数字逻辑与处理器基础实验

流水线大作业报告

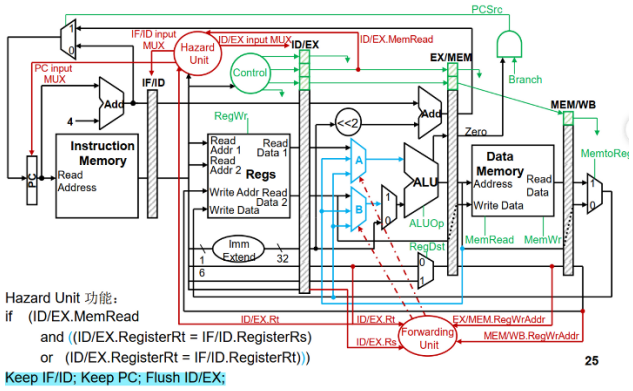
2021012828 卜丞科

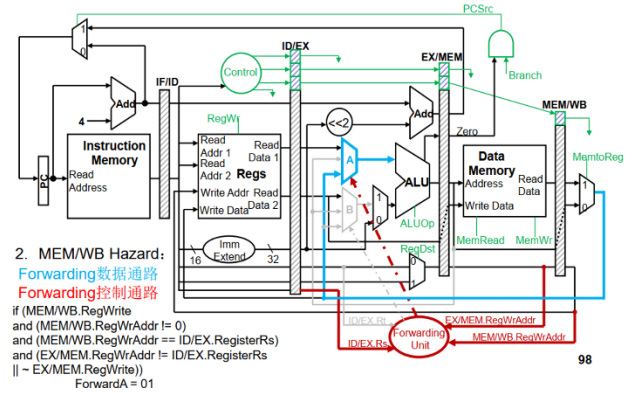
一、 实验目的

（1）熟悉了解现代的流水线处理器的原理，将数逻课程中学到的理论知识应用到工程实际中。

（2）了解单周期处理器和流水线处理器的区别，熟悉从单周期处理器到流水线处理器的转化。

二、 设计方案以及关键代码





流水线的设计图主要参考数逻课程 PPT 中的这两幅图的整体架构，数据通路和冒险控制。后面又参考单周期的外设搭载将 BCD 模块搭载在 CPU 上。

（1）级间寄存器

该模块由多个寄存器组成，输入为时钟信号、上一级的信号和 reset 信号以及 condition 信号，输出为下一级信号。每个时钟周期，当 reset 为 1 的时候，将所有信号初始化，当 reset 为 0 时，当 condition 为 0 时，信号全部置零，当 condition 为 1 的时候，信号正常流通，寄存器写入输入信号的值，当 condition 为 2 的时候，信号 stall 一个周期，寄存器写入自己储存的值。下面以 IF 和 ID 级间的寄存器为例，其余的原理相同，仅是信号不同。

```
module IFID(
clk,reset,instructionin,PCplus4in,condition,instructionout,PCplus4out
    );
    input clk;
    input reset;
    input [31:0]instructionin;
    input [31:0]PCplus4in;
    input [1:0]condition;
    output reg[31:0]instructionout;
    output reg[31:0]PCplus4out;
always@(posedge clk) begin
if(reset==1)begin
instructionout<= 32'h00000000;
PCplus4out<= 32'h00000000;
 end
 else if (condition==0) begin
instructionout<= 32'h00000000;
PCplus4out<= 32'h00000000;
end
else if (condition==1) begin
instructionout<=instructionin;
PCplus4out<=PCplus4in;
end
else if (condition==2) begin
instructionout<=instructionout;
PCplus4out<=PCplus4out;
```

```
end

end
endmodule
```

（2）PC

输入信号为时钟信号、reset 信号、PChazard、brunchyn 信号以及输入 PC 值和输出 PC 值，当分支指令发生跳转的时候，brunchyn 为 1，当分支指令不跳转的时候，brunchyn 为 0.

每个时钟周期内，当 reset 为 1 的时候复位整个系统，当 reset 为 0 的时候，当 PChazard 为 1 且 brunchyn 为 0 时，PC 阻塞一个周期，即当分支指令不跳转的时候，要阻塞一个周期来复位信号，其余情况正常更新 PC 的值。

```
module PC(
    input clk,
    input reset,
    input PChazard,
    input branchyn,
    input [31:0] PCin,
    output reg [31:0] PCnow1
);
  always @(posedge clk) begin
    if (reset == 1)
      PCnow1 <= 32'h00000000;
    else if (PChazard == 1&&branchyn==0)
      PCnow1 <= PCnow1;
    else
     PCnow1 <= PCin;
  end
      endmodule
```

PC 模块的 PCin 这个输入的处理方式和单周期相同，只是需要确定好单周期中的信号在流水线中对应哪一个过程。

```
assign branchyn=(EXBranch & EXzero);
assign Jump_target = {IDPC[31:28], IDinstruction[25:0], 2'b00};
assign Branch_target = branchyn? EXPC + {EXLU_out[29:0], 2'b00}: PCplus4;
```

```
assign PCin = (IDPCSrc == 2'b00)? Branch_target: (IDPCSrc == 2'b01)? Jump_t
arget: Databus1
```

（3）Instruction Memory

该部分几乎完全采用单周期的构造，只是修改了指令储存器的大小，单周期时只截取 PC 的[9:2],这样只支持 2^7=128 条指令，在加入软件译码之后，我的代码有 300 多行，会导致后面的指令无法读取，只会循环执行前面的指令。

（4）Control

文档中要求的 J 型指令在单周期处理器就已经实现，所以只需要加入一些代码来实现 bne,blez,bgtz,bltz,查 MIPS 表可知它们的 Opcode 为 bne:000101 blez:000110 bgtz:000111 bltz:000001

它们对于各个 control 信号的真值表为：

	PCSrc [1:0]	Branch	RegWrite	RegDst [1:0]	MemRead	MemWrite	MemtoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp
bne	0	1	0	X	0	0	X	0	0	1	0
blez	0	1	0	X	0	0	X	0	0	1	0
bgtz	0	1	0	X	0	0	X	0	0	1	0
bltz	0	1	0	X	0	0	X	0	0	1	0

因为之前单周期处理器的 ALUOP 为四位，并且只有三位能用来存不同的 ALUOP（因为最高位用来存 Opcode 的最高一位），所以最多只能存八个 ALUOP，现在的指令过多，无法用四位 ALUOP 完全表达，因此将 ALUOP 扩充为 5 位，然后进行设计，代码如下：

```
assign  PCSrc=
(OpCode == 6'h02 ||  OpCode == 6'h03)? 2'b01:
(OpCode == 6'h00 && Funct==6'h08)? 2'b10:
2'b00;
```

```
assign  Branch=(OpCode == 6'h01||OpCode == 6'h06||OpCode == 6'h07||OpCode =
= 6'h04||OpCode == 6'h05)? 1:0;
assign RegWrite=(OpCode == 6'h2b || OpCode == 6'h01||OpCode == 6'h06||OpCod
e == 6'h07||OpCode == 6'h04||OpCode == 6'h05 || OpCode == 6'h02  || (OpCode
 == 6'h00 && Funct==6'h08))? 0:1;
assign RegDst=(OpCode == 6'h0d ||OpCode == 6'h23 || OpCode == 6'h0f || OpCo
de == 6'h08 || OpCode == 6'h09 || OpCode == 6'h0c || OpCode == 6'h0a || OpC
ode == 6'h0b)? 2'b00:
(OpCode == 6'h03)? 2'b10:
2'b01;
assign  MemRead =(OpCode == 6'h23)?1:0;
assign  MemWrite =(OpCode == 6'h2b )?1:0;
assign  MemtoReg =(OpCode == 6'h23 )?2'b01:
(OpCode == 6'h03)? 2'b10:
2'b00;
assign  ALUSrc1 =((OpCode == 6'h00) &&(Funct==6'h00 || Funct==6'h02 ||Funct
==6'h03)) ?1:0;
assign  ALUSrc2 =(OpCode == 6'h00 ||OpCode == 6'h01||OpCode == 6'h06||OpCod
e == 6'h07||OpCode == 6'h04||OpCode == 6'h05||OpCode == 6'h1c)?0:1;// 加 入
mul 指令
assign  ExtOp=(OpCode == 6'h0c) ?0:1;
assign  LuOp=(OpCode == 6'h0f)?1:0;
assign ALUOp[3:0] =
        (OpCode == 6'h00)? 4'b0010:
        (OpCode == 6'h04)? 4'b0001:
        (OpCode == 6'h0c)? 4'b0100:
        (OpCode == 6'h0a || OpCode == 6'h0b)? 4'b0101:
        (OpCode == 6'h1c)?4'b0111://mul 的 ALUOp
        (OpCode == 6'h05)?4'b0110:
        (OpCode == 6'h0d)?4'b0011:
        (OpCode == 6'h06)?4'b1000:
        (OpCode == 6'h07)?4'b1001:
        (OpCode == 6'h01)?4'b1010:
        4'b0000;

    assign ALUOp[4] = OpCode[0];
```

（5）AIUContorl

和 ALUOP 配合，修改一些控制信号：

```
parameter alubne = 5'b10001;//设置 bne 对应的 ALU 控制信号编码为 5'b11111
parameter alublez = 5'b10010;
parameter alubgtz = 5'b10011;
parameter alubltz = 5'b10100;
```

```
always @(*)
        case (ALUOp[3:0])
            4'b0000: ALUCtl <= aluADD;
            4'b0001: ALUCtl <= aluSUB;
            4'b0100: ALUCtl <= aluAND;
            4'b0101: ALUCtl <= aluSLT;
            4'b0010: ALUCtl <= aluFunct;
            4'b0111: ALUCtl <= alumul;//判断是否是 mul 指令如果是，则将 ALUCtl
赋值为 alumul
            4'b0110: ALUCtl <= alubne;//判断是否是 bne 指令如果是，则将 ALUCtl
赋值为 alumul
            4'b0011: ALUCtl <= aluori;//判断是否是 ori 指令如果是，则将 ALUCtl
赋值为 alumul
            4'b1000: ALUCtl <= alublez;
            4'b1001: ALUCtl <= alubgtz;
            4'b1010: ALUCtl <= alubltz;
            default: ALUCtl <= aluADD;
        endcase
```

（6）ALU

该部分和单周期几乎完全相同，只是增加一些逻辑实现分支指令：

```
assign zero = (out == 0);
always @(*)
        case (ALUCtl)
            5'b00000: out <= in1 & in2;
            5'b00001: out <= in1 | in2;
            5'b00010: out <= in1 + in2;
            5'b00110: out <= in1 - in2;
            5'b00111: out <= {31'h00000000, Sign? lt_signed: (in1 < in2)};
            5'b01100: out <= ~(in1 | in2);
            5'b01101: out <= in1 ^ in2;
            5'b10000: out <= (in2 << in1[4:0]);
            5'b11000: out <= (in2 >> in1[4:0]);
            5'b11001: out <= ({{32{in2[31]}}, in2} >> in1[4:0]);
            5'b11111: out<=in1*in2;//设置 mul 的 ALU
            5'b10001: out<=in1==in2;//设置 bne 的 ALU
            5'b00100: out<=in1|in2;//设置 ori 的 ALU
            5'b10010: out<=in1>0;//设置 blez 的 ALU
            5'b10011: out<=in1<=0;//设置 bgtz 的 ALU
            5'b10100: out<=in1>=0;//设置 bltz 的 ALU
            default: out <= 32'h00000000;
        endcase
```

因为之前判断 beq 是否跳转的时候，当 zero==0 的时候跳转，所以

为了不改变已有结构，之后的四条指令的判断条件都和原条件相反，比如 bne 的判断条件为 in1==in2。

（7）Register File

只需要在单周期的基础上加上先写后读功能即可：

```
 assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: ((Read_reg
ister1 == Write_register&&RegWrite==1) ? Write_data : RF_data[Read_register
1]);
 assign Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000: ((Read_reg
ister2 == Write_register&&RegWrite==1) ? Write_data : RF_data[Read_register
2]);
```

（8）Data Memory

将单周期的内存扩大，然后再写入要储存的邻接矩阵的信息。

但是向内存中写入数据时需要判断一下是要往储存器中写入数据，还是向 BCD 管写入数据。

```
assign Memory_Write         = MEMmemwrite&&MEMALUresult!=32'h40000010;
```

（9）BCD

此模块输入为 sw 要写入的数据和内存地址，检测如果要写入的内存地址为 0x40000010，则写入 BCD 模块，并通过八段数码管向外显示数字。

```
module BCD (
    input reset,
    input clk,
    input [31:0] WBwritadress,
    input [31:0] WBwritdata,
    input BCDWrite,
    input BCDRead,

    output reg [31:0] BCD_Read_data,
    output reg [11:0] digi
);
    always @(posedge clk or posedge reset) begin
        if(reset) begin
```

```
            BCD_Read_data <= 32'h00000000;
        end
        else begin
            if(BCDWrite) begin
                case (WBwritadress)
                    32'h40000010: begin
                        digi <= WBwritdata[11:0];
                    end
                endcase
            end
            else if(BCDRead) begin
                case (WBwritadress)
                    32'h40000010: BCD_Read_data <= {20'h00000, digi};
                    default: BCD_Read_data <= 32'h00000000;
                endcase
            end
        end
    end
endmodule
```

（10） HazardUnit

该模块用于处理各种冒险，处理方式为控制信号使得在下一个时钟周期 PC 不变，或者 IFID 之间的级间寄存器，IDEX 之间的级间寄存器不变或者刷新。具体逻辑为：

当分支指令要执行的时候，PC 保持一个周期不更新，并且 IFID 的级间寄存器清零。当跳转指令要执行的时候，IFID 的级间寄存器清零。当 load-use 冒险发生的时候，PC、IFID 的级间寄存器和 IDEX 的级间寄存器都保持一个周期不更新，等到数据可用了以后再继续执行。

```
module Hazard(
  input [4:0]IDRs,
   input [4:0]IDRt,
   input IDBranch,
   input [1:0]PCSrc,
   input EXBranch,
   input [4:0]EXRt,
```

```
   input EXMemRead,

   output PChazard,
   output [1:0]IFIDhazard,
   output [1:0]IDEXhazard
    );
      assign PChazard = (EXBranch ||IDBranch || (EXMemRead && EXRt == IDRs)
 || (EXMemRead && EXRt == IDRt)) ? 1 : 0;

      assign IFIDhazard = ((EXMemRead && EXRt == IDRs) || (EXMemRead && EXR
t == IDRt)) ? 2'b10 : (PCSrc!=2'b00 || EXBranch || IDBranch) ? 2'b00 : 2'b0
1;

      assign IDEXhazard = ((EXMemRead && EXRt == IDRs) || (EXMemRead && EXR
t == IDRt)) ? 2'b00 : 2'b01;

endmodule
```

（11） Forwarding

该部分主要控制两种情况下的信息传递，第一种是参与 ALU 运算的值在上一个周期内刚刚由 ALU 模块运算得到，现在存在 EX 和 MEM 的级间寄存器里，因此需要判断在 MEM 阶段的这一条指令的两个条件：第一，是否要写入寄存器（还要判断写入是否有效，比如是否要写入零号寄存器）；第二，写入的地址和此时 EX 阶段需要的值的地址是否相同，如果两条都满足，则从 EX 和 MEM 的级间寄存器中将这个值转发到 EX 输入端。第二种信息传递是参与 ALU 运算的值在 MEM 和 WB 的级间寄存器中，因此需要判断三个条件：第一，WB 阶段的这一条指令是否要写入数据；第二，WB 阶段的这一条指令的写入地址是否和 EX 阶段所需要的值的地址相同；第三，要保证此时 EX 阶段需要的数据不来自 EX 和 MEM 级间寄存器，因为这两级之间的数据比 EX 和 MEM 的级间寄存器的值更新生成，这样可以确保连续更新并调用同一个寄存器的值的时候不会发生冲突。

```
module forwarding(
 input [4:0]EXRs,
   input [4:0]EXRt,
   input MEMRegWr,
   input [4:0]MEMRegWriteAddr,
   input WBRegWr,
   input [4:0]WBRegWriteAddr,

   output [1:0]ForwardA,
   output [1:0]ForwardB
    );
     assign ForwardA = (MEMRegWr && (MEMRegWriteAddr != 5'd0) && (MEMRegWri
teAddr == EXRs)) ? 2'b01 :
       (WBRegWr && (WBRegWriteAddr != 5'd0)) && ((WBRegWriteAddr == EXRs) &
& (MEMRegWriteAddr != EXRs || ~MEMRegWr)) ? 2'b10 : 2'b00;

       assign ForwardB = (MEMRegWr && (MEMRegWriteAddr != 5'd0) && (MEMRegW
riteAddr == EXRt)) ? 2'b01 :
       (WBRegWr && (WBRegWriteAddr != 5'd0)) && ((WBRegWriteAddr == EXRt) &
& (MEMRegWriteAddr != EXRt || ~MEMRegWr)) ? 2'b10 : 2'b00;
endmodule
```

在主体程序中需要加入下面两条代码控制信号传递：

```
 assign EXALUAData = (Forward_A == 2'b10) ? WBRegWriteData : (Forward_A ==
2'b01) ?MEMALUresult : EXRead_data1;
 assign EXALUBData = (Forward_B == 2'b10) ? WBRegWriteData : (Forward_B ==
2'b01) ? MEMALUresult : EXRead_data2;
```

（12）　软件译码

代码过于长，在此处只展示核心部分：

首先，用一个循环将最短路径求和：

```
li   $t0, 1
li   $t1, 4200
li   $t3, 0
add_rst:
addi $t1, $t1, 4
lw   $a0, 0($t1)
add  $t3, $t3, $a0          # add rst
addi $t0, $t0, 1
blt  $t0, $s0, add_rst
```

然后将结果一位一位的截取出来，提取十六进制第 n 位的方式为将

数字先向左移 32-4n 位，再向右移 32-4n 位，即去除掉比 n 位高的数字，然后再向右移 4（n-1）位，即将这一位移到最低位，成为一个 16 以内的数字。下面以第三位为例展示代码：

```
sll  $a2, $t0, 20
srl  $a2, $a2, 20
srl  $a2, $a2, 8
```

然后再一一对比这个数是多少，得到结果。赋值的方法为先给这个数对应的数赋值，然后再加上对应显示位置的一个固定常数，最后组成结果。下面是第一位的操作方式：

```
li   $t1, 0 # if $a0 == 0
beq  $a0, $t1, show00
li   $t1, 1 # if $a0 == 1
beq  $a0, $t1, show01
li   $t1, 2 # if $a0 == 2
beq  $a0, $t1, show02
li   $t1, 3 # if $a0 == 3
beq  $a0, $t1, show03
li   $t1, 4 # if $a0 == 4
beq  $a0, $t1, show04
li   $t1, 5 # if $a0 == 5
beq  $a0, $t1, show05
li   $t1, 6 # if $a0 == 6
beq  $a0, $t1, show06
li   $t1, 7 # if $a0 == 7
beq  $a0, $t1, show07
li   $t1, 8 # if $a0 == 8
beq  $a0, $t1, show08
li   $t1, 9 # if $a0 == 9
beq  $a0, $t1, show09
li   $t1, 10 # if $a0 == 10
beq  $a0, $t1, show0a
li   $t1, 11 # if $a0 == 11
beq  $a0, $t1, show0b
li   $t1, 12 # if $a0 == 12
beq  $a0, $t1, show0c
li   $t1, 13 # if $a0 == 13
beq  $a0, $t1, show0d
li   $t1, 14 # if $a0 == 14
```

```
beq  $a0, $t1, show0e
li   $t1, 15 # if $a0 == 15
beq  $a0, $t1, show0f
show00:
li   $a0, 64
j firstnumberend
show01:
li   $a0, 121
j firstnumberend
show02:
li   $a0, 36
j firstnumberend
show03:
li   $a0, 48
j firstnumberend
show04:
li   $a0, 25
j firstnumberend
show05:
li   $a0, 18
j firstnumberend
show06:
li   $a0, 2
j firstnumberend
show07:
li   $a0, 120
j firstnumberend
show08:
li   $a0, 0
j firstnumberend
show09:
li   $a0, 16
j firstnumberend
show0a:
li   $a0, 8
j firstnumberend
show0b:
li   $a0, 3
j firstnumberend
show0c:
li   $a0, 70
j firstnumberend
show0d:
li   $a0, 33
```

```
j firstnumberend
show0e:
li   $a0, 6
j firstnumberend
show0f:
li   $a0, 14
j firstnumberend
firstnumberend:
addi $a0, $a0, 3584
```

最后循环显示结果，此处设计为 1000 个时钟周期更换显示的数字，这样既可以通过视觉暂留让人认为所有灯一直亮着，也可以不至于闪烁过快，导致显示的数字发虚。

```
showloop:
li   $t0, 0
li   $t1, 1000
show1:
sw   $a0, 0($s4)
addi $t0, $t0, 1
blt  $t0, $t1, show1
li   $t0, 0
li   $t1, 1000
show2:
sw   $a1, 0($s4)
addi $t0, $t0, 1
blt  $t0, $t1, show2
li   $t0, 0
li   $t1, 1000
show3:
sw   $a2, 0($s4)
addi $t0, $t0, 1
blt  $t0, $t1, show3
li   $t0, 0
li   $t1, 1000
show4:
sw   $a3, 0($s4)
addi $t0, $t0, 1
blt  $t0, $t1, show4
j showloop
```

三、 文件清单

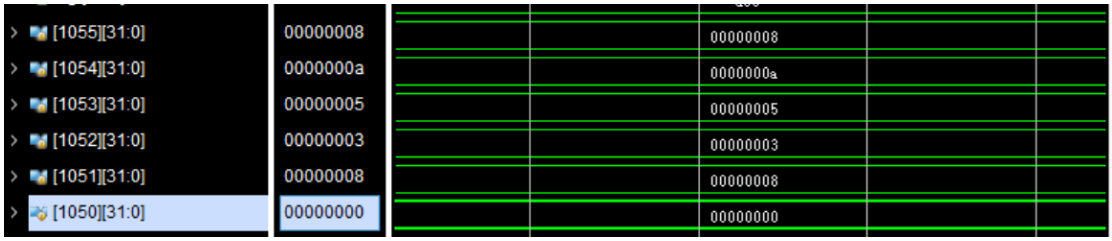
文件名	功能
IFID.v	级间寄存器
IDEX.v	级间寄存器
EXMEM.v	级间寄存器
MEMWB.v	级间寄存器
PC.v	控制指令的 PC
Control.v	生成控制信号
ALUControl.v	生成 ALU 的控制信号
ALU.v	进行 ALU 运算
Hazard.v	控制冒险
forwarding.v	控制信号通路
clock_10M	分频器，使得板子显示清晰
DataMemory.v	数据储存器
Device_BCD.v	控制八段数码管显示
InstructionMemory	指令寄存器
pipelinecpu.v	流水线主体
pipeline_CPU_top.v	仿真文件
RegisterFile.v	寄存器堆
test_cpu.v	仿真文件
pipeline_CPU.xdc	管脚约束文件
Decode.asm	软件译码
sssp_bellman.asm	Bellman 算法和结果显示的 MIPS 代码

四、 仿真结果以及分析

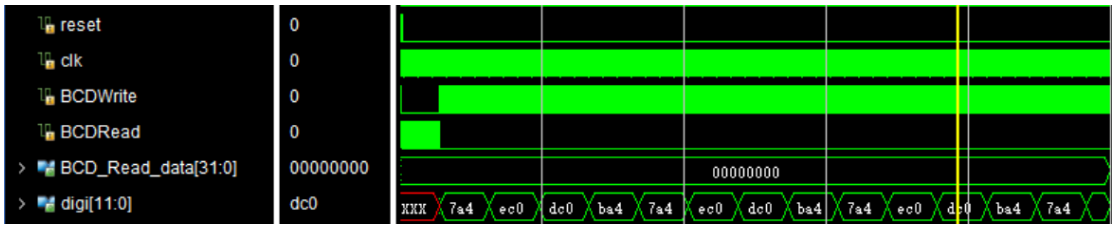
进行行为级仿真测试，因为信号过于多，仿真时间过长，只展示关键性的有代表的结果：

Bellman 算法的结果：

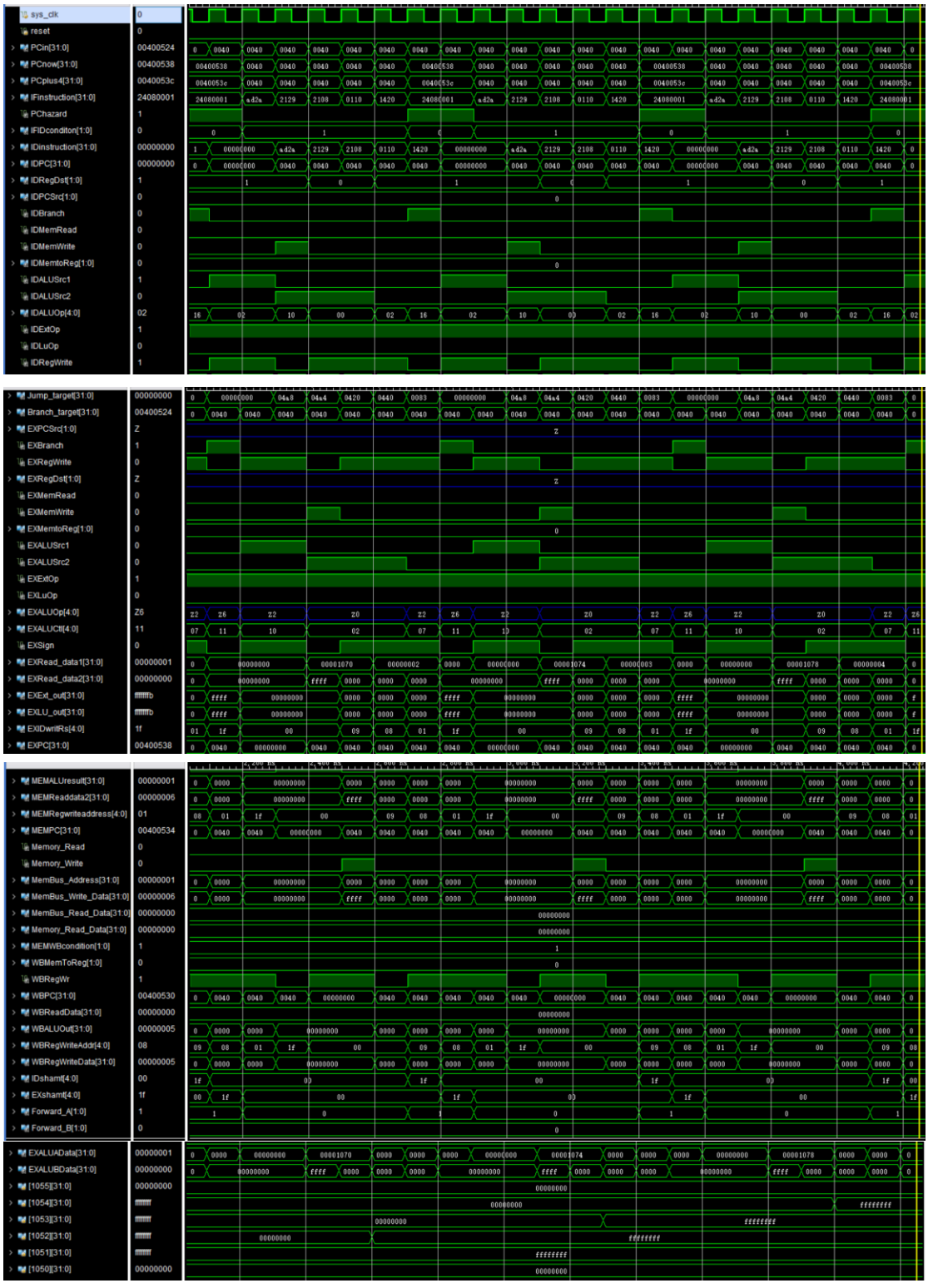
在写 MIPS 代码的时候，我将原来储存 dist 的内存地址起始位置设置为 4200，也就是第 1050 个字节的位置，可见得到的结果是正确的。



BCD 模块：



可以观察到 BCD 模块对外输出的 digi 的值在 7a4,ec0,dc0,ba4 之间循环变化，其中后面两位的 a4 表示 2，c0 表示 0，因此这四个数字分别对应八段数码管显示为：最低位显示 2，最高位显示 0，第 2 高位显示 0，第 2 低位显示 2，不断循环。符合结果显示的要求。

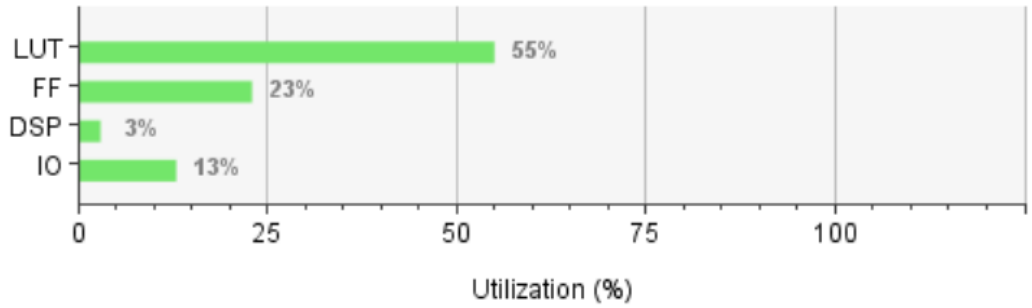


上图为初始化最短距离时的部分信号，可以观察到流水线 CPU 的各个信号工作正确无误。

五、 综合情况

资源分析报告：

Resource	Utilization	Available	Utilization %
LUT	11534	20800	55.45
FF	9775	41600	23.50
DSP	3	90	3.33
IO	14	106	13.21



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
pipelinecpu	11534	9775	1233	415	4420	11534	6878	3	14	1
ALU1 (ALU)	15	0	0	0	4	15	0	3	0	0
BCD1 (BCD)	0	12	0	0	4	0	0	0	0	0
data_memory1 (DataM...	2304	8192	1024	320	3560	2304	5	0	0	0
EXMEM1 (EXMEM)	7614	215	0	0	3168	7614	0	0	0	0
IDEX1 (IDEX)	521	161	37	12	214	521	0	0	0	0
IFID1 (IFID)	194	67	0	0	96	194	0	0	0	0
MEMWB1 (MEMWB)	131	104	0	0	88	131	0	0	0	0
PC1 (PC)	205	32	44	19	80	205	0	0	0	0
register_file1 (Register...	576	992	128	64	473	576	0	0	0	0

分析可得，CPU 占用了 11534 个 LUT、9775 个寄存器。其中 LUT 相比起单周期增加了很多，可见流水线的逻辑复杂程度远大于单周期处理器，其中寄存器数量比单周期多了以前多个，应该是级间寄存器需要的寄存器较多。

管脚约束中对于时钟的定义：

```
create_clock -period 100.000 -name clk -waveform {0.000 50.000} [get_ports clk]
```

静态时序分析：

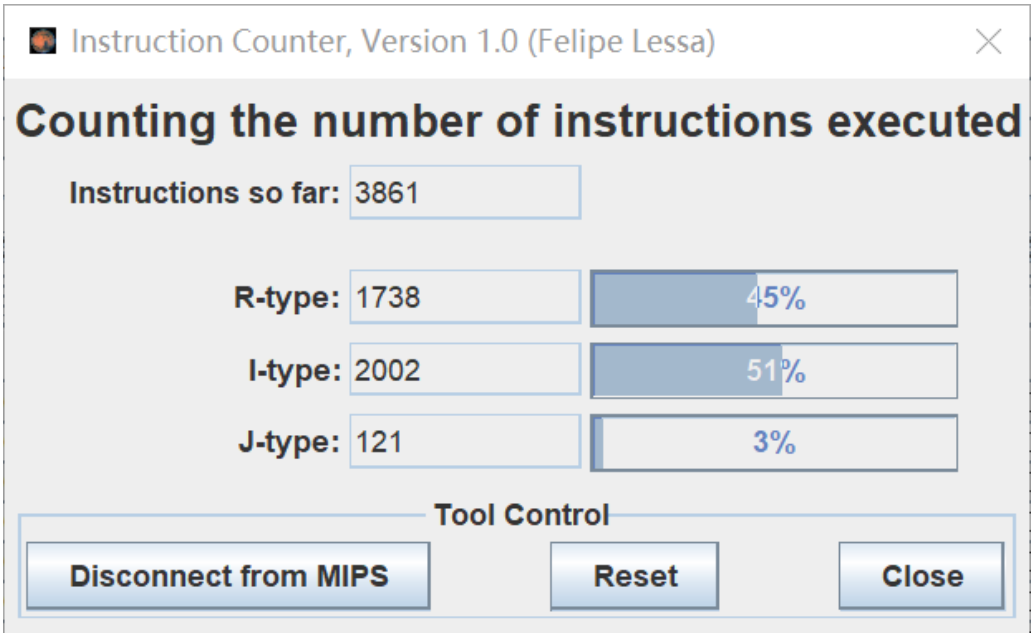
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	71.654 ns	Worst Hold Slack (WHS):	0.013 ns	Worst Pulse Width Slack (WPWS):	49.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	19459	Total Number of Endpoints:	19459	Total Number of Endpoints:	9776

All user specified timing constraints are met.

从时序分析中可以看出，最坏情况下的建立时间是 71.654ns，保持时间是 0.013ns。时钟约束设置的周期是 100ns，那么 CPU 可能达到的最高时钟频率为$\frac{1}{(100-71.654)ns} = 35.3\text{MHz}$ 相比起之前设计的单周期处理器的 32.5MHz 有所提高，符合多周期处理器将一个流程拆成多个过程而加快每个过程的处理速度，加快系统时钟的理论。

计算 CPI：

通过 MARS 可得，bellman 算法执行完的指令数为 N=3861 条指令：



系统从 100ns 时 reset，执行完全部指令为 552.1 us 时，所以总共执行了 552us，其中时钟周期为 100ns，所以总共执行了 5520 个时钟周期，C=5520，可得$CPI = \frac{C}{N} = 1.430$

分析结果 CPI 有些大，理想运转的流水线的 CPI 应该略大于 1，分析汇编代码后，我认为可能是因为 bellman 算法中的数据关联非常紧密，并且很多数据都存在内存中，在执行的时候很有可能发生 load-use hazard，并且因为该算法要执行很多个循环，一旦发生 hazard 会一直重复出现。

六、 硬件调试情况

因为在验收之前调试地比较着急没有照照片，验收之后就归还了板子，这一部分只能通过语言描述了。

我在仿真 debug 中虽然颇受周折，但是在仿真成功之后，上硬件调试比较顺利。第一次上板子调试没有任何显示，但是在我加入了分频模块以后就成功显示出了结果，但是此时屏上的数字 0 颜色比较

淡，并且数字 2 不应该亮的笔画有隐隐的发光，我认为这可能是因为数字变化太快，内部的寄存器发生了逻辑冒险，使得数字在转化的过程中有所影响，我就将软件显示的部分的每个数的显示周期数从 100 改为了 1000，之后再上板子调试数字显示就非常清晰，成功完成了任务。

七、 思想体会

对我来说，这次实验的难度极大。直接将数逻课程中的理论运用到实际设计中非常有挑战性，我一开始看着只有几行的任务要求感到非常的掣肘，不知道从何开始，感觉完全不知道应该从何开始设计。

我对照着数逻课件开始尝试列出信号通路，进行一个模块一个模块的设计，并且和同学们进行了很多讨论，发现大家对于同一个功能的实现千奇百怪，比如有人为了能提前进行分支判断并且无需扩充 ALUOP 而单独加入了 comp 模块，但是我为了能更多地复用单周期的代码依然将这一部分放在了 ALU 模块中。经过了两天的设计和交流我终于大致得到了流水线的构架，我又花了一天将其写出来，本以为即将完成，但是，这只是噩梦的开始。

我的流水线完全不能正常工作，我花费了很多时间在 debug 上，每次都是一条指令一条指令地检查它的运行，调试出了非常多的 bug，但是依然不能正常运行。我花费了好几周零零碎碎地不断 debug，感觉任务量无穷无尽，完全不知道什么时候能成功。下面列出几个让我受尽折磨的 bug：

第一，我一开始将先写后读实现成下面这样：

```
assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: ((Read_regi
ster1 == Write_register) ? Write_data : RF_data[Read_register1]);
```

似乎很正确，但是我忽略了触发先写后读需要保证这个周期有数据要往寄存器堆里写，判断条件应该加入 RegWrite==1，这导致了我的所有寄存器值的读取有概率不能正常进行。有概率出错使得这个 bug 非常恶心，因为它在少量代码测试的时候都是正确的，但是一旦大量执行代码就会出现莫名其妙的错误。它折磨了我很久很久。

第二，我最开始的 load-use 只考虑到了 ALU 模块用的数据可能来自于通路，但是我忽略了 load 阶段也会需要来自通路的数据，在我发现我的 load 指令有时候不能正确写入以后，我修改了 EX 和 MEM 级间寄存器的数据来源，解决了这个 bug。

第三，我发现我的 j 型指令不能正确跳转，我一直以为是指令的实现出错了，但是我检查了很久都没发现问题，后来我才意识到这是因为我的指令寄存器只截取了指令的[9:2]位，所以我后面的指令根本就访问不到，所以才无法跳转。

像这样的 bug 还有很多，都让我很痛苦。这次实验给我留下最深刻的影响就是磨刀不误砍柴工，多花一点时间在设计上能减少数倍 debug 的时间。在 debug 时，不能一口吃成胖子，要先写一些测试性的代码测试每一类指令能否正常工作，然后测试各类指令之间的兼容性，可以大大减少 debug 时间。