流水线MIPS处理器设计

无08 李煜彤 2020010841

目录

[实验目的 2](#_Toc109745833)

[预备工作 3](#_Toc109745834)

[指令格式表 3](#_Toc109745835)

[控制信号 4](#_Toc109745836)

[冒险处理 6](#_Toc109745837)

[结构冒险 6](#_Toc109745838)

[数据冒险 6](#_Toc109745839)

[控制冒险 9](#_Toc109745840)

[JumpRegister冒险 11](#_Toc109745841)

[数据通路 12](#_Toc109745842)

[寄存器寄存清单 13](#_Toc109745843)

[CPU代码编写 14](#_Toc109745844)

[模块设计 14](#_Toc109745845)

[ForwardingUnit：转发单元 15](#_Toc109745846)

[HazardUnit：load-use冒险单元 16](#_Toc109745847)

[BranchUnit: 分支指令处理单元 17](#_Toc109745848)

[PC：PC寄存器及其选择 18](#_Toc109745849)

[连线（顶层设计） 19](#_Toc109745850)

[CPU代码调试 20](#_Toc109745851)

[指令测试 20](#_Toc109745852)

[冒险测试 29](#_Toc109745853)

[字符串查找的实现 31](#_Toc109745854)

[仿真实现 31](#_Toc109745855)

[代码编写 31](#_Toc109745856)

[将较复杂代码写入vivado的指令存储器 33](#_Toc109745857)

[仿真结果 33](#_Toc109745858)

[添加外设 34](#_Toc109745859)

[添加外设后的仿真结果 37](#_Toc109745860)

[硬件调试 38](#_Toc109745861)

[性能分析 39](#_Toc109745862)

[时序性能 39](#_Toc109745863)

[逻辑资源 40](#_Toc109745864)

[文件清单 41](#_Toc109745865)

[心得体会 42](#_Toc109745866)

# 实验目的

运用春季学期课程《数字逻辑与处理器基础》和《数字逻辑与处理器基础实验》所学知识，在此前完成的单、多周期处理器的基础上，将其改进为流水线结构，在其上完成字符串搜索算法，并与单、多周期处理器进行性能比较。

# 预备工作

## 指令格式表

注：

空指令：nop（0x00000000，即sll $0 $0 0）

自己补充的指令包括：slti，bne，ori

## 控制信号

Control:

说明：

PCSrc：PC Source，取指令过程控制。00-顺序执行，PC = PC + 4；01-表示其为分支指令；10-对于j和jal，会跳到target；11-对于jr和jalr，会跳到$rs。

RegWrite：是否写入寄存器。jal和jalr需要，因为要将某个寄存器写入当前PC位置。

RegDst：写入的寄存器。00-对于R型指令和jalr，需写入Rd；01-对于I型指令，要写入Rt；10-对于jal需写入$31。若RegWrite为0，则RegDst为x。

MemRead：是否读内存。只有lw需要。

MemWrite：是否写入内存。只有sw需要。

MemtoReg[1:0]：判断写回寄存器的来源。00-将ALU计算的结果写回寄存器；01-将内存的结果写回寄存器，只有MemRead为1时，MemtoReg才有可能是01；10-将PC+4写入寄存器。若RegWrite为0，则MemtoReg为x。（注：为什么lui也是alu的结果呢？因为lui是先进行了扩展后与$0相加）

ALUSrc1：ALU第一个操作数的来源。0-rs寄存器；1-shamt，因此只有sll、srl和sra的ALUSrc1是1。对于跳转指令，由于无需经过ALU，因此为x。

ALUSrc2：ALU第二个操作数的来源。0-rt寄存器；1-立即数。

ExtOp：立即数扩展方式。0-0扩展，只有andi是0扩展；1-符号扩展。对于非I型指令，为x。Lui由于不需要扩展后的高位，因此也是x。

LuiOp：判断是否是lui指令。对于非立即数的指令，LuOp可以为x。

BranchOp:判断分支指令操作。若PCSrc不为1，则为x。

ALUControl:



控制信号的最后使用阶段：

IF: 生成

ID: ExtOp, LuiOp

EX: PCSrc, RegDst, ALUSrc1, ALUSrc2, BranchOp, ALUOp, Sign

MEM: MemRead, MemWrite,

WB: RegWrite, MemtoReg

## 冒险处理

### 结构冒险

现有的电路图已经解决了结构冒险，包括增加硬件资源、调整步骤执行周期等。

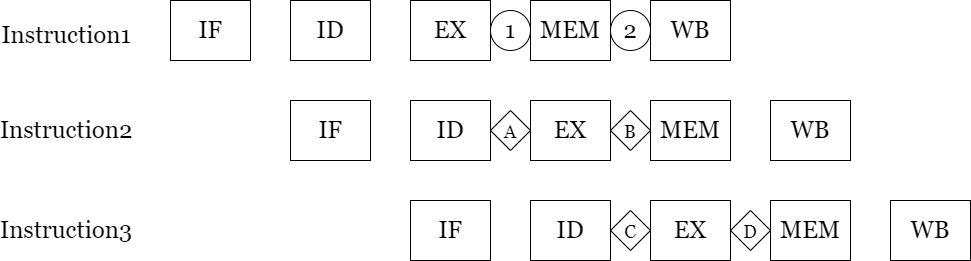
### 数据冒险

1. 寄存器堆的访问

由于有可能对寄存器堆进行读和写的操作，因此在硬件上希望实现先写后读。对于RegisterFile，由于其没有读使能，因此输出数据随时更新，满足了先写后读的需求。

1. Read after write data hazards

下面对于寄存器数据的前后关联进行讨论，包括计算指令与load指令。

假设Instruction1产生了后续可能需要的数据，如下图所示：

可能的数据来源：1——计算指令；2——lw指令

可能需要数据处：A/C——计算指令;B/D——sw指令

若再往后的指令需要用到Instruction1产生的数据，则最早也会在其WB阶段结束后需要，因此不存在数据冒险。

接下来对这2\*4种情况进行讨论：

|  |  |  |
| --- | --- | --- |
|  | 1 | 2 |
| A | EX/MEM->ID/EX | load-use hazard |
| B | 与2-B相同 | MEM/WB->EX/MEM |
| C | 与2-C相同 | MEM/WB->ID/EX |
| D | 不需要转发 | 不需要转发 |

通过上表分析可知：通过转发可以解决五种数据冒险。其中1-B的转发可以通过2-B实现，1-C的转发可以通过2-C实现。下面对这三种情况进行分析（在实际过程中，Forwarding Unit生成的是控制信号）：

注：RegWriteAddr是经过RegDst选择后的信号

1-A:

判断条件：

EX/MEM.RegWrite（指令1需要写入寄存器）

EX/MEM.RegWriteAddr（指令1不写入$0）

EX/MEM.RegWriteAddr == ID/EX.rs（需要进行转发）

转发结果：

ALUSrc1 = EX/MEM.out

判断条件：

EX/MEM.RegWrite（指令1需要写入寄存器）

EX/MEM.RegWriteAddr（指令1不写入$0）

EX/MEM.RegWriteAddr == ID/EX.rt（需要进行转发）

转发结果：

ALUSrc2 = EX/MEM.out

2-B:

判断条件：

MEM/WB.RegWrite（指令1需要写入寄存器）

MEM/WB.RegWriteAddr（指令1不写入$0）

MEM/WB.RegWriteAddr == EX/MEM.rt（需要进行转发）

转发结果：

MemWriteData = RegWriteData

2-C：

判断条件：

MEM/WB.RegWrite（指令1需要写入寄存器）

MEM/WB.RegWriteAddr（指令1不写入$0）

MEM/WB.RegWriteAddr == ID/EX.rs（需要进行转发）

EX/MEM.RegWriteAddr != ID/EX.rs || !EX/MEM.RegWrite（保证数据最新）

转发结果：

ALUSrc1 == MEM/WB.ALUout

判断条件：

MEM/WB.RegWrite（指令1需要写入寄存器）

MEM/WB.RegWriteAddr（指令1不写入$0）

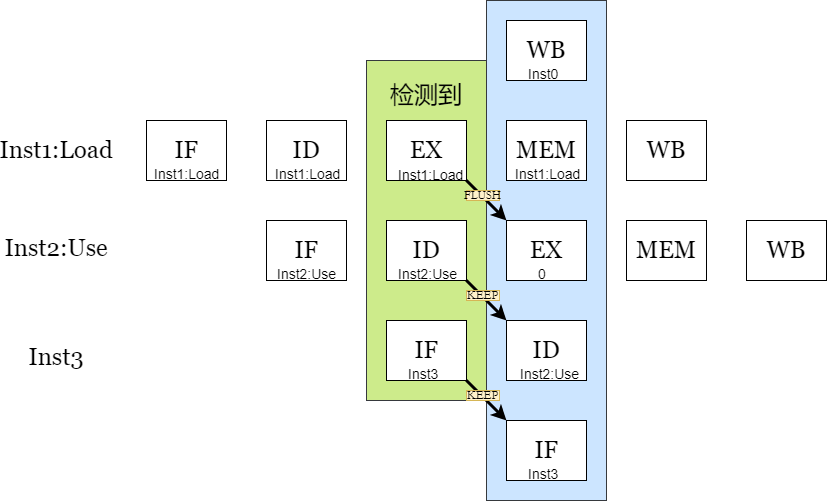
MEM/WB.RegWriteAddr == ID/EX.rt（需要进行转发）

EX/MEM.RegWriteAddr != ID/EX.rt || !EX/MEM.RegWrite（保证数据最新）

转发结果：

ALUSrc2 == MEM/WB.ALUout

1. load-use hazard

当上一条指令为load、下一条指令需要use时，需要通过Hazard单元进行stall控制。如下图所示：

其中图中纵向的方框表示同一时间执行的周期。

若Inst1为Load，Inst2为Use，则当Inst2译码后（绿色方块），可以开始判断是否存在load-use冒险。

若检测到load-use冒险，则对于下一阶段（蓝色方块）的MEM可以继续执行Load指令，EX则需要进行flush，ID、IF则需要进行keep，这样可以实现一个stall。后续可以继续进行冒险检测和forwarding。

EX的FLUSH实现：在ID/EX寄存器前放置多路选择器，选择正常的上一路信号还是0，控制信号由Hazard Unit生成。

ID的KEEP实现：在IF/ID寄存器前放置多路选择器，选择正常的上一路信号还是自己本身的值，控制信号由Hazard Unit生成。

IF的KEEP实现：在PC寄存器前放置多路选择器，选择正常的上一路信号还是自己本身的值，控制信号由Hazard Unit生成。

Hazard Unit的实现：

判断条件：

ID/EX.MemRead（判断为Load指令）

ID/EX.rt==rs || ID/EX.rt == rt（考察下一指令是否需要use）

实现结果：

KEEP PC

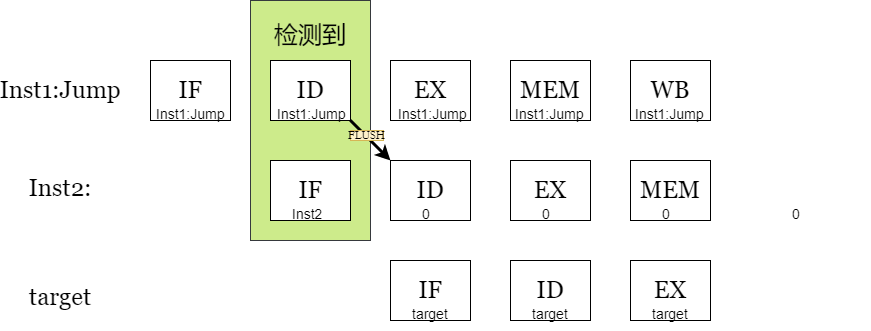
KEEP IF/ID

FLUSH ID/EX

### 控制冒险

1. 跳转指令

跳转指令包括j、jal、jr、jalr。

当出现跳转指令时，过程如下：

其中图中纵向的方框表示同一时间执行的周期。

若第一条指令为跳转指令，则在其ID阶段（绿色方框）可以被检测到，且PCSrc将对PC的值作出指示。同时应将ID进行flush，flush方法同load-use hazard。

判断条件：

当出现跳转指令时，也即PCSrc[1]==1。因为在我设计的控制信号中，PCSrc承担了标记跳转指令和分支指令的任务。

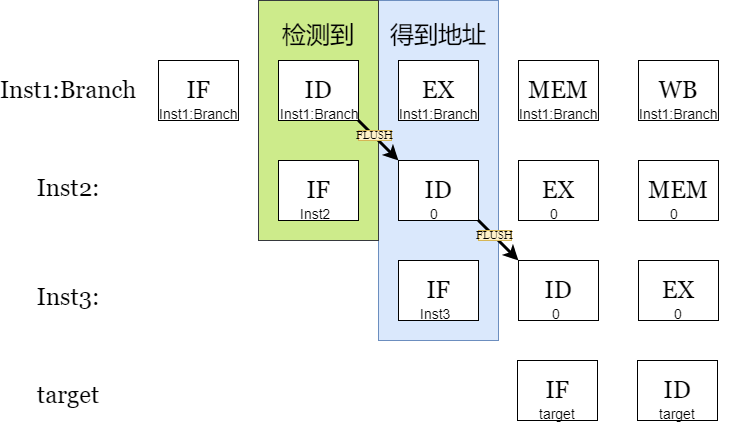


实现结果：

FLUSH IF/ID

1. 分支指令

跳转指令包括beq、bne、blez、bgtz、bltz。

当出现跳转指令时，过程如下：

其中图中纵向的方框表示同一时间执行的周期。

若第一条指令为分支指令，则在其ID阶段（绿色方框）可以被检测到，在其EX阶段（蓝色方框）可以判断出跳转的地址。那么将有以下两件工作需要做：

①在其ID、EX阶段对IF/ID进行flush

若PCSrc为01，则说明现处于绿色方框阶段，需要对ID进行flush。结合上面对于跳转指令的分析，可知：若Inst1为分支或者跳转指令，也即PCSrc!=0时需要对IF/ID进行flush。

若ID/EX.PCSrc为01，则说明现处于蓝色方框阶段，依然需要对ID进行flush。

综上所述，IF/ID的flush条件为：PCSrc || ID/EX.PCSrc == 01

②在EX阶段给出下一条指令的地址，并传递给PC

由于其处理相对复杂，将其封装为Branch Unit来进行处理。大致如下：

输入：ID/EX.BranchOp，ALUout，ID/EX.PCplus4，BranchAddr

输出：BranchTarget

Branch Unit可以通过BranchOp和ALUout来决定下一条指令，即BranchTarget。

### JumpRegister冒险

以上是根据课上所学而得出的结果。然而可能有更多种类的冒险需要进一步斟酌，例如jr和jalr。二者需要在ID阶段拿到rs的值，但很有可能这时候rs的值还没有得到。即：若jr/jalr的上上条指令给出rs的值，则需要EX/MEM->IF/ID的转发；若jr/jalr的上条指令给出rs的值，则需要ID/EX->IF/ID的转发。

判断条件：(优先级更高)

PCSrc == 3

ID/EX.RegWrite == 1

ID/EX.RegWriteAddr != 0

ID/EX.RegWriteAddr == rs

转发结果：

JumpRegisterTarget = ALUout

判断条件：

PCSrc == 3

EX/MEM.RegWrite == 1

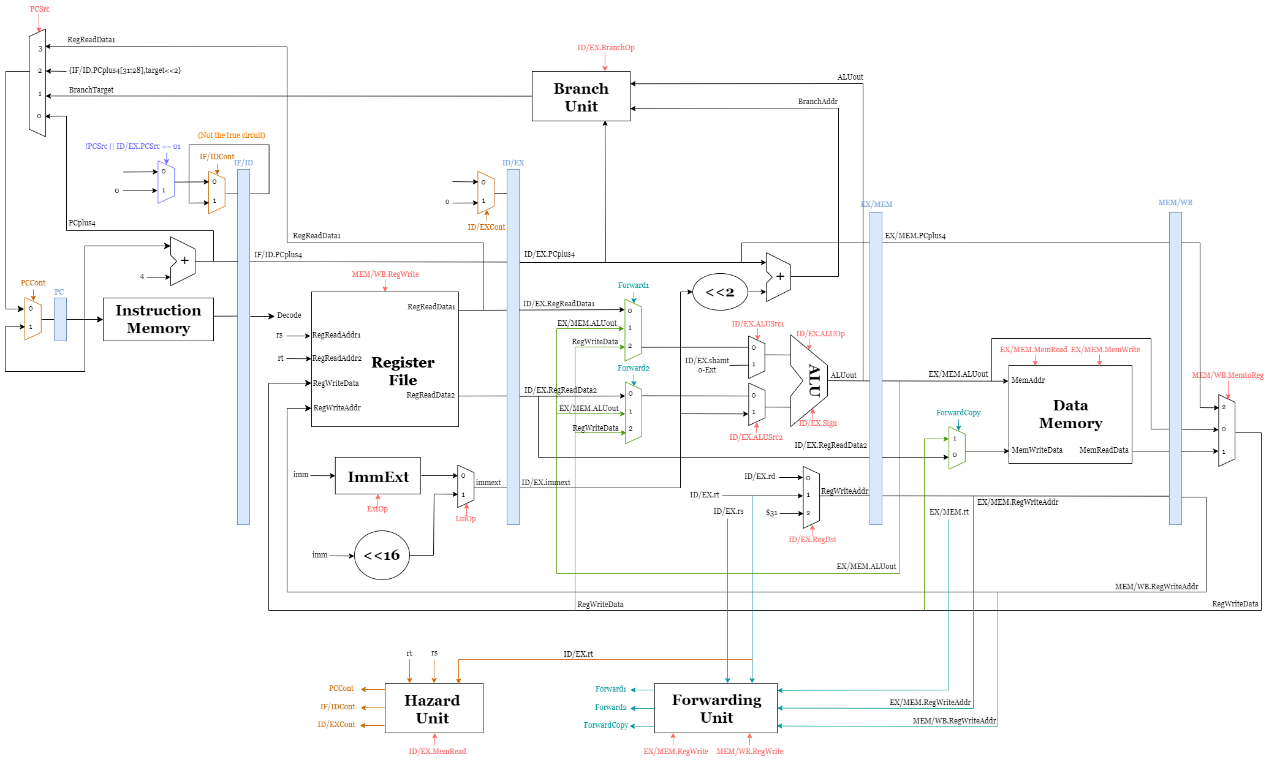
EX/MEM.RegWriteAddr != 0

EX/MEM.RegWriteAddr == rs

转发结果：

JumpRegisterTarget = EX/MEM.ALUout

## 数据通路

 综合上述分析，按照本人理解，可以得到（呕心沥血画出的）数据通路图如下：

余以为大部分地方都是比较严谨而合理的，唯独PC的选择之处难以用电路图完整描述，但用verilog代码会相对容易一些。故做如下说明：

* PC默认为PC+4。特殊情况则特殊判断（e.g.Jump，必须在下一个周期才能判断出其为跳转指令）。而在上图中，PC+4却要经过ID阶段才出现的PCSrc来进行选择，对于顺序执行的指令是不合理的。
* IF/ID寄存器中表示KEEP的回环型信号表示其包含的全部信号。
* 如有需要多级传递的信号，例如rs，则在其第一次出现的阶段，如ID，则以其本名出现；传递后，其名称为上一级间寄存器名称.本名，如ID/EX.rs。

## 寄存器寄存清单

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 寄存器 | IF/ID | ID/EX | EX/MEM | MEM/WB |
| 控制信号 | RegWrite  MemtoReg  MemRead  MemWrite  PCSrc  RegDst  ALUSrc1  ALUSrc2  BranchOp  ALUOp  Sign  ExtOp  LuiOp | RegWrite  MemtoReg  MemRead  MemWrite  PCSrc  RegDst  ALUSrc1  ALUSrc2  BranchOp  ALUOp  Sign | RegWrite  MemtoReg  MemRead  MemWrite | RegWrite  MemtoReg |
| 指令信息 | Instruction | rt  rs  rd  shamt | rt |  |
| 中间数据 | PCplus4 | PCplus4  RegReadData1  RegReadData2  immext | PCplus4  RegReadData2  ALUout  RegWriteAddr | PCplus4  ALUout  RegWriteAddr  MemReadData |

注：IF/ID不存储控制信号，存Instruction

# CPU代码编写

## 模块设计

根据上述前期准备工作，做以下模块设计：

Control：生成控制信号

ALUControl：生成ALU的控制信号

ALU：算术逻辑单元

DataMemory：内存

InstructionMemory：指令内存

RegisterFile：寄存器堆

ForwardingUnit：转发单元

HazardUnit：load-save冒险单元

BranchUnit: 处理分支指令单元

PC：PC寄存器及其选择

IF\_ID：IF/ID寄存器

ID\_EX：ID/EX寄存器

EX\_MEM：EX/MEM寄存器

MEM\_WB：MEM/WB寄存器

下面对关键模块（个性化色彩较重）进行说明。

### ForwardingUnit：转发单元

FowardingUnit，即转发单元，用于判断出需要转发的情况，并生成相应的控制信号。可以解决数据冒险中需要转发的部分。

输入：

input wire EX/MEM.RegWrite

input wire MEM/WB.RegWriteAddr

input wire [4:0] ID/EX.rs

input wire [4:0] ID/EX.rt

input wire [4:0] EX/MEM.rt

input wire [4:0] EX/MEM.RegWriteAddr

input wire [4:0] MEM/WB.RegWriteAddr

输出：

output reg [1:0] Forward1

output reg [1:0] Forward2

output reg ForwardCopy

其中三个输出信号分别负责控制ALU的两个输入数据和写入内存的数据。

在进行转发判断时，使用if-else if-else来判断，因而可以首先判断EX/MEM到ID/EX的转发，保证了其优先级，使得在都需要转发的情况下，使数据保证最新。判断过程如下：

  if(EX/MEM.RegWrite && EX/MEM.RegWriteAddr && EX/MEM.RegWriteAddr == ID/EX.rs) Forward1 <= 2'd1;

        else if(MEM/WB.RegWrite && MEM/WB.RegWriteAddr && MEM/WB.RegWriteAddr == ID/EX.rs) Forward1 <= 2'd2;

        else Forward1 <= 2'd0;

        if(EX/MEM.RegWrite && EX/MEM.RegWriteAddr && EX/MEM.RegWriteAddr == ID/EX.rt) Forward2 <= 2'd1;

        else if (MEM/WB.RegWrite && MEM/WB.RegWriteAddr && MEM/WB.RegWriteAddr == ID/EX.rt) Forward2 <= 2'd2;

        else Forward2 <= 2'd0;

        if(MEM/WB.RegWrite && MEM/WB.RegWriteAddr && MEM/WB.RegWriteAddr == EX/MEM.rt) ForwardCopy <= 1'd1;

        else ForwardCopy <= 1'd0;

### HazardUnit：load-use冒险单元

HazardUnit，即load-use冒险单元（实际上应该负责控制所有的FLUSH，KEEP等等，但其它的另有处理）。它可以在load-use冒险出现时，生成相关的控制信号，使相应寄存器发生FLUSH或KEEP，以达到stall的效果。

输入：

input wire ID/EX.MemRead

input wire [4:0] rt

input wire [4:0] rs

input wire [4:0] ID/EX.rt

输出：

output reg PCCont

output reg IF/IDCont

output reg ID/EXCont

其中三个输出分别控制PC、IF/ID、ID/EX的情况。它们不能完全判断，需和PCSrc等信号共同决定其行为。

判断过程：

if(ID/EX.MemRead && (ID/EX.rt==rs || ID/EX.rt == rt)) begin

        PCCont <= 1'b1;

        IF/IDCont <= 1'b1;

        ID/EXCont <= 1'b1;

    end

    else begin

        PCCont <= 1'b0;

        IF/IDCont <= 1'b0;

        ID/EXCont <= 1'b0;

    end

### BranchUnit: 分支指令处理单元

BranchUnit，即分支指令处理单元，可以在branch指令进行到EX阶段的时候给出下一指令的跳转地址，当然有可能为PC+4，有可能为跳转的地址。

输入：

input wire [2:0] ID/EX.BranchOp

input wire [31:0] ALUout

input wire [31:0] BranchAddr

input wire [31:0] ID/EX.PCplus4

输出：

output reg [31:0] BranchTarget

其中ID/EX.BranchOp表示该分支指令是哪一条，配合ALUout可以判断是否要进行分支跳转，在BranchAddr和ID/EX.PCplus4中选择出正确的一项，即BanchTarget并传递给PC。

判断过程：

case(ID/EX.BranchOp)

            3'd0: begin

                // beq

                if(!ALUout) BranchTarget <= BranchAddr;

                else BranchTarget <= ID/EX.PCplus4;

            end

            3'd1: begin

                // bne

                if(ALUout) BranchTarget <= BranchAddr;

                else BranchTarget <= ID/EX.PCplus4;

            end

            3'd2: begin

                // blez

                if(ALUout) BranchTarget <= BranchAddr;

                else BranchTarget <= ID/EX.PCplus4;

            end

            3'd3: begin

                // bgtz

                if(ALUout) BranchTarget <= BranchAddr;

                else BranchTarget <= ID/EX.PCplus4;

            end

            3'd4: begin

                // bltz

                if(ALUout) BranchTarget <= BranchAddr;

                else BranchTarget <= ID/EX.PCplus4;

            end

            default: begin

                BranchTarget <= ID/EX.PCplus4;

            end

        endcase

ALU操作为SUB，其余指令都进行了相应的ALU操作分配，使其为1时跳转。

### PC：PC寄存器及其选择

在我设计的PC模块中，不仅包括了寄存器功能，还包括了下一个指令的选择。思路为：

* 默认PC <= PC + 4
* 若IF/ID.PCSrc == 2’b10或2’b11，说明此时跳转指令执行到ID阶段，下一周期将跳转到target
* 若ID/EX.PCSrc == 2’b01，说明此时branch指令执行到EX阶段，下一周期将跳转到BranchTarget
* 若PCcont == 1，说明PC需要保持，也即不用做任何赋值操作。

判断过程：

always @(posedge clk or posedge reset) begin

        if (reset) begin

            // reset

            pc <= 32'h00400000;

        end

        else if(!PCcont) begin

            // doesn't need to keep

            if(IF/ID.PCSrc == 2'b10) begin

                // j or jal

                pc <= {IF/ID.PCplus4[31:28], target << 2};

            end

            else if(IF/ID.PCSrc = 2'b11) begin

                // jr or jalr

                pc <= RegReadData1;

            end

            else if(ID/EX.PCSrc == 2'b01) begin

                // branch

                pc <= BranchTarget;

            end

            else begin

                pc <= pc + 4;

            end

        end

        else begin

            // keep

        end

    end

其它级间寄存器大致同理，不再赘述。

## 连线（顶层设计）

由于时序逻辑（寄存器）电路和复杂的组合逻辑部件已经封装为模块，因此只需将它们之间的线连好即可。不再赘述。

# CPU代码调试

## 指令测试

注：此阶段测试仅测试单条指令运行情况，不考虑冒险。因此每个指令间有4个nop。

**test1：****addi，addiu，slti，sltiu，andi，ori**

测试指令：

addi，addiu，slti，sltiu，andi，ori

汇编指令：

addi $a0 $0 1

addiu $a0 $a0 1

slti $a1 $a0 3

sltiu $a2 $a0 -1

andi $t0 $a0 255

ori $t1 $a0 255

机器码：

0x20040001

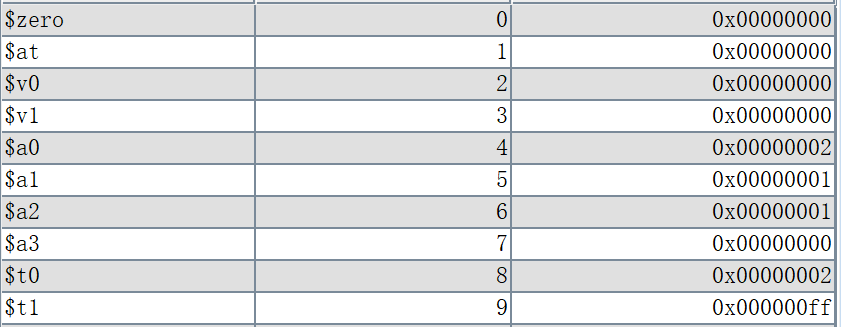
0x24840001

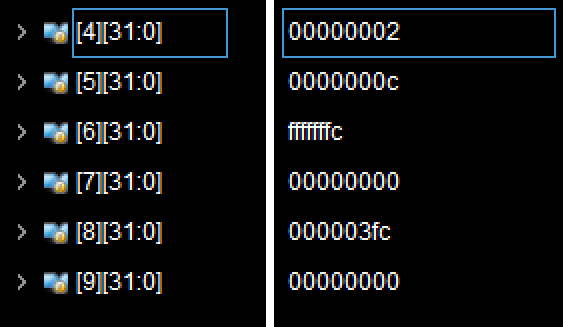
0x28850003

0x2c86ffff

0x308800ff

0x348900ff

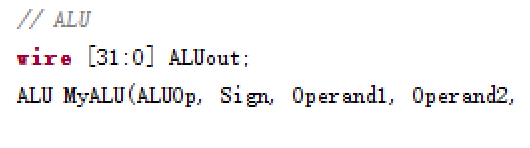
预期结果：

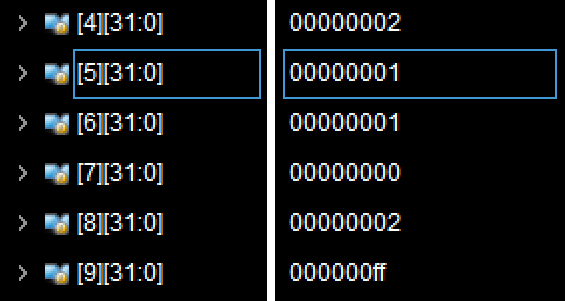
实际结果：

一塌糊涂。。。

Debug：

5号寄存器应为1，实际为c，说明slti $a1 $a0 3指令出现问题。

检查发现，虽然ALU的操作数一个是2一个是3，但输出直接给了c。检查ALUCtrl信号，当进行到EX阶段时，传给ALU的ALUCtrl是6（SL，即nop的结果）而不是8（Shift Left）。发现是由于给ALU接线时，接的是当时的ALUOp，而不是经过ID/EX寄存器之后的ALUOp。因此前两个指令对了，也仅是一个happy coincidence。

debug后的结果：

非常nice。

**test2：****add, addu, sub, subu, and, or, xor, nor**

测试指令：add, addu, sub, subu, and, or, xor, nor

汇编指令：

addi $1 $0 2022

addiu $2 $0 2059

addi $3 $1 -3599

add $4 $1 $2

addu $5 $2 $3

sub $6 $3 $4

subu $7 $4 $5

and $8 $5 $6

or $9 $6 $7

xor $10 $7 $8

nor $11 $8 $9

机器码：

0x200107e6

0x2402080b

0x2023f1f1

0x00222020

0x00432821

0x00643022

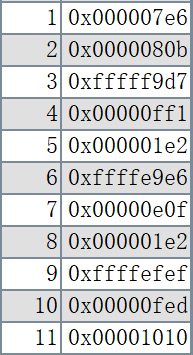
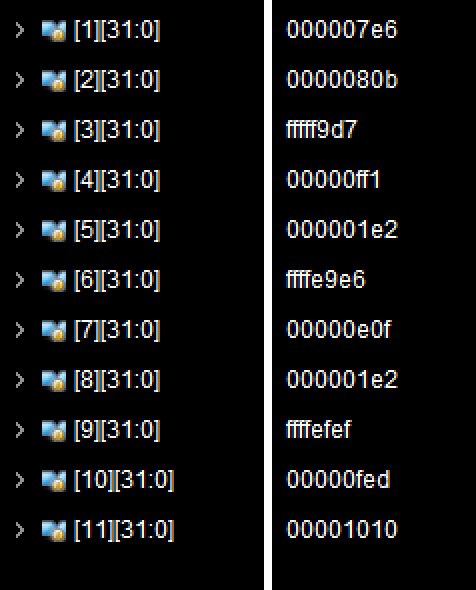
0x00853823

0x00a64024

0x00c74825

0x00e85026

0x01095827

预期结果： 实际结果：

完全一致，不错。

**test3：slt，sltu**

在test2的代码后加上两行：

slt $12 $8 $9

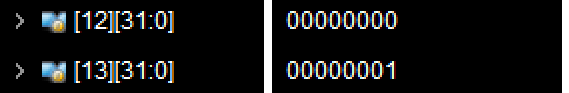
sltu $13 $8 $9

机器码：

0x0109602a

0x0109682b

预期结果：

实际结果：

一致。

（实际上测试的时候出过问题，发现只是把机器码copy错了）

**test4：lui, sll，srl, sra**

测试代码：

addi $1 $0 1234567890(伪代码)

sll $2 $1 4

srl $3 $2 4

sra $4 $2 6

机器码：

0x3c014996

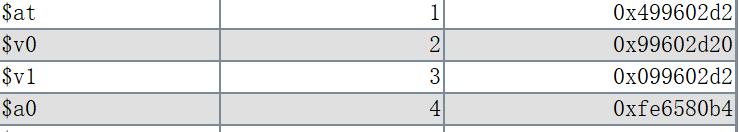
0x342102d2

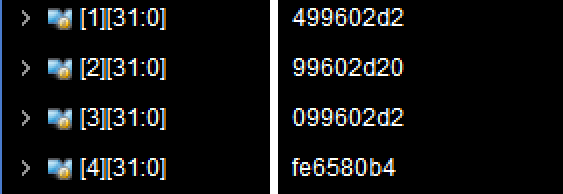
0x00010820

0x00011100

0x00021902

0x00022183

预期结果：

实际结果：

结果一致。

**test5：lw, sw**

测试代码：

addi $2 $0 5

sw $2 12($0)

lw $3 12($0)

机器码：

0x20020005

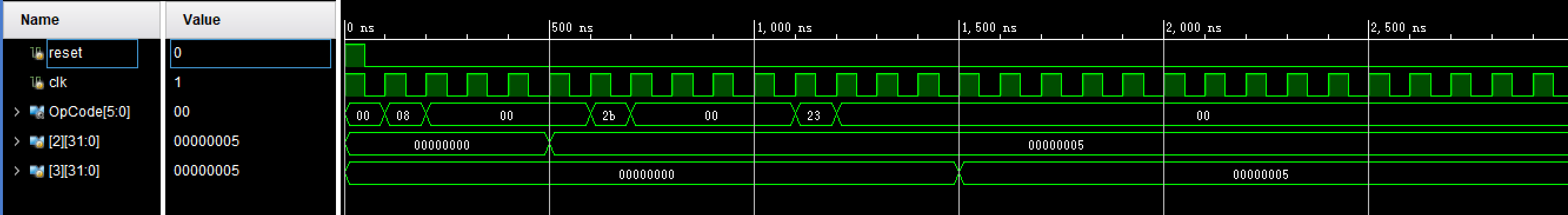
0xac02000c

0x8c03000c

预期结果：

$2 5

$3 5

测试结果：

与预期一致。

**test6：j, jal, jr, jalr**

测试代码：

addi $1 $0 1

jal test1

addi $3 $0 0x0040001c

jalr $3

j end

test1:

add $2 $1 $1

jr $ra

test2:

add $4 $2 $2

jr $ra

end:

addi $5 $0 5

机器码：

20010001

0c100007

3c010040

34210024

00011820

0060f809

0810000b

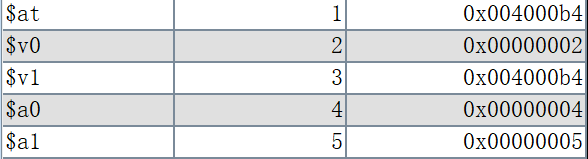
00211020

03e00008

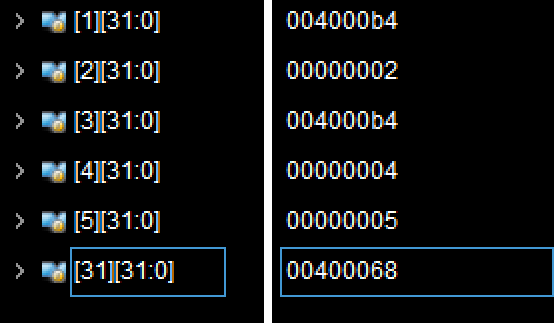
00422020

03e00008

20050005

预期结果：

同时$31为0x00400068

测试结果：

与预期结果一致。

**test7：beq**

测试代码：

addi $2 $0 78

addi $3 $0 79

addi $4 $0 79

beq $2 $3 wrong

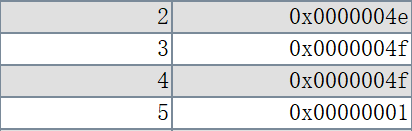
beq $3 $4 right

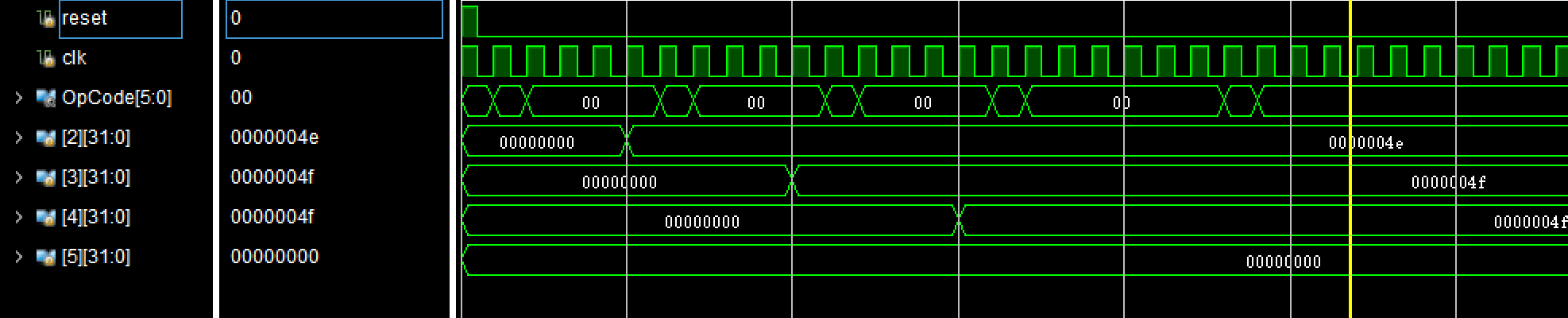
wrong:

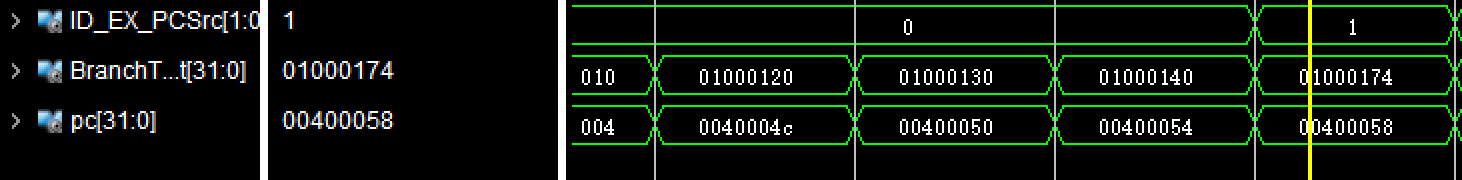
addi $5 $0 -1

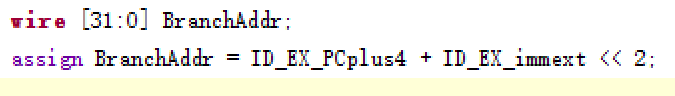
right:

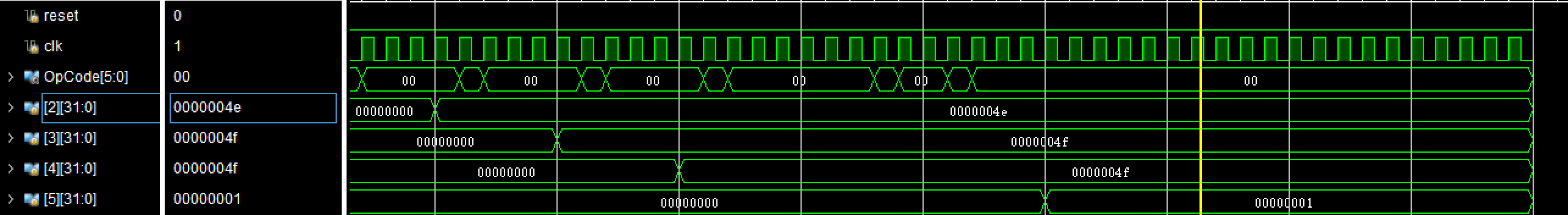
addi $5 $0 1

预期结果：

然而仿真时：

可以看到，在执行第一个beq时，如期进行两个stall，并继续进行第二个beq。但是第二个beq并没有如期进行跳转。

可以发现，BranchTarget并没有给出正确的地址，而其BranchAddr已经有误。

这里是BranchAddr的赋值，发现BranchAddr之所以离谱是因为+的优先级高于<<，所以它先相加再左移，拴Q了

调整后，可以得到预期结果。

**test8：bne**

测试代码：

addi $2 $0 78

addi $3 $0 79

addi $4 $0 79

bne $3 $4 wrong

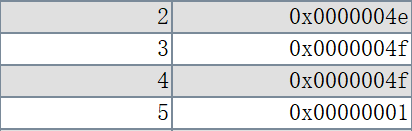
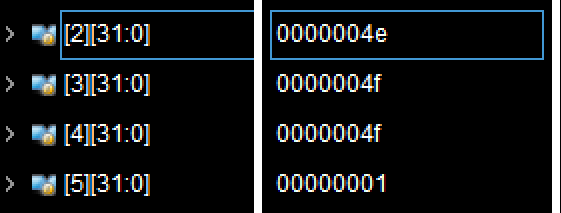
bne $2 $3 right

wrong:

addi $5 $0 -1

right:

addi $5 $0 1

预期结果： 测试结果：

与预期一致。

**test9：blez、bgtz、bltz**

对blez、bgtz、bltz进行类似的测试，均获得正确的跳转结果。由于代码类似，不再赘述。

综上，对每一条指令均进行了单独的验证，可以说明其可以进行正常运作（如果愿意每个指令都接四个nop。接下来进行冒险测试。

## 冒险测试

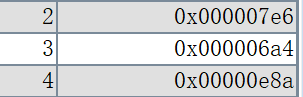
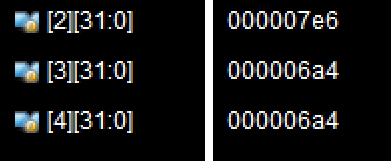
**test1：1-A & 2-C**

测试代码：

addi $2 $0 2022

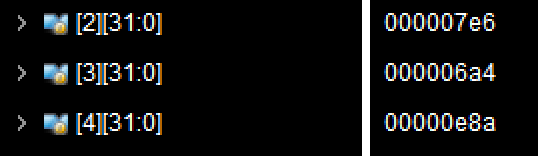
addi $3 $0 1700

add $4 $2 $3

预期结果： 实际结果：

通过这个错误，找到两个bug：

1. 在top文件里，Forward1和Forward2都设计成了1位信号
2. 连线的时候，误把MEM/WB.RegWrite传成了MEM/WB.RegWriteAddr

改正后，可得正确结果：

**test2：2-B**

测试代码：

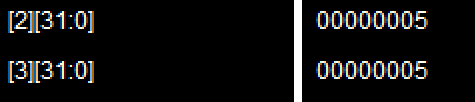
addi $2 $0 5

sw $2 12($0)

lw $3 12($0)

预期结果：

$2 5

$3 5

测试结果：

与预期一致。

**test3：load-use**

测试代码：

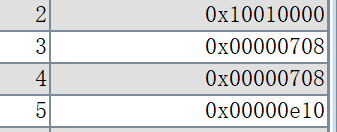
addi $2 $0 0x10010000

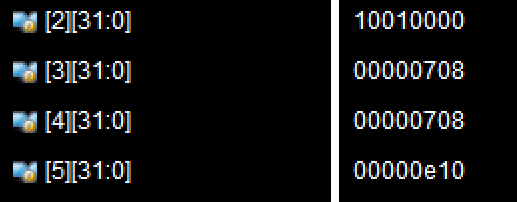
addi $3 $0 1800

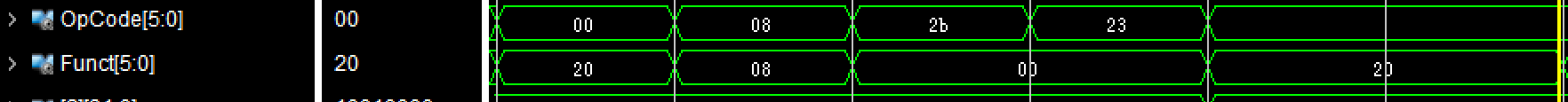
sw $3 0($2)

lw $4 0($2)

add $5 $4 $4

预期结果：

测试结果：

可见结果一致。

同时观察Funct信号，停留在20两个周期，说明由于load-use，add得到了keep。

控制冒险在单独测试指令的时候可以说明设计的完整性与正确性。

# 字符串查找的实现

## 仿真实现

### 代码编写

首先写一个简单的案例在MARS上试运行：

# str: 0x10010000

# pattern: 0x10010200

# load the str and the pattern

li $a1 0x10010000   # $a1: second argument:str

li $a3 0x10010200   # $a3: fourth argument:pattern

li $t0 1

li $t1 2

sw $t0 0($a1)

sw $t1 4($a1)

sw $t0 8($a1)

sw $t1 12($a1)

sw $t0 16($a1)

sw $t1 20($a1)

sw $t0 24($a1)

sw $t0 0($a3)

sw $t1 4($a3)

sw $t0 8($a3)

# find the length of the str and the pattern

# a0 : length of the str

# a2 : length of the pattern

add $a0 $0 $0

move $t0 $a1    # $t0: pointer

For\_begin\_str:

lw $t1 0($t0)   # $t1: element

beq $t1 $0 For\_end\_str

addi $a0 $a0 1

addi $t0 $t0 4

j For\_begin\_str

For\_end\_str:

add $a2 $0 $0

move $t0 $a3

For\_begin\_pattern:

lw $t1 0($t0)   # $t1: element

beq $t1 $0 For\_end\_pattern

addi $a2 $a2 1

addi $t0 $t0 4

j For\_begin\_pattern

For\_end\_pattern:

sll $a0 $a0 2

sll $a2 $a2 2

jal brute\_force

End:

beq $0 $0 End

brute\_force:

# # # # #  your code here # # # # #

li $t0 0            # $t0: i

li $t1 0            # $t1: j

li $v0 0            # $v0: cnt

bigger\_for\_judge:

sub $t2 $a0 $a2

blt $t2 $t0 bigger\_for\_exit

li $t1 0

smaller\_for\_judge:

slt $t2 $t1 $a2

beq $t2 $zero smaller\_for\_exit

add $t2 $t0 $t1     # i + j

add $t2 $t2 $a1     # adddress of str[i + j]

lw $t2 0($t2)

add $t3 $a3 $t1     # address of pattern[j]

lw $t3 0($t3)

bne $t2 $t3 smaller\_for\_exit

addi $t1 $t1 4

j smaller\_for\_judge

smaller\_for\_exit:

bne $t1 $a2 no\_need\_to\_add\_one

addi $v0 $v0 1

no\_need\_to\_add\_one:

addi $t0 $t0 4

j bigger\_for\_judge

bigger\_for\_exit:

jr $ra

这段代码的大致意思为：

先用sw在内存中放置好字符串和模式串。在实际完成时，可以事先在内存中初始化，以避免用代码完成过于繁琐。在这个程序中，字符串是1212121，模式串是121。并给出二者首地址。

之后凭借两个字符串的首地址，判断出字符串和模式串的长度。

之后调用之前数逻大作业完成的代码。但是由于在该CPU不支持lb运算，因此需要修改。修改的地方包括：把lb变为lw，把相应的变量\*4。

最后完成的结果保存在$v0中，并循环重复一条beq指令以视结尾。

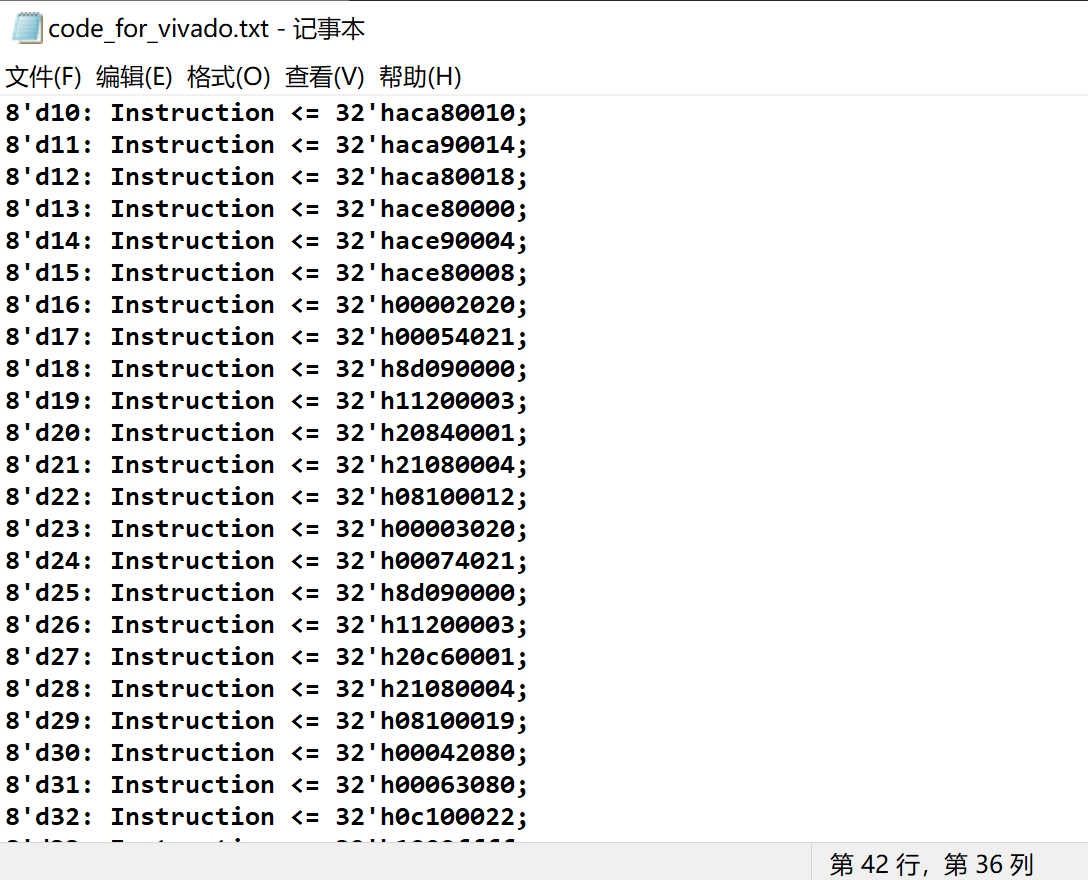
由于暴力算法、horspool和KMP算法都有现成的汇编代码，因此他们之间没有本质区别，在此选用暴力算法完成任务。

### 将较复杂代码写入vivado的指令存储器

将指令一条条复制到指令存储器是很繁琐而浪费时间的。采取下面方法加快该进程：

1. 将MARS生成的机器码以十六进制文本的形式导出。
2. 编写python程序，为机器码添加统一的格式。

file1 = open('C:\\Users\\liyutong\\Desktop\\code.txt', 'r')  
asmCode = file1.read()  
file1.close()  
  
asmCode = asmCode.split("\n")  
  
file2 = open('C:\\Users\\liyutong\\Desktop\\code\_for\_vivado.txt', 'w')  
i = 0  
for code in asmCode:  
 print(file2.write("8'd" + str(i) + ": Instruction <= 32'h" + code + ';\n'))  
 i = i + 1

1. 复制粘贴进入指令存储器即可。

### 仿真结果

在程序运行足够长的事件后，得到了预期的结果。

### 添加外设

在完成CPU的基本实现之后，可以在思维上将其封装，并添加外设接口。但由于要求利用软件的方法完成外设，将BCD绑定于存储器上，因此需要调整内存的状况，同时在程序主体结束后添加外设部分。则修改内存如下：

// Read data

assign ReadData = MemRead == 1'b0 ? 32'h0 :

                Address == 32'h4000000c ? {20'h0, AN, BCD} :

                Address == 32'h40000010 ? {24'b0, leds} :

                Address == 32'h40000014 ? Counter : RAM\_data[Address[9:2]];

// Write data

integer i;

always @\* begin

    if(reset) begin

        AN <= 4'b0;

        BCD <= 8'b0;

        for (i = 0; i < RAM\_SIZE; i = i + 1)

            RAM\_data[i] <= 32'h00000000;

    end

    else if(MemWrite) begin

        if(Address == 32'h4000000c) begin

            AN <= WriteData[11:8];

            BCD <= WriteData[7:0];

        end

        else if(Address == 32'h40000010) begin

            leds <= WriteData[7:0];

        end

        else begin

            RAM\_data[Address[9:2]] <= Writedata;

        end

    end

end

always @(posedge reset or posedge clk) begin

    if(reset) begin

        Counter <= 32'b0;

    end

    else begin

        Counter <= Counter + 1;

    end

end

注：对于内存的初始化有两种选择：一种是在程序正式开始之前，将数据sw放入内存；另一种是在DataMemory收到reset信号时，将数据加载进去。上述代码是第一种写法，后附的代码文件为第二种写法。

同时应在汇编程序之后添加用于显示的代码部分：

# Decode

# t0:

andi $a0 $v0 0x0000000f

jal Decode

move $t0 $v1

ori $t0 $t0 0x00000e00

# t1:

andi $a0 $v0 0x000000f0

srl $a0 $a0 4

jal Decode

move $t1 $v1

ori $t1 $t1 0x00000d00

# t2

andi $a0 $v0 0x00000f00

srl $a0 $a0 8

jal Decode

move $t2 $v1

ori $t2 $t2 0x00000b00

# t3

andi $a0 $v0 0x0000f000

srl $a0 $a0 12

jal Decode

move $t3 $v1

ori $t3 $t3 0x00000300

li $a0 0x10010400

Display\_t0:

sw $t0 0($a0)

jal count

sw $t1 0($a0)

jal count

sw $t2 0($a0)

jal count

sw $t3 0($a0)

jal count

j Display\_t0

count:

li $s0 0

li $s1 2500

add\_one:

addi $s0 $s0 1

bne $s0 $s1 add\_one

jr $ra

Decode:

# a0 16������

beq $a0 0 zero

beq $a0 1 one

beq $a0 2 two

beq $a0 3 three

beq $a0 4 four

beq $a0 5 five

beq $a0 6 six

beq $a0 7 seven

beq $a0 8 eight

beq $a0 9 nine

beq $a0 10 aa

beq $a0 11 bb

beq $a0 12 cc

beq $a0 13 dd

beq $a0 14 ee

beq $a0 15 ff

li $v1 0xff

j ok

zero:

li $v1 0xc0

j ok

one:

li $v1 0xf9

j ok

two:

li $v1 0xa4

j ok

three:

li $v1 0xb0

j ok

four:

li $v1 0x99

j ok

five:

li $v1 0x92

j ok

six:

li $v1 0x82

j ok

seven:

li $v1 0xf8

j ok

eight:

li $v1 0x80

j ok

nine:

li $v1 0x90

j ok

aa:

li $v1 0x88

j ok

bb:

li $v1 0x83

j ok

cc:

li $v1 0xc6

j ok

dd:

li $v1 0xa1

j ok

ee:

li $v1 0x84

j ok

ff:

li $v1 0x8e

j ok

ok:

jr $ra

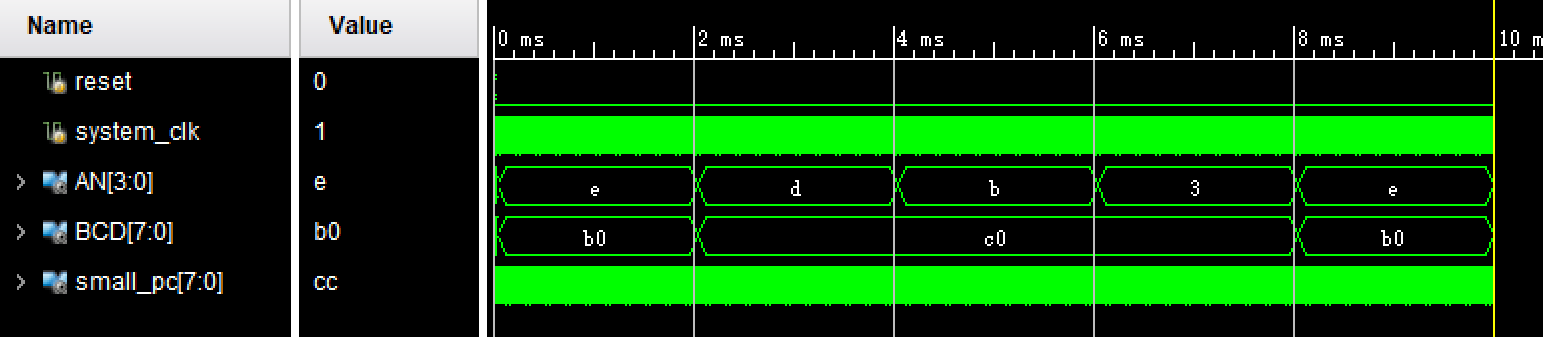
以上代码的含义为：

此前已将结果存在$v0中，目标将$v0的结果的十六进制低四位显示在七段数码管上。

将每一位进行相应的译码——这里采取的是一种类似于case的想法实现——并分别将译码后的结果加载到$t0~$t3中，并利用ori将使能信号与其拼接起来。

每过1ms，利用sw将结果写入0x40000010中。这里的1ms是利用循环计数实现的。

### 添加外设后的仿真结果

观察运行一段时间后的程序运行结果，得到预期结果:

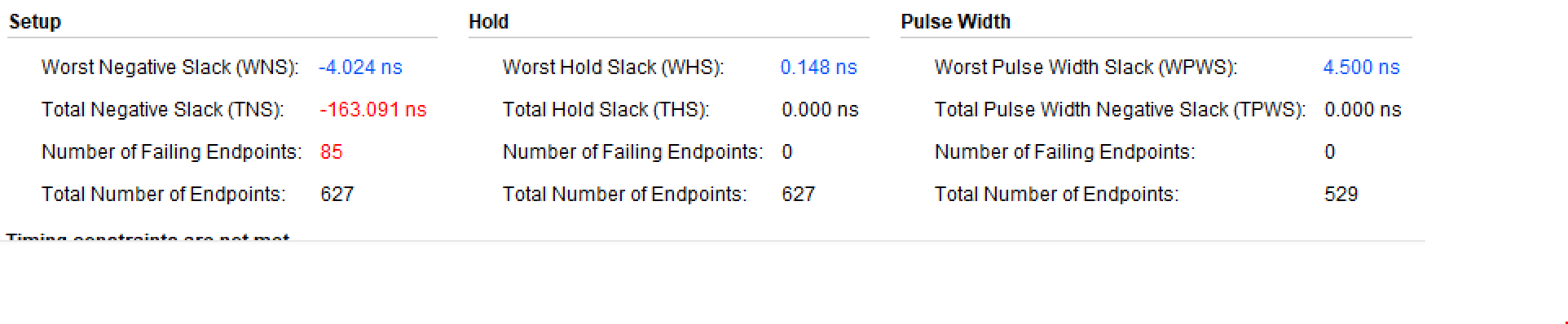
注：可以看到，显示的数字管每2ms换一次数字。实际上汇编代码完成的功能是1ms，上述结果是尝试了分频器后的结果。

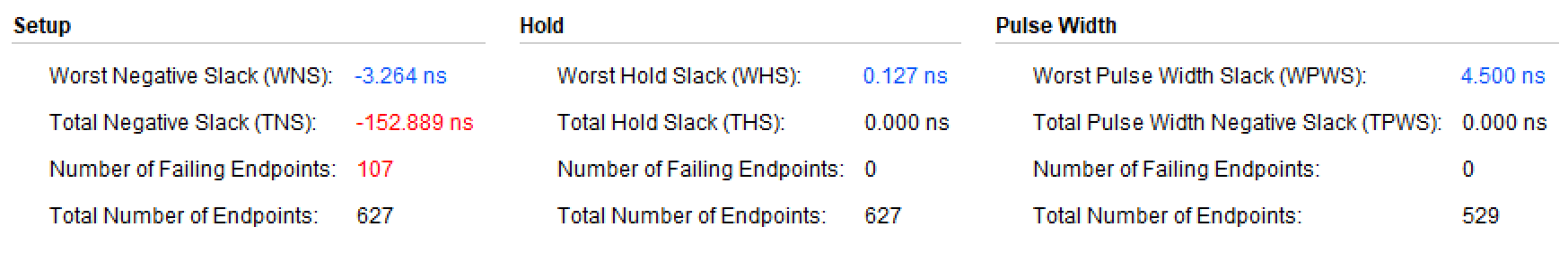
## 硬件调试

程序烧录后即可显示正确结果，具体情况已在视频验收中有所汇报。

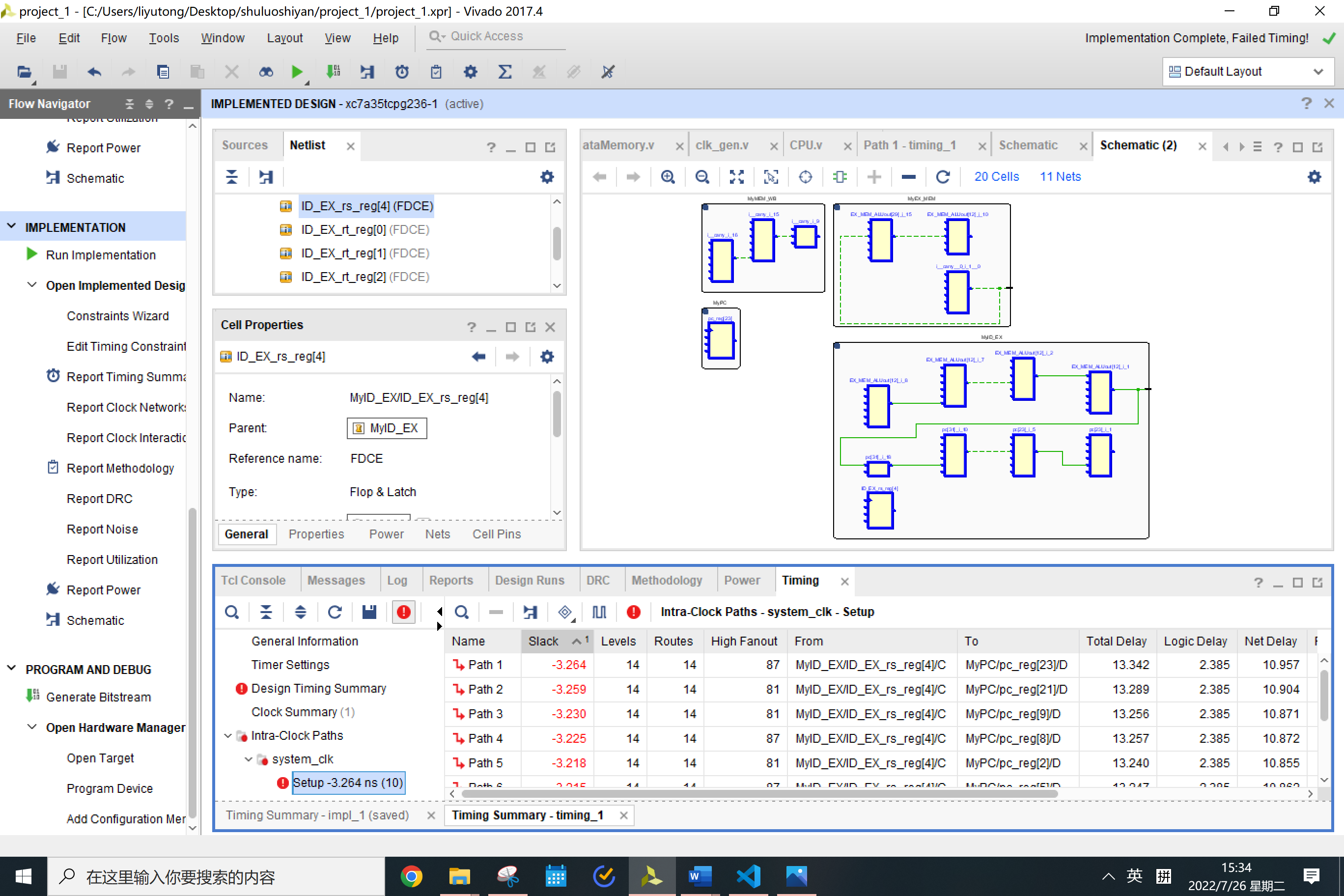
# 性能分析

## 时序性能

当设置时钟为10ns时，综合后的时序情况如下：

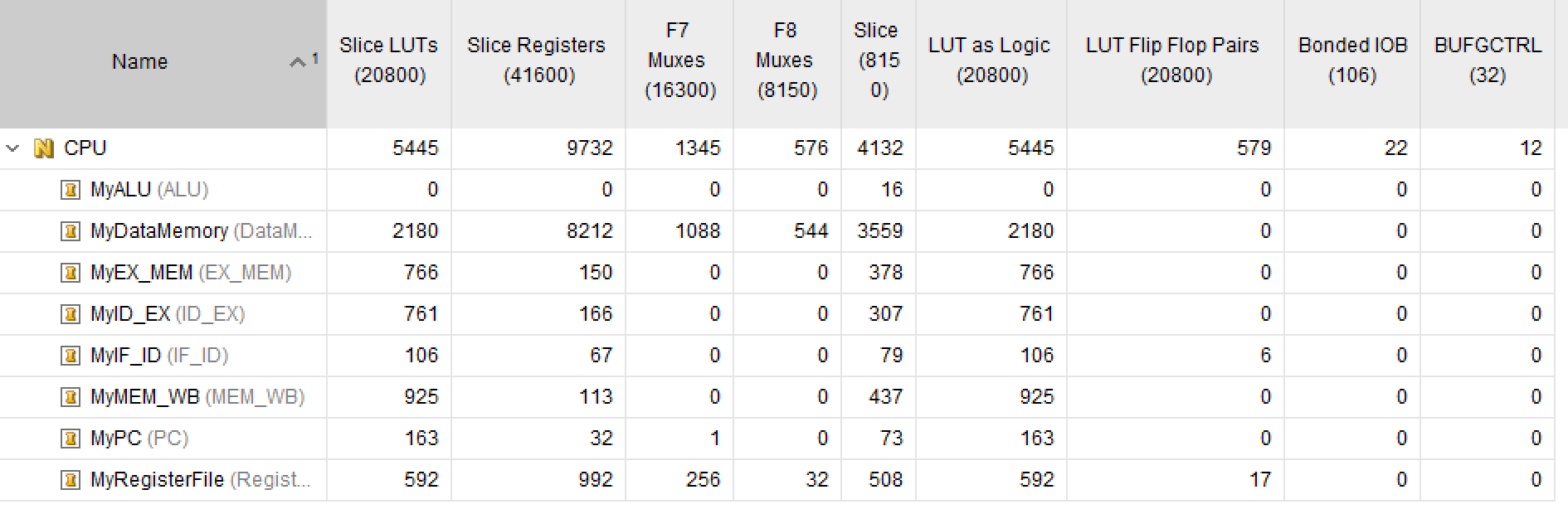
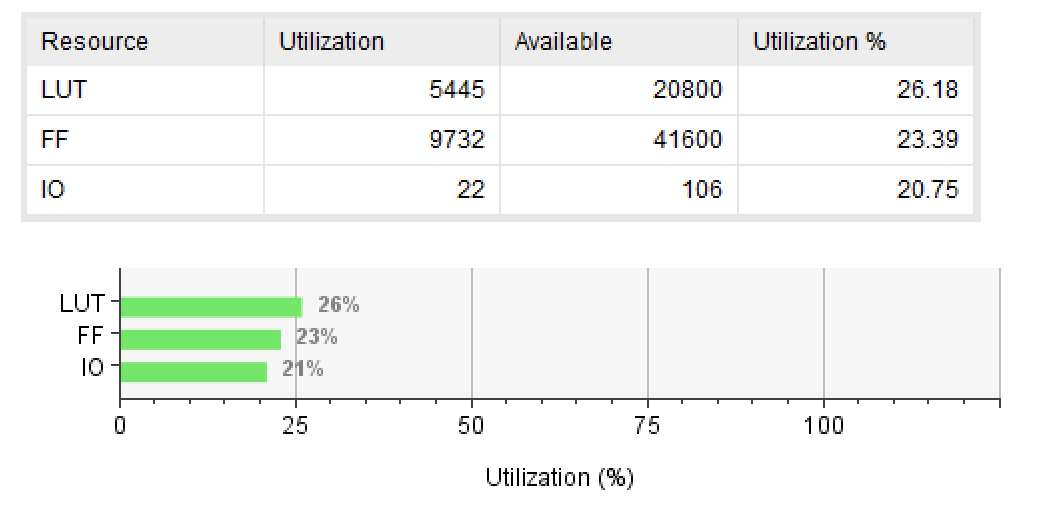
布线后的时序情况如下：

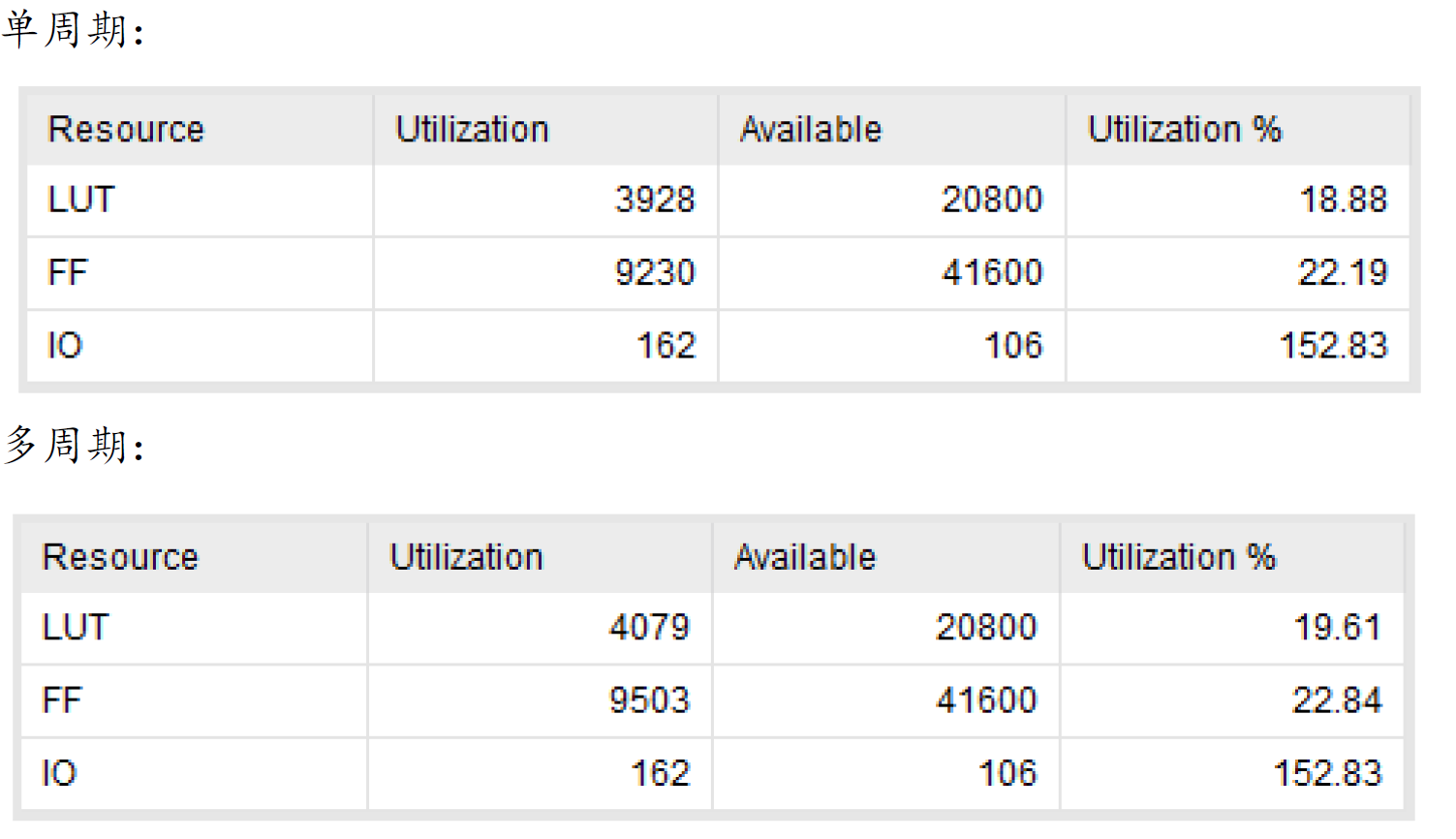
则最短时钟周期约为14.024ns，最高时钟频率约为。

观察关键路径如下：

可以发现，最长路径起始是ID/EX.rs，终点是pc，也即对于分支指令的判断和冒险是这里的最长路径。

## 逻辑资源

 该CPU占据的逻辑资源状况如下：

而以下是之前做单、多处理器的逻辑资源占用状况：

对比可知，流水线为了照顾不同指令的兼容性和处理各种各样的冒险，导致占用的资源较多。与此同时，其将指令拆分成多个周期并行的思想使其具有更高的时钟周期，使其执行效率更高。

# 文件清单

注：要求中提到需要“关键代码及文件清单”，关键代码在上述分析过程中已有相应的展示，不再赘述。下整理文件清单。

|  |  |  |
| --- | --- | --- |
| 文件夹 | 文件名 | 备注 |
| Asm\_Code | bf.asm | 用于在MARS上调试的代码 |
| bf\_vivado.asm | 变为机器码后适用于vivado |
| Constraints | CPU.xdc | 约束文件 |
| Design\_Source | ALU.v | 设计文件  具体含义在前均有介绍 |
| ALUControl.v |
| BranchUnit.v |
| clk\_gen.v |
| Control.v |
| CPU.v |
| DataMemory.v |
| Display.v |
| EX\_MEM.v |
| ForwardingUnit.v |
| HazardUnit.v |
| ID\_EX.v |
| IF\_ID.v |
| InstructionMemory.v |
| MEM\_WB.v |
| PC.v |
| RegisterFile.v |
| Other\_Docs | asmconverter.py | 给机器码加格式的python文件 |
| branch\_hazard.drawio | 报告中的图示 |
| branch\_hazard.drawio.png |
| data\_hazard.drawio |
| data\_hazard.drawio.png |
| jump\_hazard.drawio |
| jump\_hazard.png |
| test.drawio |
| test.png |
| 指令格式表及控制信号.xlsx | 指令格式表及控制信号 |
| Simulation\_Source | test\_cpu.v | 仿真文件 |

# 心得体会

流水线作为上半个学期的学习重点，虽然已经系统学习过其原理，但真正去实操起来才发现有很多细节之前都没有想明白，尤其是种种冒险。可以说这次作业是对以往内容的一个综合、全面而完整的回顾，无论是知识上还是思维上。

对于一个较为繁重的任务，我认为我这次的处理步骤是较为合理的：先确定指令和控制信号，之后边绘制线路图边思考冒险处理和连线，最后进行测试和分析——把任务拆解成小步骤然后一步一步进行——这是这次实验给予我的一些方法论的思考。

另外，我认为这次实验中我也有很多不足的地方，例如时钟频率较低还没有着手去进行改进、有些地方思维较为繁琐不够简洁等等。

也要感谢这学期以来，张老师、助教们的指导以及同学们的帮助，让我磕磕绊绊地收获许多。