数字逻辑与处理器基础实验

流水线MIPS处理器设计实验报告

2020010816

无07 陈宇阳

**目录**

[一、实验名称与内容 1](#_Toc110078748)

[二、处理器设计 1](#_Toc110078749)

[（一）实现的指令集 1](#_Toc110078750)

[（二）数据通路 1](#_Toc110078751)

[（三）处理器模块 2](#_Toc110078752)

[（四）工作过程 3](#_Toc110078753)

[I. 工作阶段 3](#_Toc110078754)

[II. 分支与跳转 3](#_Toc110078755)

[III. 冒险 4](#_Toc110078756)

[Ⅳ. 外设 6](#_Toc110078757)

[Ⅴ. 字符串匹配 7](#_Toc110078758)

[三、综合与实现情况 8](#_Toc110078759)

[（一）时序性能 8](#_Toc110078760)

[（二）逻辑资源占用情况 9](#_Toc110078761)

[（三）处理器CPI计算 10](#_Toc110078762)

[四、仿真验证 11](#_Toc110078763)

[五、硬件调试情况 12](#_Toc110078764)

[六、心得体会 13](#_Toc110078765)

[附录 14](#_Toc110078766)

[附录一 指令格式表 14](#_Toc110078767)

[附录二 指令说明表 15](#_Toc110078768)

# 一、实验名称与内容

实验名称：流水线MIPS处理器设计

实验内容： 将春季学期实验四设计的MIPS处理器改进为流水线结构， 并利用此处理器和理论课MIPS汇编语言大作业中的任意一种算法，求解字符串搜索问题。需要设计外设，解决流水线中的冒关联问题险和数据关联问题，最后做出测试验证和性能分析。

# 二、处理器设计

## （一）实现的指令集

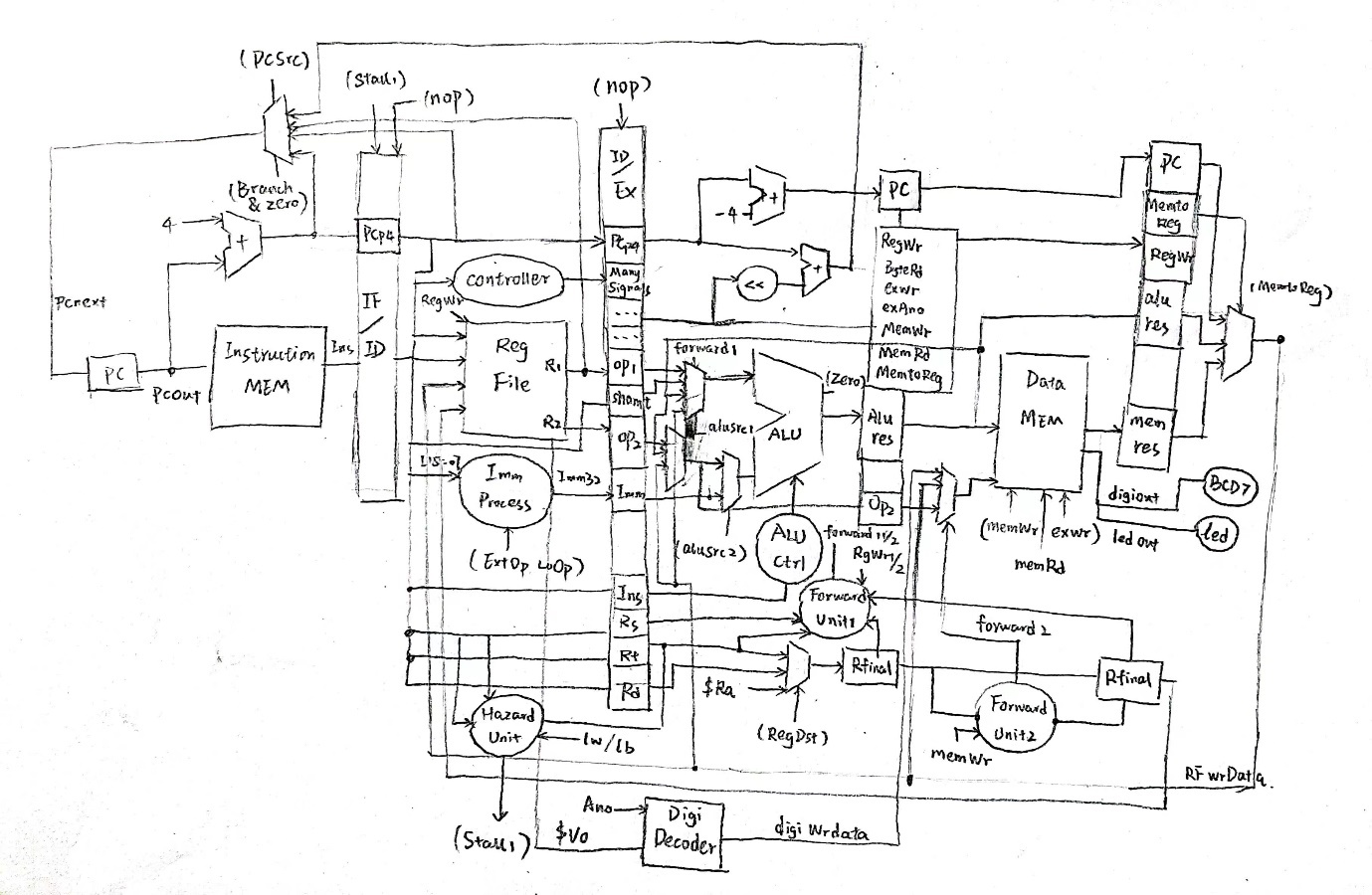
本次实验实现的指令集为：

1. R型算术指令：add、addu、sub、subu、and、or、xor、nor、slt、sltu、sll、、srl、sra；
2. I型算术指令：lui、addi、addiu、andi、sltiu；
3. 存取指令：lw、sw、lb、esw；
4. 分支指令：beq、bne、blez、bgtz、bltz；
5. 跳转指令：j、jal、jr、jalr。
6. 空指令 nop 即32’b0，sll 0 0 0

其中存取指令中的lb和esw是自己添加的指令，上述指令的具体格式与各个指令的作用参见附录一与附录二。

## （二）数据通路

数据通路图如下：



## （三）处理器模块

根据上述指令集，我设计了如下模块：

1. waterCPU：总连线以及冒险控制模块；
2. Controller：控制信号模块；
3. ALU：算术逻辑单元模块；
4. RegisterFile：寄存器堆模块；
5. DataMemory\_new：加入了外设地址的数据存储器模块；
6. RegTemp： 寄存器模块，实例化为了PC计数器
7. InstructionMemory：指令存储器模块；
8. Digi\_decoder：外设BCD解码控制模块；
9. ImmProcess：立即数扩展模块；
10. RegIF\_ID、RegID\_EX、RegEX\_MEM、RegMEM\_WB：流水线寄存器模块；
11. TestCPU：Test bench模块

## （四）工作过程

### I. 工作阶段

该处理器为五级流水线处理器，每条指令基本可以分为5个阶段完成。

IF阶段负责程序计数器的更新以及取指令，以PC寄存器的输出为指令地址得到指令，并储存在IF/ID寄存器中。

ID阶段负责指令的解码、寄存器堆的读取和立即数扩展，并进行j、jr、jal、jalr跳转指令的判断。本阶段将对读入的指令进行解码，生成所有控制信号，同时从寄存器堆中读取需要的寄存器数据以及进行对应的立即数扩展，所有产生的数据存入到ID/EX寄存器中。

EX阶段进行ALU运算，以及将要写入的寄存器地址RFinal的选择，将ALU计算结果以及其他控制信号写入EX/MEM。同时进行各种Branch指令的计算以及PC的跳转。

MEM阶段进行数据存储器以及外设的读取或写入。

WB阶段选择写入寄存器堆的数据，并将数据写入寄存器堆。

### II. 分支与跳转

**核心代码**

assign PCnext = (Branch\_EX&&Zero)?PCp4\_EX+(Imm32\_EX<<2): // beq EX priority

                    (PCSrc\_ID==2'b10)?{PCp4\_ID[31:28],Ins\_ID[25:0],2'b00}: // jump ID

                    (PCSrc\_ID==2'b11)?Op1\_ID: // jump reg ID

                    PCp4\_origin;    // PC+4

J型指令在ID阶段跳转，Branch类型指令在EX阶段跳转，如果Branch指令后紧接着一条J型指令，则应该先判断Branch的跳转再决定J是否跳转。

分支指令仅仅使用控制信号Branch和ALU的输出zero来进行控制。由于ALU的zero输出作用较小，因此我扩展了其功能：zero并非在结果等于0时输出才为1，对于特定的branch指令，ALU会进行相应的判断，并在符合分支条件时输出zero为1 。这样就简化了分支跳转的控制，当且仅当EX阶段的Branch信号和ALU输出的Zero同时为1时，PC会发生Branch跳转。

跳转指令通过PCSrc控制信号控制PC的写入，在每个周期上升沿PC都会根据PCSrc来决定自己更新什么数据。J型和jr有着不同的PCSrc控制信号。

### III. 冒险

**冒险判断核心代码**

// hazard control

    assign stall1 = ((Ins\_ID[25:21]==Rt\_EX||Ins\_ID[20:16]==Rt\_EX)&&(Ins\_EX[31:26]==6'h23||Ins\_EX[31:26]==6'h20))?1:0; // only when load-use

    assign forward1\_1 = (Rs\_EX==Rfinal\_MEM && RegWr\_MEM==1)?2'b01:

                        (Rs\_EX==Rfinal\_WB && RegWr\_WB==1)?2'b10:

                        0;

    assign forward1\_2 = (Rt\_EX==Rfinal\_MEM && RegWr\_MEM==1)?2'b01:

                        (Rt\_EX==Rfinal\_WB && RegWr\_WB==1)?2'b10:

                        0;

    assign Op2\_reg = (forward1\_2==2'b01)?alures\_MEM:

                     (forward1\_2==2'b10)?RFWr\_data:

                     Op2\_EX;

    assign forward2 = (MemWr\_MEM==1&&(Rfinal\_MEM==Rfinal\_WB))?1:0;

    assign Null\_IFID = ((Branch\_EX&&Zero)||PCSrc\_ID==2'b10||PCSrc\_ID==2'b11)?1:0;  // jump or beq

assign Null\_IDEX = ((Branch\_EX&&Zero)||stall1)?1:0;

整个处理器中有三个冒险控制单元，分别对应forward1 / forward2 两个控制单元，还有一个流水线stall的控制单元。

1. 数据冒险

数据冒险来自于上一条指令尚未进入写入寄存器的WB阶段，后面的指令就在EX阶段需要调用同一个寄存器的数据。我们在EX阶段把该阶段对应的指令所需要写入的寄存器记为Rfinal并在pipeline中传递下去，通过与EX阶段Rs和Rt的比较，判断是否产生数据冒险。若产生了，修改对应forward1\_1和forward1\_2的控制信号，则将在MEM或者WB阶段待写入的数据先行转发到ALU的输入端。

**转发代码：**

alu1 =  (forward1\_1==2'b01)?alures\_MEM:

                (forward1\_1==2'b10)?RFWr\_data:

                ((AluSrc1\_EX==1)?Ins\_EX[10:6]:Op1\_EX); // no forward Ins or Reg

        alu2 = (AluSrc2\_EX==1)?Imm32\_EX:Op2\_reg; // Imm or reg (have been forwarded)

在执行save word 指令的时候需要提取寄存器数据，同样会出现数据冒险。 我们在MEM阶段判断此阶段所使用的Rt是否与WB阶段将要写入的寄存器相同，同时MEM阶段是否需要写入内存，如果皆为真，则forward2控制信号为真。从WB阶段把将要写入的数据转发到MEM的写入端口。

**转发代码：**

assign MemWr\_data = (forward2==1)?RFWr\_data:

                        (ex\_WR\_MEM==1)?digi\_wr\_data:

                        Op2\_MEM;

ii) 控制冒险

当需要进行跳转时，会产生控制冒险，需要擦除掉已经执行的指令。跳转指令需要擦除一条指令，而分支指令需要擦除两条指令。

**核心代码**：

assign Null\_IFID = ((Branch\_EX&&Zero)||PCSrc\_ID==2'b10||PCSrc\_ID==2'b11)?1:0;  // jump or beq

    assign Null\_IDEX = ((Branch\_EX&&Zero)||stall1)?1:0;

擦除flush操作需要使用IFID pipeline和IDEX pipeline各一个Null(其实是flush，当时想不起来是什么单词了)控制信号，当对应的Null信号拉高时，所有寄存器在下个上升沿刷新为0，相当于插入了一条空指令。

iii) 阻塞解决load-use冒险

我们还没有解决load-use冒险，从MEM中取出来的数据要到MEM阶段末才可以获取，因此如果出现寄存器冲突，需要stall阻塞一个周期。使用stall1控制信号控制此进程。

**检测load-use冒险：**

assign stall1 = ((Ins\_ID[25:21]==Rt\_EX||Ins\_ID[20:16]==Rt\_EX)&&(Ins\_EX[31:26]==6'h23||Ins\_EX[31:26]==6'h20))?1:0; // only when load-use

当stall1信号拉高的时候，ID/EX pipeline需要插入一条nop指令(也即null)，同时IF/ID中的状态需要保持不变（即上升沿不更新数据），同时PC也不能更新。

**核心代码：**

assign PCp4\_origin = (stall1==1)?PCout:PCout+4;

iv) 先写后读

WB阶段我们不再设置转发单元，而采用先写后读的方式，以充分利用时间，节省硬件资源。我设计寄存器堆写入操作在时钟下降沿时进行，这样就可以保证在同一周期WB数据写入一定在ID数据读出以前进行。

### Ⅳ. 外设

本次实验需要设计外设，并通过软件编码的方式使BCD管扫描显示数字。外设所需要的数据通过特定的编码方式储存在Data Memory的特定地址空间中。处理器采用哈弗结构，指令储存与数据储存分离，外设地址设置在数据储存当中。由于在FPGA板上访问内存并不需要额外的时间，因此可以直接连线到内存中获取外设所对应的数据。

通过ex\_wr控制信号控制数据写入到特定的地址。

**核心代码：**

else if (ex\_wr) begin

            casez(Address)

                32'h4000000C: led <= Write\_data[7:0];

                32'h40000010: digi <= Write\_data[11:0];

            endcase

        end

// 其中write\_data通过一个digi\_decoder模块生成，实现了寄存器值与扫描对应ano的结合。

`timescale 1ns / 1ps

module Digi\_decoder(

    input wire [31:0] v0,

    input wire [1:0] ano,

    output reg [11:0] code\_o

);

wire [3:0] num;

assign num =(ano == 2'b00)?v0[3:0]:

            (ano == 2'b01)?v0[7:4]:

            (ano == 2'b10)?v0[11:8]:

            v0[15:12];

always @(\*)

begin

    code\_o[7] <= 0;

    casez(num)

            4'h0: code\_o[6:0] <= 7'b0111111;

            4'h1: code\_o[6:0] <= 7'b0000110;

            4'h2: code\_o[6:0] <= 7'b1011011;

            4'h3: code\_o[6:0] <= 7'b1001111;

            4'h4: code\_o[6:0] <= 7'b1100110;

            4'h5: code\_o[6:0] <= 7'b1101101;

            4'h6: code\_o[6:0] <= 7'b1111101;

            4'h7: code\_o[6:0] <= 7'b0000111;

            4'h8: code\_o[6:0] <= 7'b1111111;

            4'h9: code\_o[6:0] <= 7'b1101111;

            4'hA: code\_o[6:0] <= 7'b1110111;

            4'hB: code\_o[6:0] <= 7'b1111100;

            4'hC: code\_o[6:0] <= 7'b0111001;

            4'hD: code\_o[6:0] <= 7'b1011110;

            4'hE: code\_o[6:0] <= 7'b1111001;

            4'hF: code\_o[6:0] <= 7'b1110001;

            default: code\_o[6:0] <= 7'b0;

        endcase

    casez(ano)

        2'b00: code\_o[11:8] <= 4'b0001;

        2'b01: code\_o[11:8] <= 4'b0010;

        2'b10: code\_o[11:8] <= 4'b0100;

        2'b11: code\_o[11:8] <= 4'b1000;

        default: code\_o[11:8] <= 4'b0;

    endcase

end

endmodule

最后只需要在程序执行完成后，通过循环指令实现扫描即可。

 8'd3: Instruction <= {6'h3F,5'd0,5'd2,5'd0,5'd0,6'h0};

        //      esw2 $v0[11:8] $digi

        8'd4: Instruction <= {6'h3F,5'd0,5'd2,5'd0,5'd0,6'h1};

        //      esw3 $v0[7:4] $digi

        8'd5: Instruction <= {6'h3F,5'd0,5'd2,5'd0,5'd0,6'h2};

        //      esw4 $v0[3:0] $digi

        8'd6: Instruction <= {6'h3F,5'd0,5'd2,5'd0,5'd0,6'h3};

        //     beq $zero,$zero,Loop # always loop

        8'd7: Instruction <= {6'h04, 5'd0, 5'd0, 16'hFFFB};

### Ⅴ. 字符串匹配

由于字符都是ASCII码，每个字符仅使用1Byte的储存空间，因此如果使用原本以word为基本单位的内存，使用lw和sw指令来处理内存中的每个字符，将导致大量内存的浪费。

因此我改变了内存的储存结构，以Byte为单位，并增加了lb指令，在取都内存时可以读任意地址，而lw指令则只能读字对齐的地址。而sw指令需要一次写入四个地址(4Byte = 1 word)

**核心代码：**

 assign Read\_data =  MemRead? {RAM\_data[base\_Address],RAM\_data[base\_Address+2'b01],RAM\_data[base\_Address+2'b10],RAM\_data[base\_Address+2'b11]}:

ByteRead? {24'h000000,RAM\_data[Address[9:0]]}: 32'h00000000;

**// -----------------------------------------------------------**

else if (MemWrite)

            // fetch data: 32bit

            begin

            RAM\_data[base\_Address] <= Write\_data[31:24];

            RAM\_data[base\_Address+2'b01] <= Write\_data[23:16];

            RAM\_data[base\_Address+2'b10] <= Write\_data[15:8];

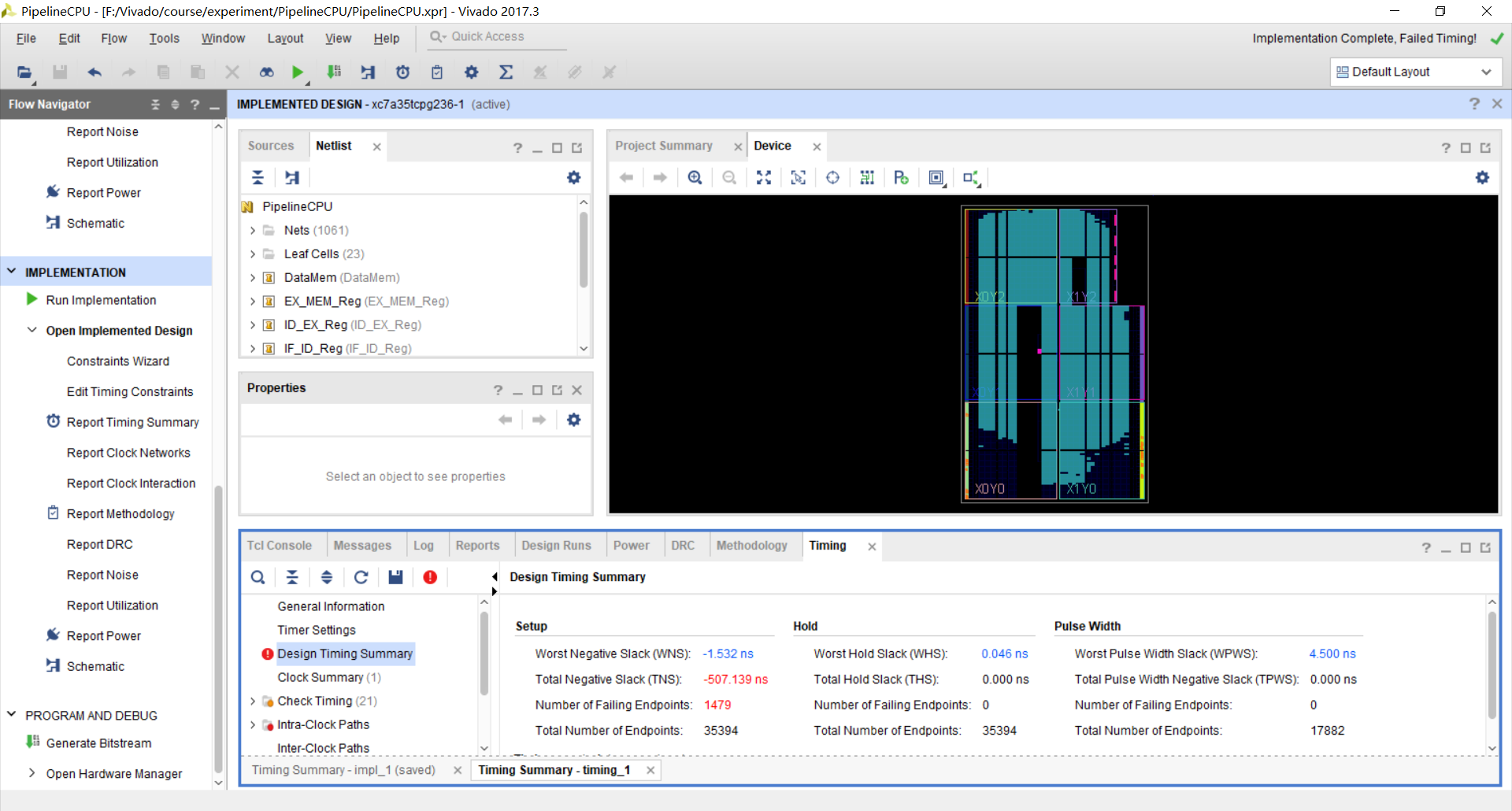
            RAM\_data[base\_Address+2'b11] <= Write\_data[7:0];

            end

# 三、综合与实现情况

## （一）时序性能

将时钟周期设置为10.00ns，观察综合后的时序裕量为-1.532ns：

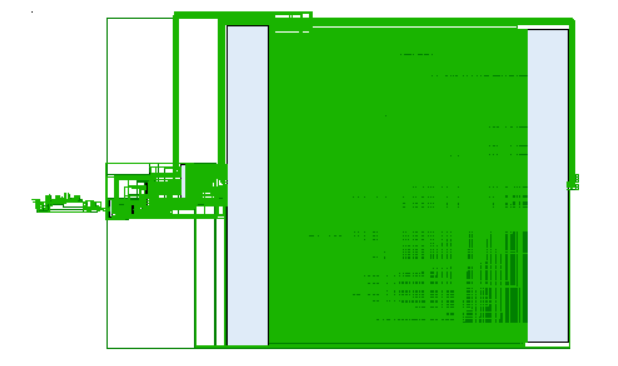


时序裕量为负值，说明时钟周期设置偏短。

最短时钟周期Tmin = 11.53ns

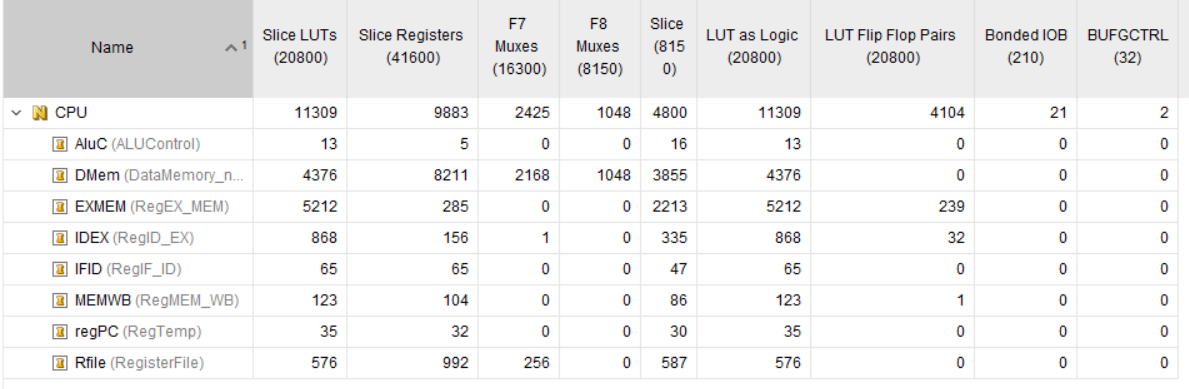
最高时钟频率fmax = 86.7MHz

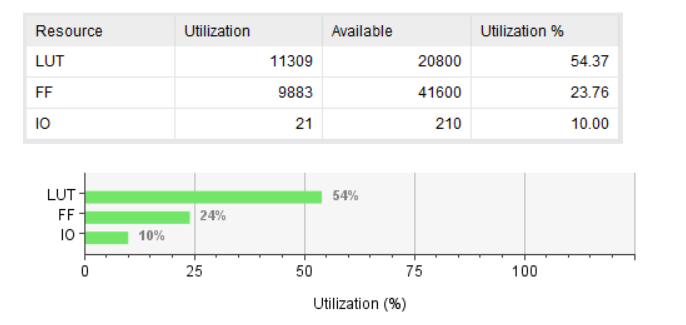
**Schematic：**

****

## （二）逻辑资源占用情况

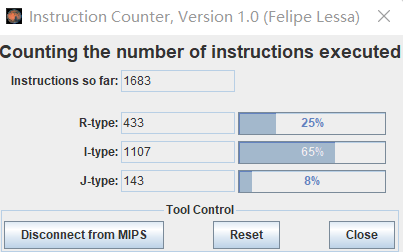
处理器的逻辑资源占用情况如下：





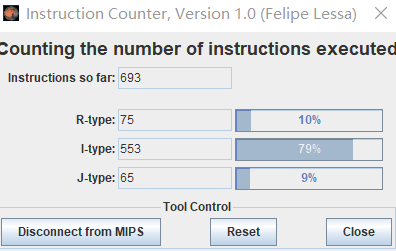
## （三）处理器CPI计算

**指令数：**



程序指令数计算：1683 ，考虑到前期使用了较多指令进行文件读取，而在仿真时文件读取直接通过初始化完成，故需要减去一部分指令。

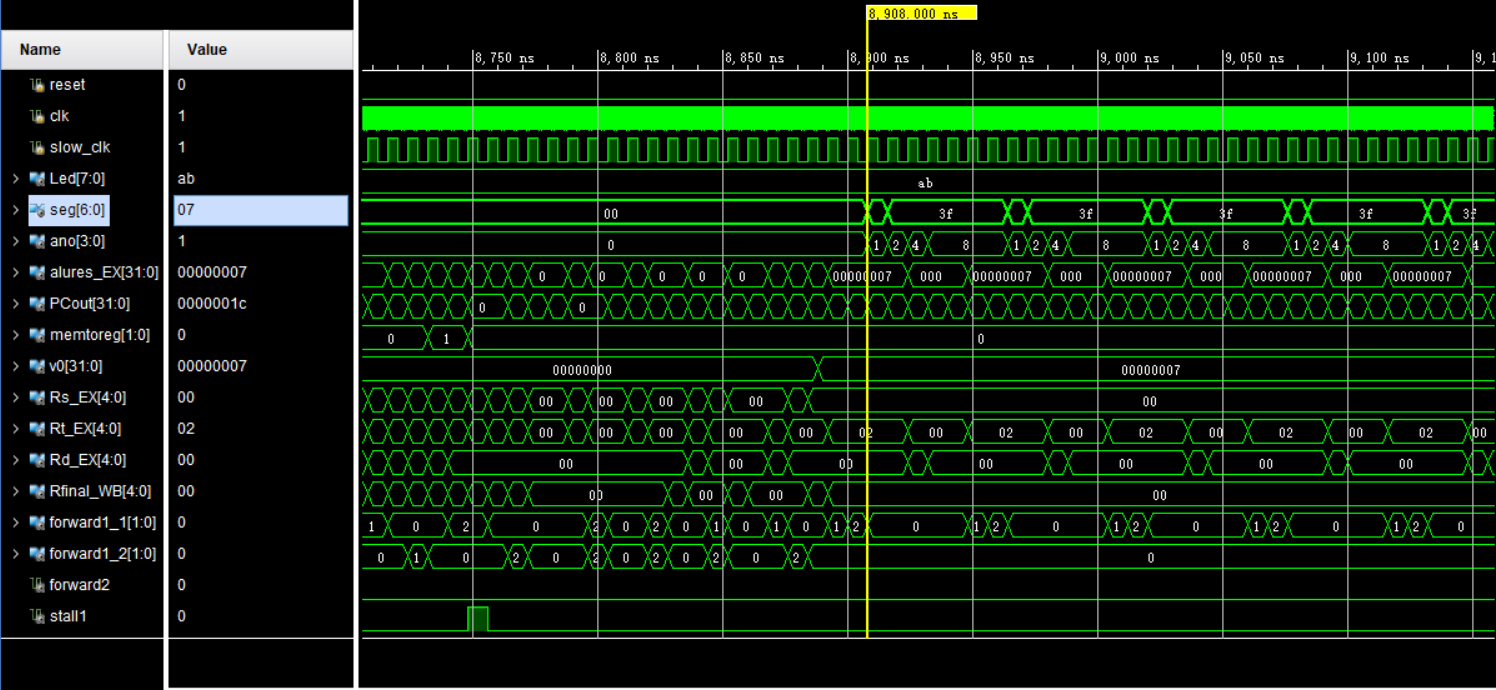
注释掉jal到字符串匹配的指令后再计算指令数如下：



可以认为，运行brute-force汇编程序一共使用了990条指令。

**时钟周期数：**

由仿真测试可以得到，从开始运行到外设显示循环，一共经历了1113个周期。



故程序的CPI = C/N = 1.124 (各种冒险和阻塞使得CPI高于1)

每秒执行的指令数：1s / (Tmin × CPI) = 7.70 M(条)

# 四、仿真验证

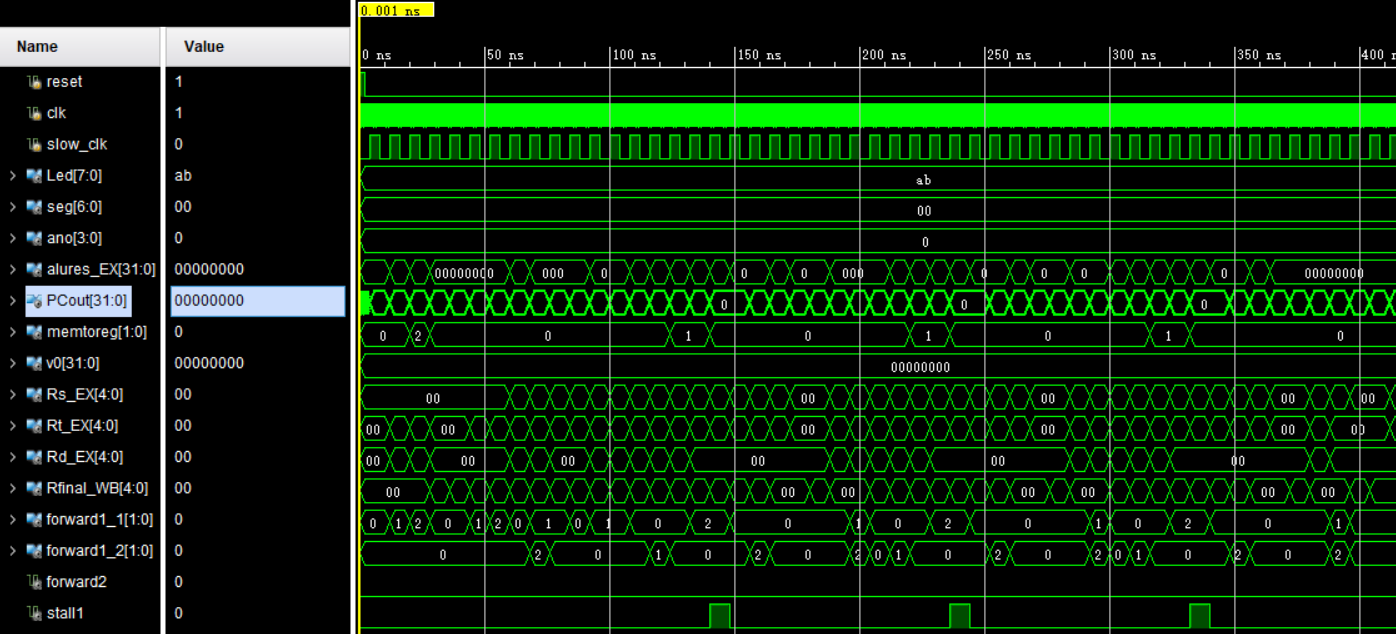
仿真测试直接使用最终要跑的brute-force字符串匹配代码。再字符串匹配代码中存在大量的数据冒险和控制冒险，如果能得到正确的结果，可以认为处理器工作正常。

测试语句：

Str: can a canner can a can like a canner can a can?

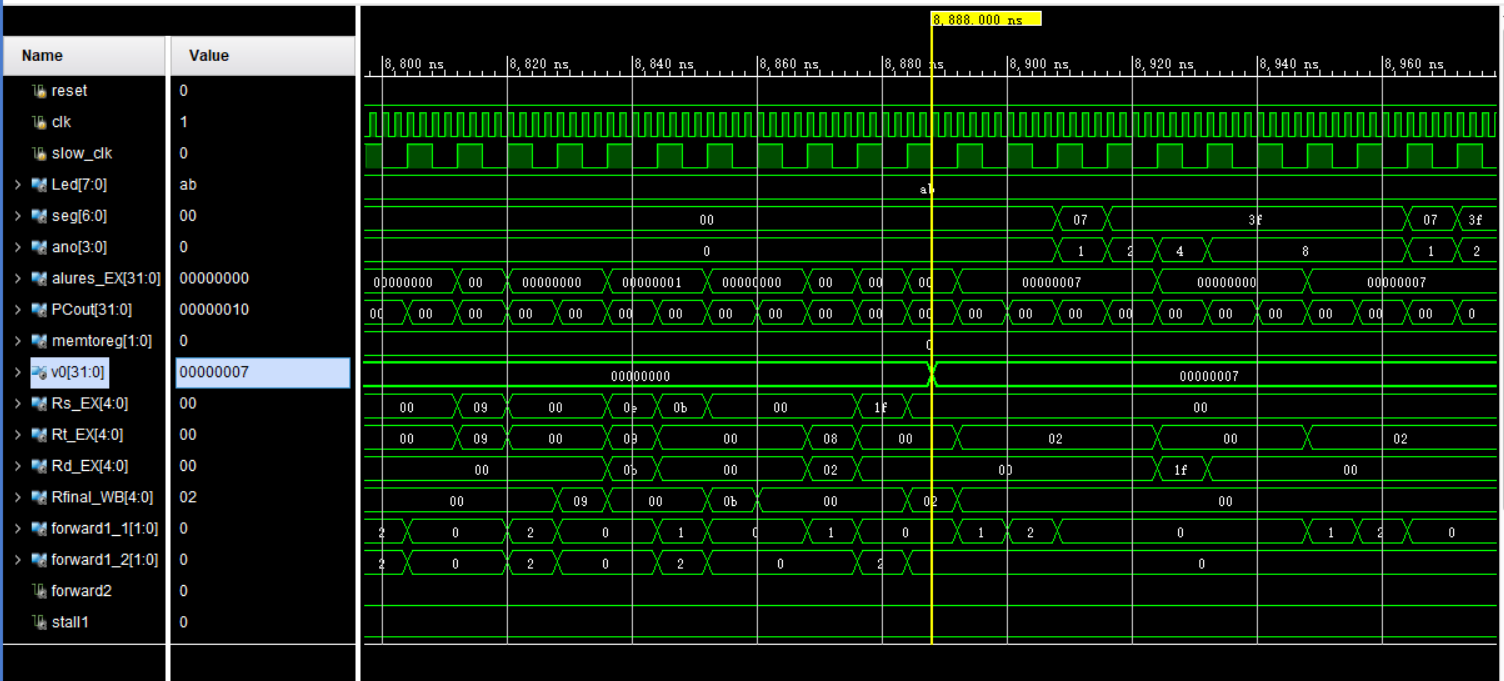
Pattern: can

仿真运行过程如下：



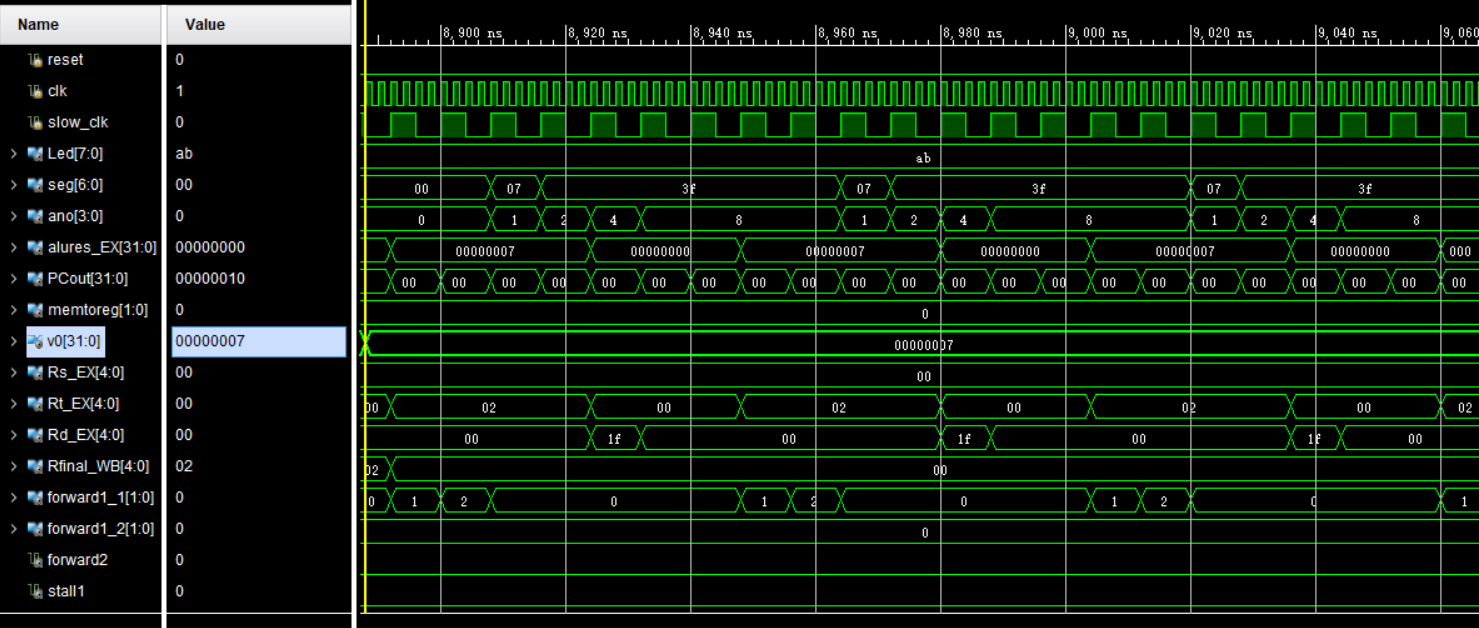
在结果尚未计算完成时，seg和ano的值不会发生改变。（seg对应BCD所显示的数字）

在程序运行完成并将bruteforce返回值7写入v0寄存器后，进入软件扫描显示BCD管阶段。



此时seg和ano的数值发生改变。

最后程序循环地将数据写入外设的地址。



可以看到，最终BCD对应值显示在七段数码管上也确实是0007，因此仿真结果正确。

# 五、硬件调试情况

测试语句：

Str: can a canner can a can like a canner can a can?

Pattern: can

FPGA板很快通过扫描的方式显示“0007”。

可见硬件测试成功，详见视频。

# 六、心得体会

本次实验花费了较长的时间，但整体上说较为顺利。可能因为是在单周期和多周期的设计基础上进行流水线的设计，有了不少经验，也已经踩过了很多坑，所以没有特别破防的情况出现。

写代码的过程大多数时候是枯燥而机械的工作，但一定不可以掉以轻心，可能当初1bit的错误在日后你需要花费1h的时间把它找到。ALU、controller等模块是事先已经写好的，我再把几个pipeline大寄存器写好，过程当中体验了控制信号的逐级传递，然后对不同类型的指令分别进行连线并补充pipeline寄存器中的内容。待处理器基本的电路逻辑实现后，再添加冒险控制单元和转发单元。冒险和转发当初学的时候觉得简单，但要自己真正设计细节的时候发现很繁琐，有一些判断条件很容易漏掉，最后真正写出来的时候又发现语句其实也很简单。

Debug本身也花了很多时间。通过前仿真逐条指令比对，找到执行的指令不正确的再取逐个控制信号比对排查问题。其中印象比较深刻的bug就是先写后读，当时思考了很久为什么数据写不进去。在盯着波形debug的过程中也更深刻理解了流水线中指令的逐级传递与运行，还理解了命名规则的重要性。

最后再添加了外设，同时意识到字符串匹配不能简单靠lw实现。幸好verilog语言的模块化设计让我得以通过简单修改DataMemory的储存结构就可以让代码成功运行。录入代码和初始化数据本身也是非常枯燥的过程。但未来的科研和学术想必也有很多这种枯燥乏味的搬砖工作吧。

需要感谢老师和助教们的悉心指导，让我掌握了处理器基本的工作流程，更是学会了自己去写一个处理器。原本以为的复杂无比的内容自己竟然也可以在一个学期内掌握，期间有很多破防的瞬间，但最后处理器运行起来时，一切都显得值得了。多谢老师、助教还有同学们的帮助。

# 附录

## 附录一 指令格式表



* lb 0x20 rs rt offset
* esw 0x3F rs rt 0 0 ano

## 附录二 指令说明表

Rs,

* lb ： R[rt] = M[R[rs]+SignExtImm] 但只1 Byte的数据，不进行字对齐。
* esw ： M[32'h40000010] = Digi\_decoder(rs, ano); //生成特定的地址