Lab 3 报告

实验目的

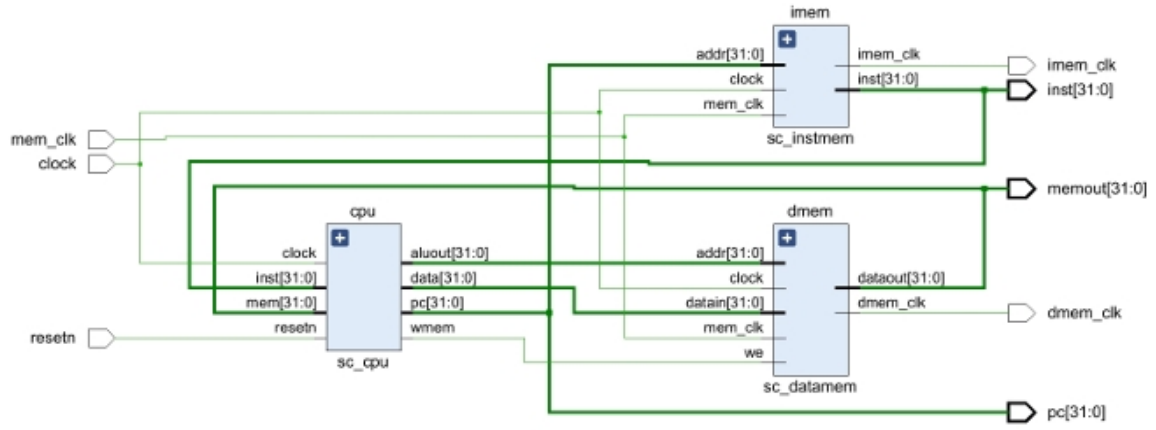
1. 搭建一个单周期CPU模型，通过连接Instruction ROM、寄存器堆、立即数扩展、算术单元、Data RAM等模块，掌握不同类型指令在数据通路中的执行路径。
2. 掌握Vivado仿真方式，学会利用波形图Debug解决问题；验证CPU正常执行IF、ID、EX、MEM、WB五个阶段。

设计思路

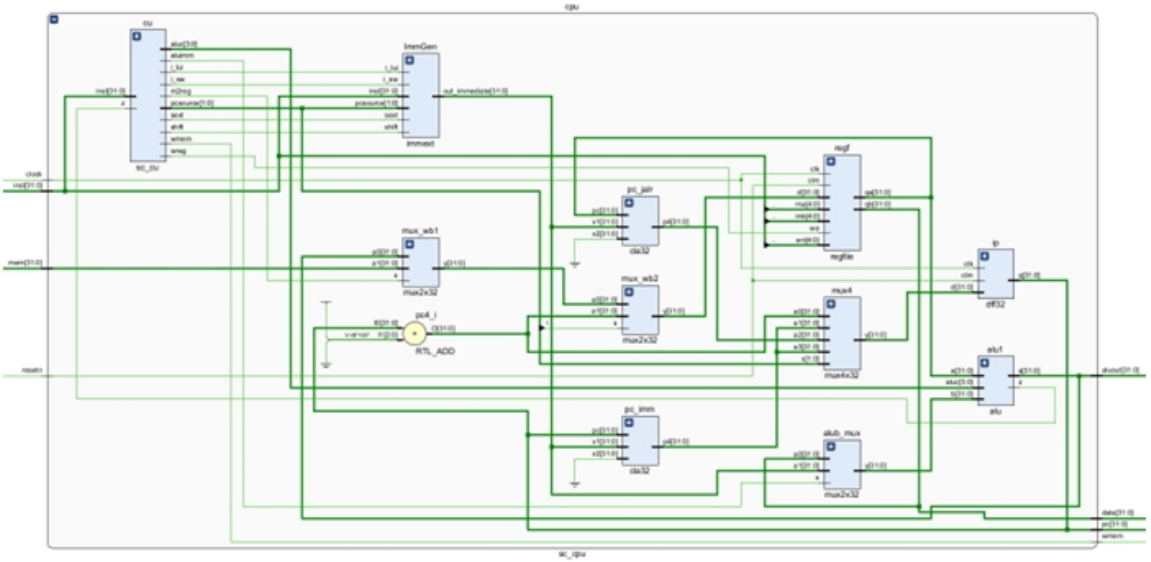
1. 导入之前实验中完成的一系列代码，如控制器、PC、ALU、有符号立即数扩展模块。
2. 实例化指令内存irom和数据内存iram，正确设置相关选项，并导入实验提供的coe文件。
3. 结合单周期 CPU 框架图和不同指令的数据通路，在sc_cpu.v中完成数据通路的连线。
4. 导入模拟模块进行模拟，观察波形图，结合Ripes中的结果，检查CPU是否可以正常运行。如果有问题，通过添加合适的观察项，结合波形图检查代码、连线等问题。

以下是本次实验要求的连线图

整体连线图：



CPU内部连线图：



主要连线代码：

```
// sc_cpu.v （部分）
wire [31 0]   pc4=pc+4;

dff32 ip (npc,clock,resetn,pc);  // pc指针寄存单元，在clock上升沿时将npc写入pc

immext ImmGen(inst,pcsource,sext,i_lui,i_sw,shift,immediate);   // 立即数生成模块

regfile
regf(inst[19 15],inst[24 20],regf_din,inst[11 7],wreg,clock,resetn,ra,rb);  // 寄存器堆，要注意writeback端口的正确实现

sc_cu cu (inst,zero,wmem,wreg,m2reg,aluc,aluimm,pcsource,sext,i_lui,i_sw,shift);

// 控制单元
lu alu1 (ra,alub,aluc,aluout,zero);  // 算术单元，其中a端口直接连接寄存器堆的ra输出

mux2x32 alub_mux (rb,immediate,aluimm,alub);  // alub端口输入多路选择器

cla32 pc_imm (pc,immediate,0,branchpc);  // branchpc加法器：branchpc=pc+immediate

cla32 pc_jalr (ra,immediate,0,jalrpc);  // jalr pc加法器：jalrpc=ra+immediate

mux4x32 mux4 (pc4,branchpc,jalrpc,branchpc,pcsource,npc);  // next pc多路选择器，通过pcsource判断pc4、branchpc、jalrpc和branchpc

mux2x32 mux_wb1 (aluout,mem,m2reg,alu_mem);  // 两个write back多路选择器

mux2x32 mux_wb2 (alu_mem,pc4,pcsource[1],regf_din)
```

实验结果演示

PC+4与取指

```
0:        00000537        lui x10 0x0
4:        00056213        ori x4 x10 0
8:        00100c93        addi x25 x0 1
c:        00200d13        addi x26 x0 2
10:        00300d93        addi x27 x0 3
14:        00400e13        addi x28 x0 4
18:        01922023        sw x25 0 x4
1c:        01a22223        sw x26 4 x4
20:        01b22423        sw x27 8 x4
24:        01c22623        sw x28 12 x4
28:        00400293        addi x5 x0 4
```

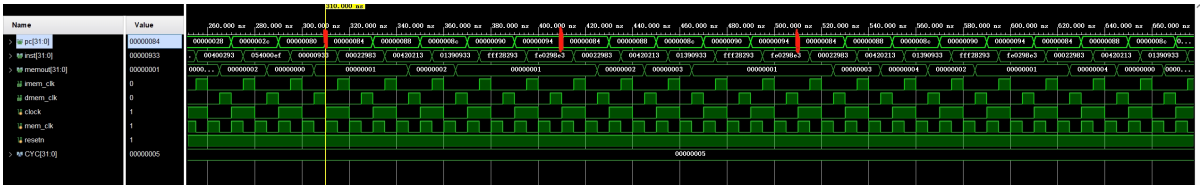
vivado模拟数据图

```
0:      00000537      lui x10 0x0
4:      00056213      ori x4 x10 0
8:      00100c93      addi x25 x0 1
c:      00200d13      addi x26 x0 2
10:      00300d93      addi x27 x0 3
14:      00400e13      addi x28 x0 4
18:      01922023      sw x25 0 x4
1c:      01a22223      sw x26 4 x4
20:      01b22423      sw x27 8 x4
24:      01c22623      sw x28 12 x4
28:      00400293      addi x5 x0 4
```

相应Ripes指令

如图所示，当resetn变为高位，CPU开始运行。PC值每加4，指令都会发生相应的变化，且取值结果与Ripes中的指令完全相同。

指令跳转与循环



vivado中的指令循环

```
00000084 <loop>:
    84:      00022983      lw x19 0 x4
    88:      00420213      addi x4 x4 4
    8c:      01390933      add x18 x18 x19
    90:      fff28293      addi x5 x5 -1
    94:      fe0298e3      bne x5 x0 -16 <loop>
    98:      00091613      slli x12 x18 0
    9c:      00008067      jalr x0 x1 0
```

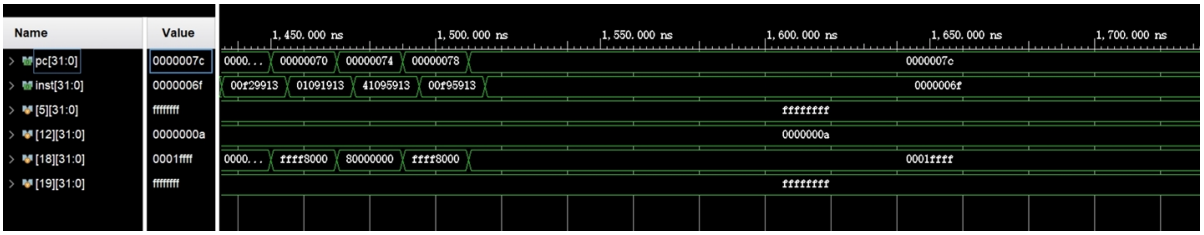
Ripes中的loop

如上图所示，**PC=2c**的指令执行完后进入**PC=84**至**PC=94**的Loop。结合Ripes所示，该loop执行四次，寄存器*x5*从4开始每次循环减少1，减少至0后 `bne x5 x0 -16` 不再成立跳出循环。 同时每次循环中*x18*储存了循环累加 **1+2+3+4** 的值，最终将函数调用结果存于**x12=0000000a**。

loop完成后的jalr跳转

如上图，四次循环完成后进行jalr跳转，PC从**9c**跳至**30**。从存储器中读到的**0000000a**被正确地读入寄存器*x19*中，*x19*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，并且计算结果*x12*能被正确写入数据存储器，对应指令 **sw x12 0 x4** ，并在此时被 **lw x19 0 x4** 正确读出来写入寄存器*x19*。读写数据存储器的指令也得到正确执行。

程序结束



程序结束状态

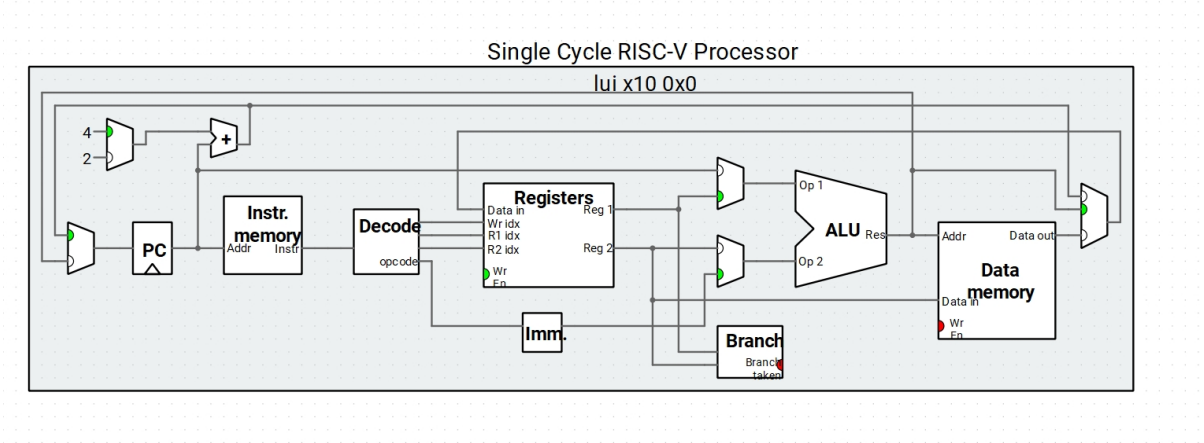
```
00000068 <shift>:
    68:         fff00293        addi x5 x0 -1
    6c:         00f29913        slli x18 x5 15
    70:         01091913        slli x18 x18 16
    74:         41095913        srai x18 x18 16
    78:         00f95913        srli x18 x18 15

0000007c <fi>:
    7c:         0000006f        jal x0 0 <fi>
```

x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0xffffffff
x17	a7	0xffffffff
x18	s2	0x0001ffff
x19	s3	0xffffffff

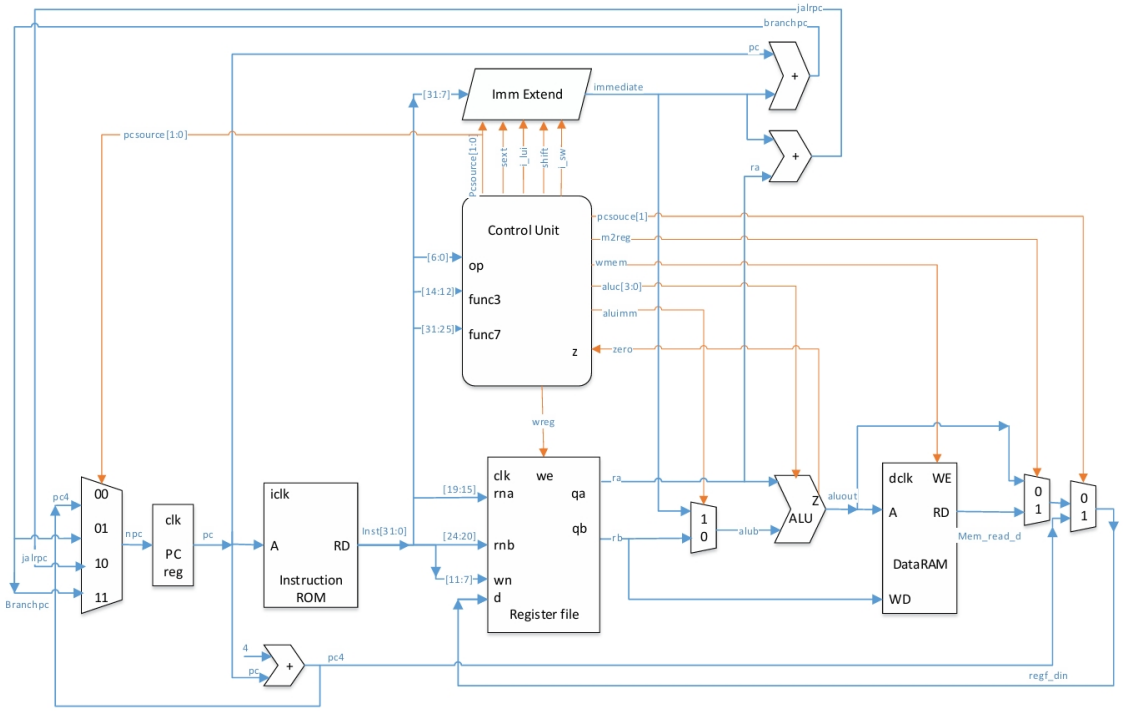
Ripes中对应代码及状态

如上图所示，程序执行到最后，停在最后一条死循环跳转指令 `jal x0 0` ， 此时寄存器 *x18*的值为**0001ffff**，指令`inst=0000006f`， PC指针**pc=0000007c**。结合上图Ripes运行的结果，也说明我们的CPU能正确处理**shift**中的 **slli**、**srai**、**srli** 这些移位指令，我们的CPU运行结果与预期相符。

拓展思考



Ripes框架图



实验实现框架图

由上面两张图可以发现，我们实现的数据通路确实存在一些不必要的FPGA资源，例如我们利用ALU计算PC的目标地址，这样我们就不需要利用加法器来分别计算*branchpc*和*jalrpc*，同时，我们还避免了使用四选一多路选择器导致的FPGA资源额外消耗，从而使得我们的硬件设计更为符合RISC-V的简单原则.