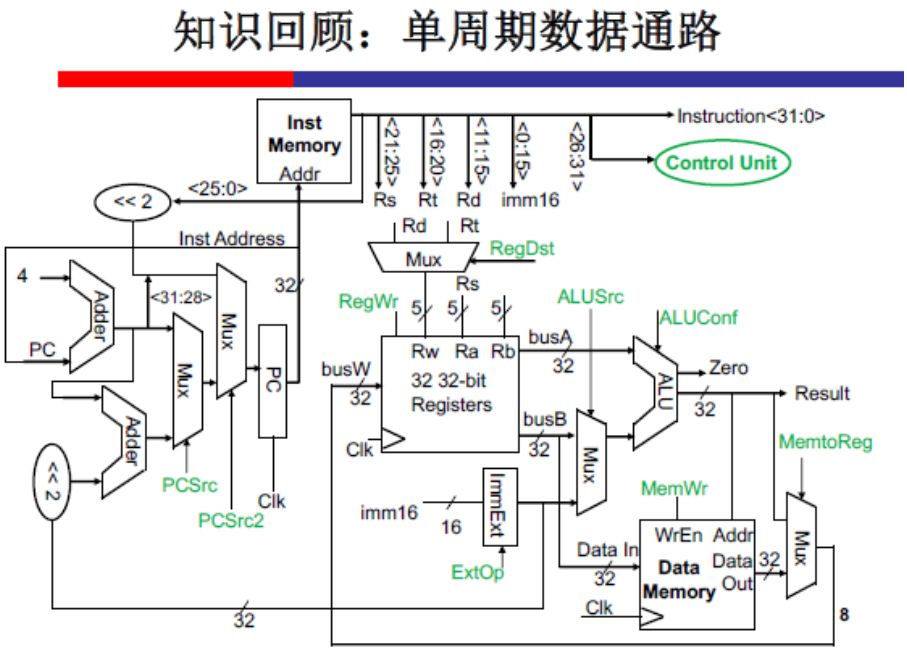
实验报告

一、实验内容

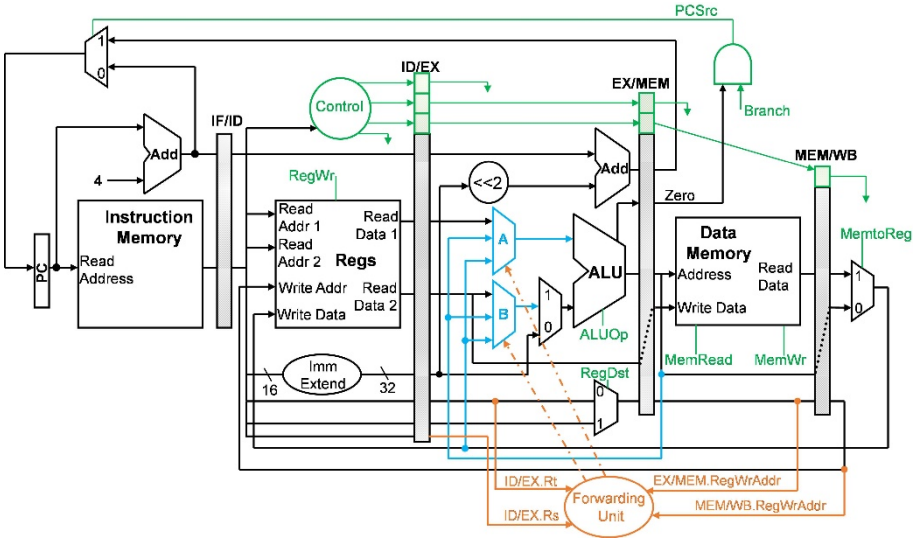
将理论课处理器大作业中设计的单周期 MIPS 处理器改进为流水线结构，并利用此处理器 完成最短路径算法

二、设计思路

（1）MIPS 基础框架，单周期：



多周期：



（2）单周期升级流水线

1.增加流水线寄存器

在单周期处理器的设计中,增加了 4 个流水线寄存器:IF_ID,ID_EX,EX_MEM,MEM_WB。4 个寄存器均为上升沿触发，分别需要保存流水线中的各种中间值和控制信号:

IF_ID:IF 阶段的 PC，取出的指令和时钟、stall 等控制信号。

ID_EX:ID 阶段的 PC，指令的操作码 OpCode，指令解码产生的 Rs、Rd、Rt，位移量 Shamt，R 型指令的 Funct，从寄存器当中读出的数据值，符号扩展单元扩展过的立即数，控制单元产生的所有控制信号（不区分在哪个阶段被使用）。

EX_MEM: EX 阶段的 PC，目标寄存器的地址 RDest，ALU 计算结果，即将写入内存中的数据，跳转目标 BranchTarget，MEM 及其之后阶段使用的控制信号。

MEM_WB:MEM 阶段的 PC，目标寄存器的地址 RDest，内存当中读出的数据，ALU 计算结果，写回时需要的控制和使能。

2.修改分支和跳转逻辑

按照要求，需要支持新的分支指令，因此原有的 Zero 信号需要扩展。依旧在需要分支时使得 Zero 为 1，在 ALU 中增加了所有新的分支指令的判断条件，使得 Zero 信号仍在需要分支时为 1，Control 模块也要做出相应的修改。

由于 ALU 的这个结构，就完成了分支指令在 EX 阶段判断，而跳转指令在 ID 阶段判断的要求。也正是由于分支和跳转发生的阶段不同，设置 flush 信号，在分支指令发生时，冲刷 IF_ID 和 ID_EX 的控制信号;而跳转只需要冲刷 IF_ID 的控制信号。

此时,还应当修改生成 PC 下一个值的逻辑。下一个 PC 的来源主要有:PC+4，跳转目标，分支目标，中断和异常处理程序的地址，以及 stall 信号拉高时的 PC 本身。

3.冒险处理

实现 stall 和 flush，Branch 类指令和 J 型指令都在 ID 阶段完成 PC 更新，拿到错误指令时需要 flush。

Flush：beq 类和 j 类在遇到需要 flush 时，将 flush 从 ID 发到 IF_ID，对 ID_EX 阶段的 flush 配合 stall 进行；

Stall：用于 load_use 和 beq

```
89    assign flush_IFID = (ID_IsJump | ID_Branch)&(stall_IFID == 0);
90    assign stall_IFID = (ID_Branch &&
91              ((  EX_RegWrite && (EX_rd == IF_ID_Instruction[25:21] || EX_rd == IF_ID_Instruction[20:16]) )
92              || (MEM_MemRead && (MEM_rd == IF_ID_Instruction[25:21] || MEM_rd == IF_ID_Instruction[20:16])))
93              || (EX_MemRead && (EX_rd == IF_ID_Instruction[25:21] || EX_rd == IF_ID_Instruction[20:16]))
94              )? 1:0;
```

加入 Fowarding unit

a.转发到 ID: Branch 提前需要的数据，来源:EX/MEM

b.转发到 Ex:EX/MEM，MEM/WB

c.转发到 MEM 只会来源于 MEM/WB

d.beq 前一条为 R 或是 lw 都需要 stall 然后转发

到 ID 阶段的转发与 ID 阶段的判断计算：

```
Forwarding_ID BrForwarding(IF_ID_Instruction[25:21], IF_ID_Instruction[20:16],
                           MEM_rd, MEM_RegWrite,
                           MEM_MemRead, WB_rd, BrForwardingA, BrForwardingB);
assign BrALUData1 = BrForwardingA == 1 ? MEM_ALU_out :
                    BrForwardingA == 2 ? WB_Databus3 : ID_Databus1;
assign BrALUData2 = BrForwardingB == 1 ? MEM_ALU_out :
                    BrForwardingB == 2 ? WB_Databus3 : ID_Databus2;
BranchALU BrAlu(IF_ID_Instruction[31:26], BrALUData1, BrALUData2, ID_Branch, Zero);
```

到 EX 阶段的转发：

```
Forwarding_EX ALUForward(
EX_rs,EX_rt,
MEM_rd,WB_rd,
MEM_RegWrite,WB_RegWrite,MEM_MemRead,
ALUForward1,ALUForward2
);

assign ALU_in1 = EX_ALUSrc1? {27'h00000, EX_Shamt}:
                 (ALUForward1==2)? MEM_ALU_out:
                 (ALUForward1==1)? WB_Databus3: EX_Databus1;
assign ALU_in2 = EX_ALUSrc2? EX_LU_out:
                 (ALUForward2==2)? MEM_ALU_out:
                 (ALUForward2==1)? WB_Databus3: EX_Databus2;
ALU alu1(
    .in1    (ALU_in1),
    .in2    (ALU_in2),
    .ALUCtl (EX_ALUCtrl),
    .Sign   (EX_Sign),
    .out    (EX_ALU_out),
    .zero   (nouse_Zero)
);
```

Load-use 冒险：先 stall 一个周期，再行转发。根据理论课所学知识，stall 信号产生的条件是：EX 阶段执行 load 指令；ID 阶段的 Rs、Rt 源寄存器与 EX 阶段的 load 指令的 Rd 寄存器相同。stall 信号一旦拉高，PC 寄存器的值保持不变，IF_ID 寄存器不变，而后续寄存器控制信号全部置零.

三、汇编代码

1. 采用 bellman-ford 算法计算最短路径，累加得到输出结果
2. 没有除法计算，故使用累减获得带余除法

```
#16进制转10进制存储
#循环计算余数
addi $t2, $t0, 0
addi $t4, $zero,0
addi $t3, $zero, 10
div_loop1:
  sub $t2, $t2, $t3                    #被除数减除数
  addi $t4, $t4, 1                     #统计除法结果
  bgtz $t2, div_loop1                  #结果大于等于0，循环
  beq  $t2, $zero, div_loop1
add $s0, $t2, $t3                      #余数，结果小于0则加上出书
addi $t4,$t4, -1                       #结果
```

3. 0-9 翻译成 BCD 控制信号

0	00111111	0x3f
1	00000110	0x06
2	01011011	0x5b
3	01001111	0x4f
4	01100110	0x66
5	01101101	0x6d
6	01111101	0x7d
7	00000111	0x07
8	01111111	0x7f
9	01101111	0x6f

4. 软件 BCD 显示控制：

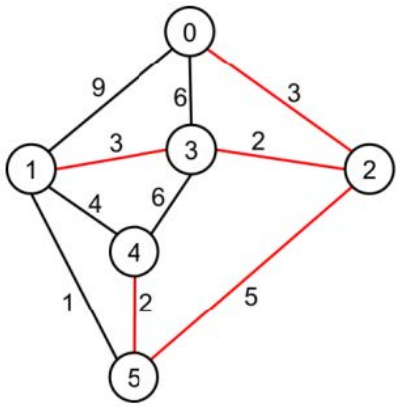
```
print:
addi $t0, $zero, 1              # t0: count time
addi $t8, $zero, 100
print_loop:
    beq $t0, $t8, final_end
    lui $t9, 0x4000
    addi $t9, $t9, 0x0010                  # the address for BCD control
    sw $s4 , 0($t9)
    jal wait_func
    sw $s5 , 0($t9)
    jal wait_func
    sw $s6 , 0($t9)
    jal wait_func
    sw $s7 , 0($t9)
    jal wait_func
    j print_loop

wait_func:
    addi $t1, $zero, 1
    addi $t2, $zero, 10
    wait_loop:
        beq $t1, $t2, end_wait_loop
        addi $t1, $t1, 1
        nop
        j wait_loop
    end_wait_loop:
    jr $ra

final_end:
    nop
```

四、调试

1.采用理论课样例测试



$$G=\begin{pmatrix} 0 & 9 & 3 & 6 & -1 & -1 \\ 9 & 0 & -1 & 3 & 4 & 1 \\ 3 & -1 & 0 & 2 & -1 & 5 \\ 6 & 3 & 2 & 0 & 6 & -1 \\ -1 & 4 & -1 & 6 & 0 & 2 \\ -1 & 1 & 5 & -1 & 2 & 0 \end{pmatrix}$$

答案应为 8 3 5 10 8，即输出 34

2.调试

（1)在只有R-beq或者lw-beq这类时会同时产生IF_ID的flush和stall，此时会产生同为1的冲突。调试时选择beq留在原地，即flush失效

（2）汇编指令与 verilog 设计时要注意单位的不同。lw 指令中的单位是bytes，而 verilog 中的单位一般是 bit，在设计寄存器大小时要注意大小的换算。

（3）ID阶段更新PC时，计算PC+4要用ID_PC不能直接用IF的PC，这样beq 在 IDstall 的时候 PC 也不会跑掉了。实际上这段 PC 更新还是要再考虑考虑，原来的 PC 是直接在 ID 阶段的逻辑里算出来的，问题就是 ID_PC 和现在的PC已经不是一个东西了。要分开讨论用哪个进行更新

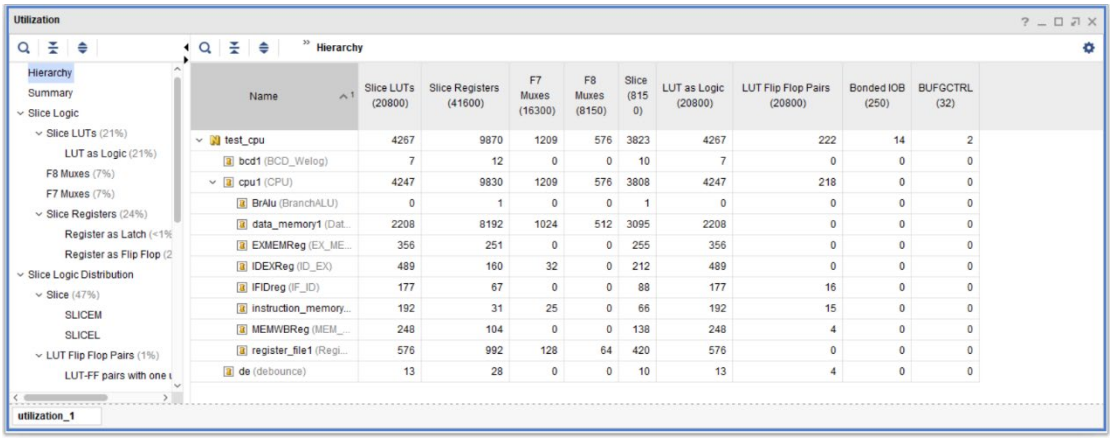
```
182     assign ID_PC_plus_4 = ID_PC + 32'd4;
183     wire [32 -1:0] Jump_target;
184     assign Jump_target = {ID_PC_plus_4[31:28], IF_ID_Instruction[25:0], 2'b00};
185     wire [32 -1:0] Branch_target;
186     assign Branch_target = (ID_Branch & Zero)? ID_PC_plus_4 + {ID_LU_out[29:0], 2'b00}:
187                            (ID_Branch)? ID_PC_plus_4: PC + 32'h4;
```

（4）bgtz 和bltz 要注意不能直接使用>,<号比较，这是无符号比较，要check符号位和是否是0。

```
else if(OpCode == 6'h07) begin  // bgtz
    Zero <= ((data1[31] == 0) & ~(data1 == 0));
end
else if(OpCode == 6'h01) begin  // bltz
    Zero <= (data1[31] == 1 );
end
```

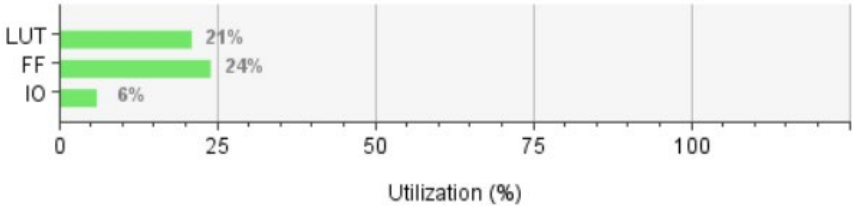
五、资源与时序性能使用

资源使用情况

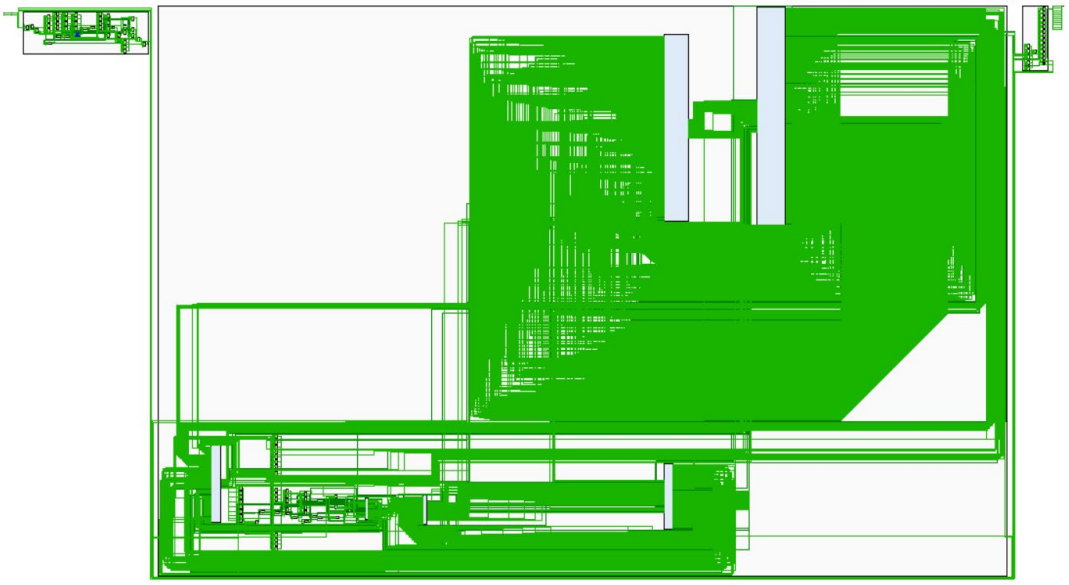


Summary

Resource	Utilization	Available	Utilization %
LUT	4267	20800	20.51
FF	9870	41600	23.73
IO	14	250	5.60

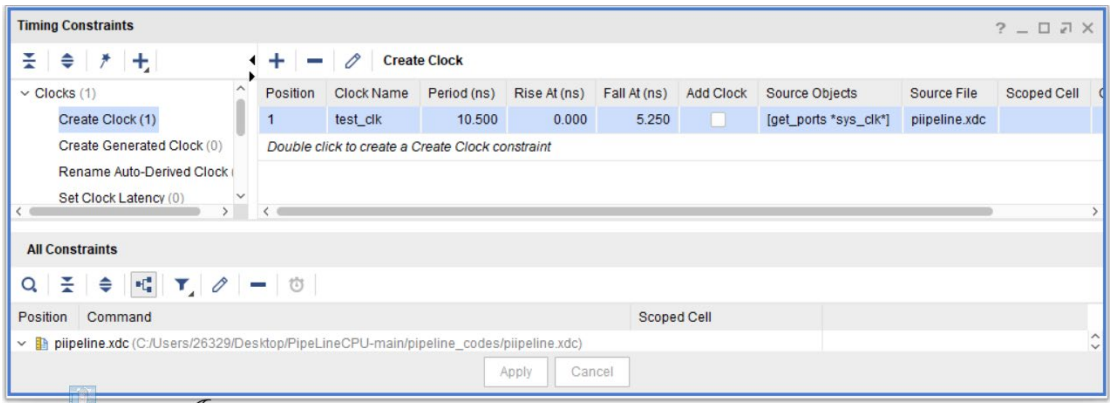


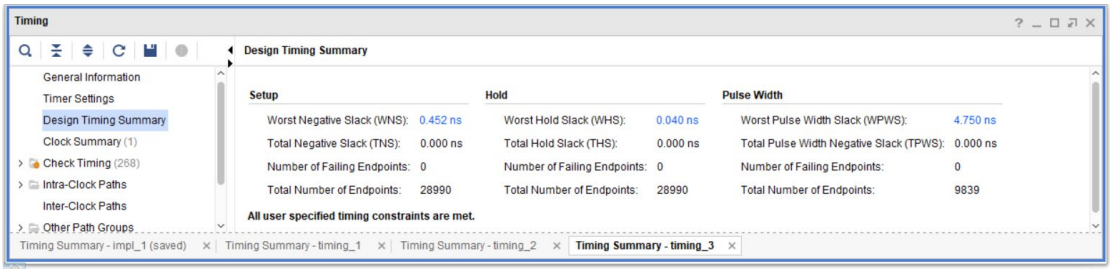
Schematic

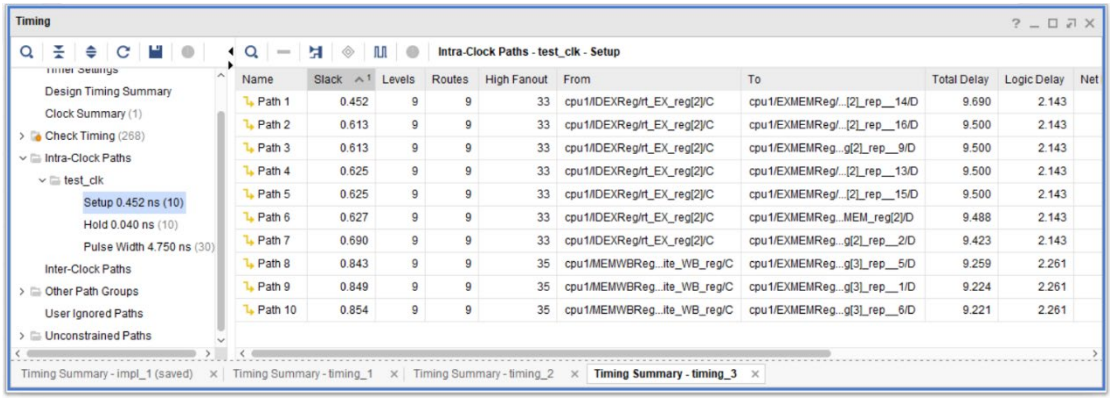


时序情况

设置时钟约束为 10.5ns（测试时初始设置 10ns，此时 WNS 和 TNS 为负数，效果并不好。调试后选择了 10.5ns），实际使用从管脚接来的 100M 信号



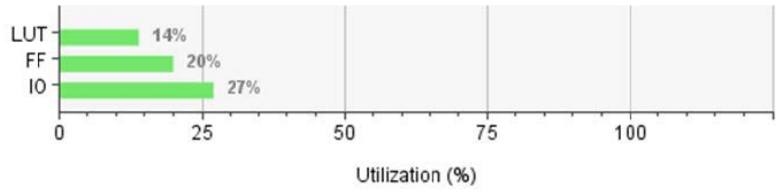


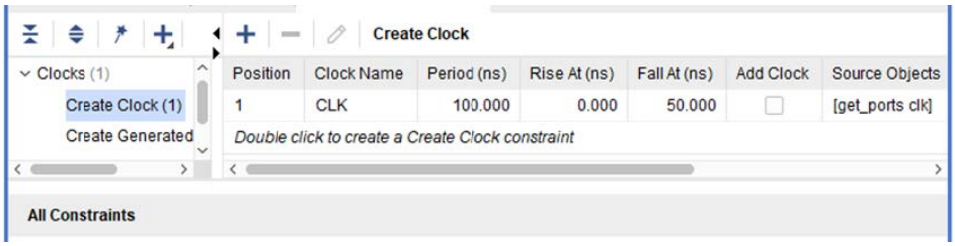


最短周期大约在 9.690ns 左右，最大时钟频率$F = \frac{1}{9.690ns} = 103.2MHz$

与单周期处理器对比

Resource	Utilization	Available	Utilization %
LUT	2928	20800	14.08
FF	8425	41600	20.25
IO	68	250	27.20





最大时钟频率约为$Fm = \frac{1}{(clk-WNS)} = \frac{1}{(100-87.965)} = 99MHz$

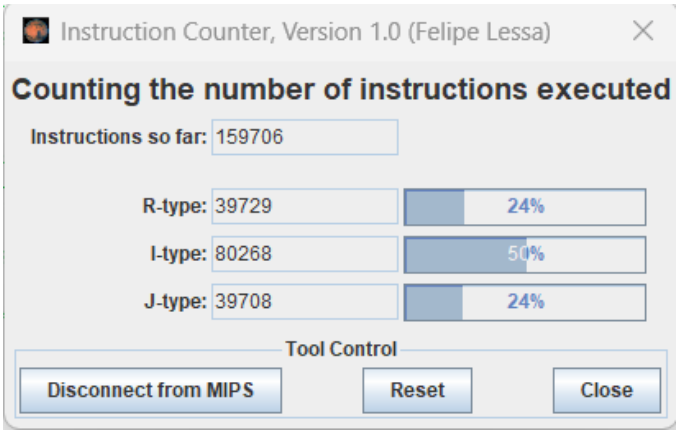
两者对比，可以发现流水线处理器用到了更多的 LUT 和 FF，其中 LUT 增长比较明显。因为虽然流水线中确实有了更多的级间寄存器，但是实际消耗主要是存储器实现，FF 消耗增加较少；LUT 则是由于增加了各种控制信号、转发处理等消耗量增大。

但是优化结果显示频率增加不明显，一方面可能是单周期实现功能有所不同，逻辑更简单，以及理论课测试的并没有很精确。另一方面也有可能是 vivado 综合实现过程中设计的问题，单周期综合实现所需时间远高于 pipeline 的综合实现时间。

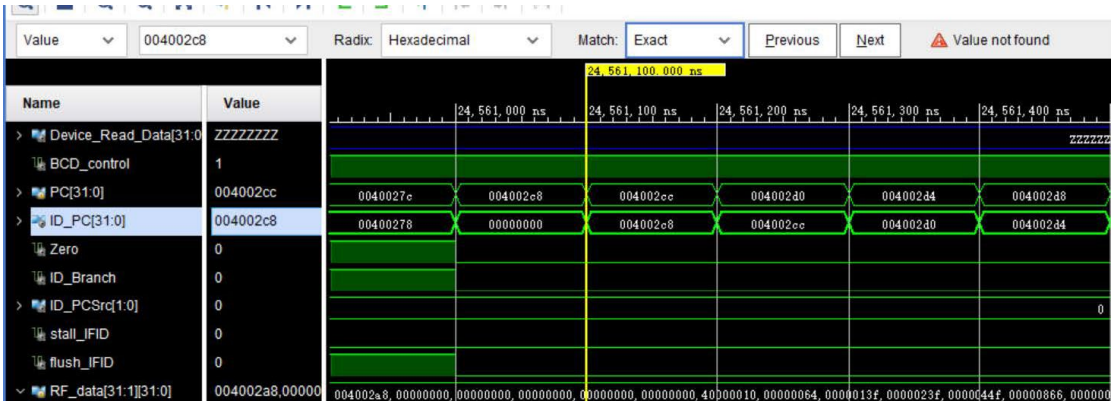
六、仿真结果

仿真结果：16-19 寄存器中存储的是每位上的十进制数字，20-23 存储译码结果，与测试答案 34 符合

CPI



循环执行原汇编程序，得到指令数 159706 个。仿真得到流水线使用时间为 24561400ns，即 245614 个 cycle



$$CPI = \frac{245614}{159706} = 1.5379134519$$

硬件运行：

运行正确。

七、总结

本次实验系统比较复杂，需要的知识全面，工作量较大，设计方案需要从总体角度考虑周全。

在进行流水线的设计时，首先考虑的是添加流水线寄存器。在这一过程中，一点点添加每个流水线寄存器需要保存的数据和控制，中间寄存器的宽度逐渐增大，代码行数也逐渐膨胀。到了最后，反倒要重新整理一遍这些信号，把它们在 CPU 顶层文件当中写清。

编写汇编代码同样也是很困难的一部分，对这一部分的工作量预估不够也导致未能如期完成实验。所幸从头开始复习了一遍汇编之后，写出的代码能够在处理器上运行。这深刻地说明，硬件和软件的设计是密切关联的。

八、文件清单

- pipeline 约束和设计文件
  - ALU.v
  - ALUControl.v

    - BCD_Welog.v
    - BranchALU.v
    - clk_gen.v
    - Control.v
    - CPU.v
    - DataMemory.v
    - debounce.v
    - Device.v
    - EX_MEM.v
    - Forwarding_EX.v
    - Forwarding_ID.v
    - ID_EX.v
    - IF_ID.v
    - InstructionMemory.v
    - MEM_WB.v
    - piipeline.xdc 约束文件
    - RegisterFile.v
    - sim_pip.v 仿真文件
    - test_cpu.v 最终框架设计
- asm
    - bellman_4pipeline.asm 原始的汇编程序
    - bellman_4pipeline_4CPIcalculate.asm 有限循环次
    - machinecode.txt 汇编译码后导出的文件
    - mq.txt 上一个文件提出来的机器指令
    - instruction.txt 机器码指令代码
- 实验报告