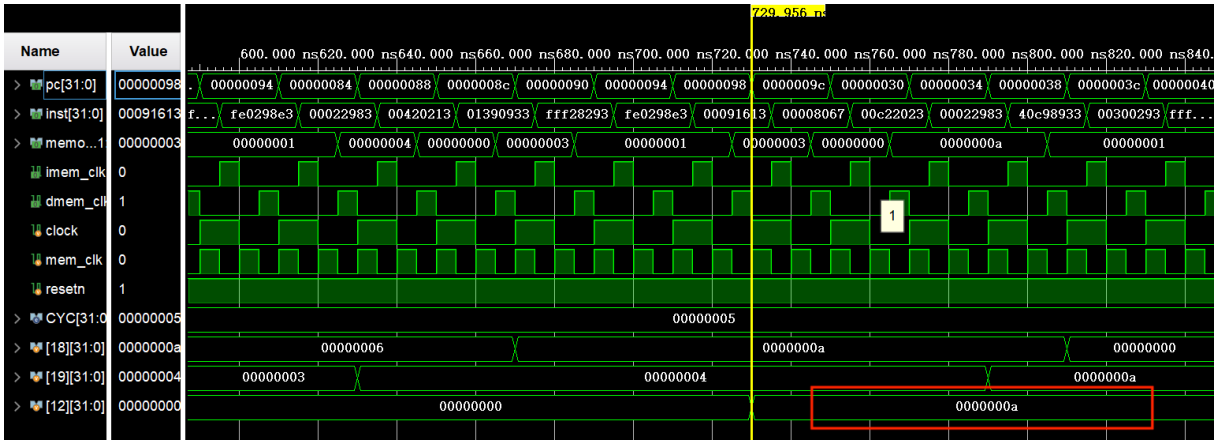

Figure 10: 730ns 处波形图

图 11 中标注了程序执行到 790ns 处的情形。这时可以看到寄存器 reg18 的值为 0000000a。是四个数 1，2，3，4 之和。此刻执行的指令是 lw x19，0(x4)，从存储器中读到的 0000000a 被正确地读入寄存器 reg19 中，reg19 的值改变为 0000000a。说明前面的调用子程序循环计算的一系列指令得到正确执行，计算结果被正确写入数据存储器，并在此时被正确读出来写入寄存器 reg19。读写数据存储器的指令也得到正确执行。

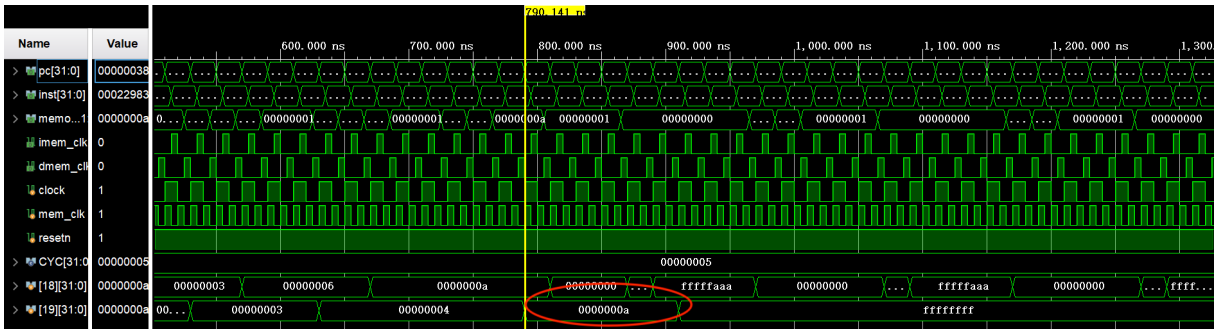

Figure 11: 790ns 处波形图

图 12 中标注了程序执行到 1430ns 处的情形。这时可以看到寄存器 reg18 的值为从 0xffff8000，到 0x80000000，变换回 0xffff8000，最后到 0x0001ffff。这一过程中运行的指令是是 slli x18, x18, 16,slli x18, x18, 16,srai x18, x18, 16,srli x18, x18, 15。

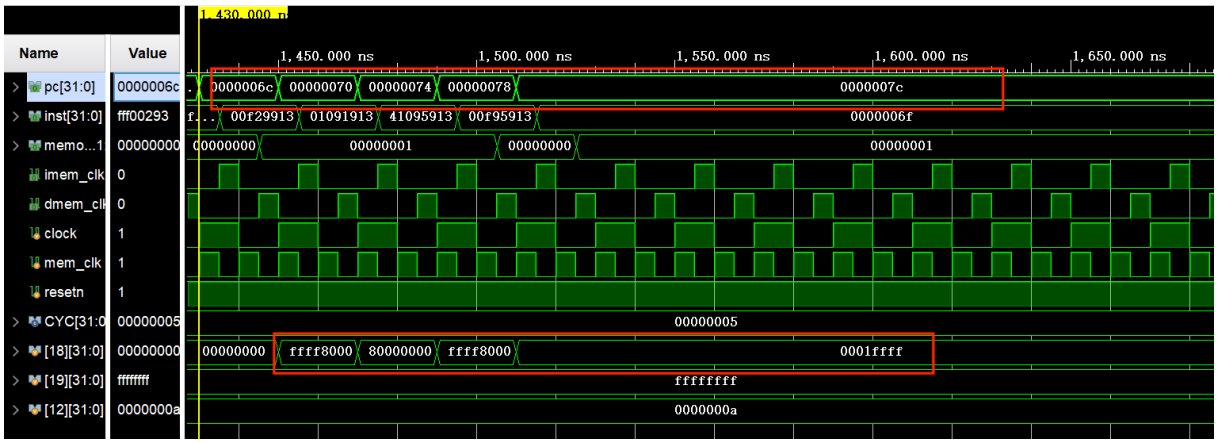

Figure 12: 1430ns 处波形图

与 lab2.1 中指令运行图14进行参照可知，slli，srai，srli 位移指令均工作正常。

68	shift :	shift: addi x5, x0, -1	#X5=0xffffffff
6c		slli x18, x5, 15	#X18=0xffff8000
70		slli x18, x18, 16	#X18=0x80000000
74		srai x18, x18, 16	#X18=0xffff8000
78		srli x18, x18, 15	#X18=0x0001ffff

Figure 13: lab2.1 指令运行图表 (部分)

5 拓展思考部分

5.1 要求

本次 lab3 中完成的数据通路其实使用了许多不必要的 FPGA 资源，考虑尽量减少使用硬件资源，用于产生 branchpc 和 jalrpc 地址的加法器是可以省掉的，同样使用 ALU 来计算目标地址，这样也能省掉四选一多路选择器。最终我们仅使用二选一多路选择器即可完成完整的数据通路。

5.2 数据通路修改

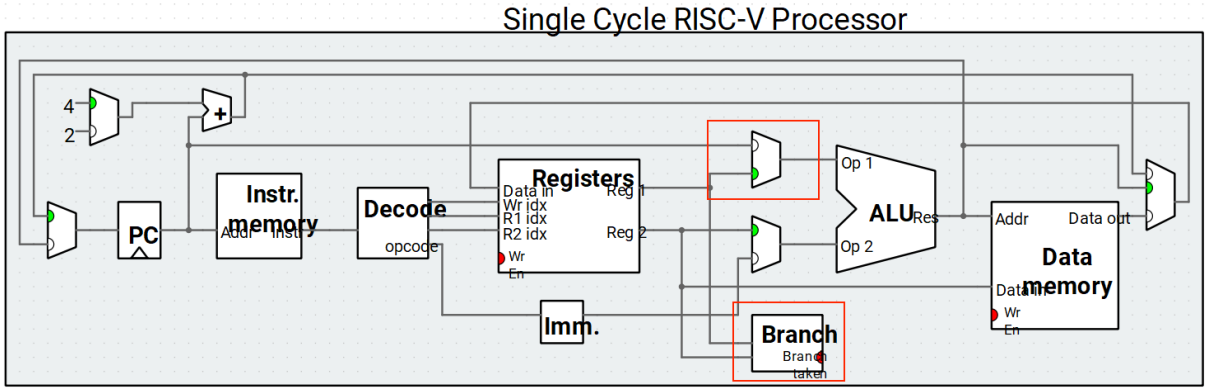

Figure 14: RIPES 上的单周期处理器数据通路

从 RIPES 的单周期处理器示意图中可以看出，与本实验中的数据通路对比，它在 alua 的部分加了一个 mux 让 pc 可以作为 alua 进行输入，同时还增加了一个 Branch Taken 模块用于判断跳转条件是否成立。根据这两处设计，我们可以对原代码作如下修改：

- 在 sc_cpu 文件中增加一个二路选择器 MUX，用于选择 alua 的来源是 pc 还是寄存器的输出 ra。同时还需要增加一个控制信号，记为 aluac，用于控制这个 MUX。这个 aluac 应赋值为 i_beq | i_bne | i_jal
- 值得注意的是，对于 branch 的判断，我们需要用到 ALU 单元的 Zero 输出，这意味着我们无法同时得到 Zero 并对 Branchpc 在 ALU 上进行运算。RIPES 的解决方法是增加一个 Branch Taken 单元来处理这个问题。简化起见，我们可以直接赋值 Zero = ra - rb, 然后让 ALU 单元用于 pc 的计算。
- 去除用于计算 Branchpc 和 Jalpc 的两个加法器，还有四路选择器。