# 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

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

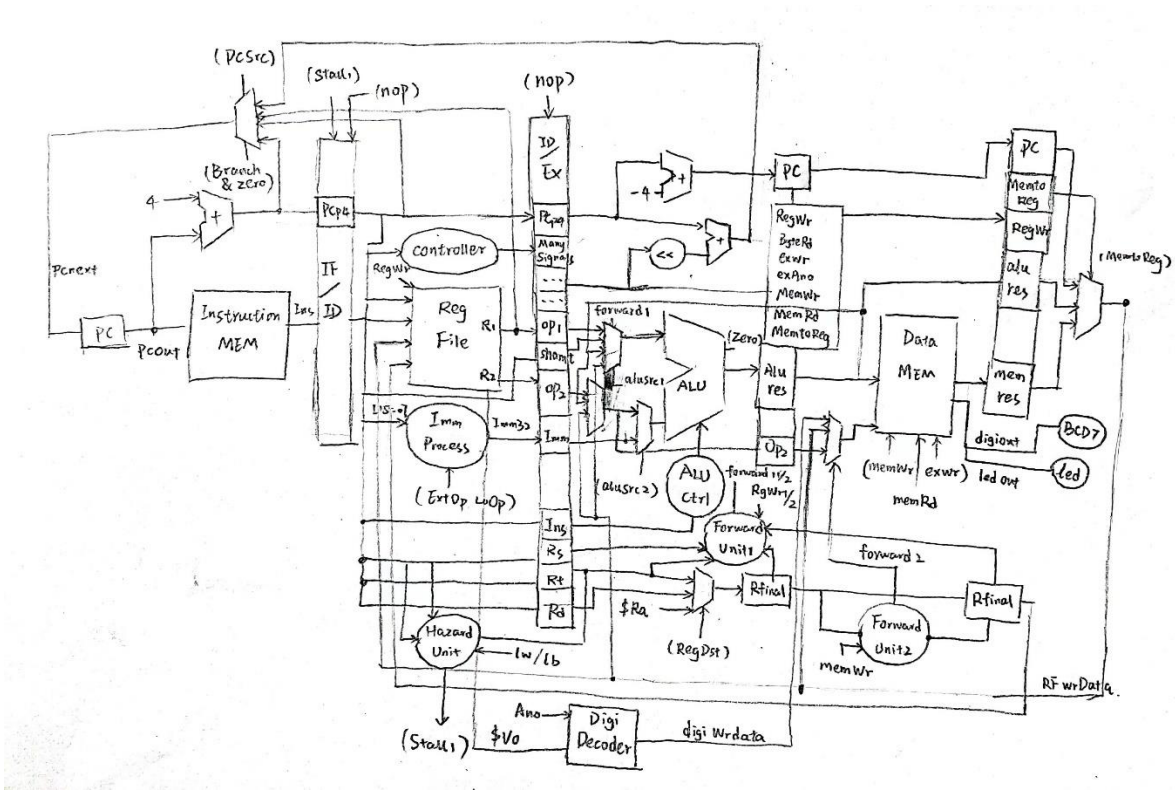
本次实验实现的指令集为：

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。

i) 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_E
X[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：**

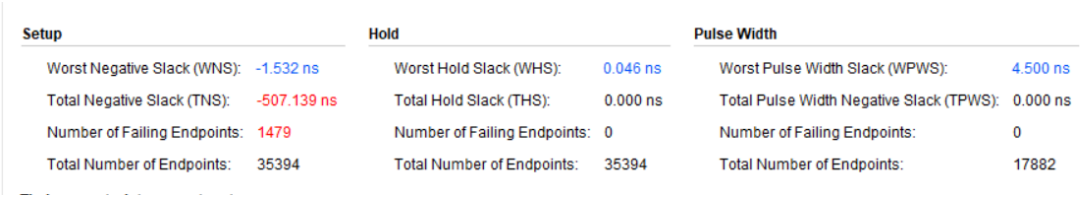
```
 assign Read_data =  MemRead?
{RAM_data[base_Address],RAM_data[base_Address+2'b01],RAM_data[base_Address+
2'b10],RAM_data[base_Address+2'b11]}:
 ByteRead? {24'h000000,RAM_data[Address[9:0]]}: 32'h00000000;

// ----------------------------------------------------
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：

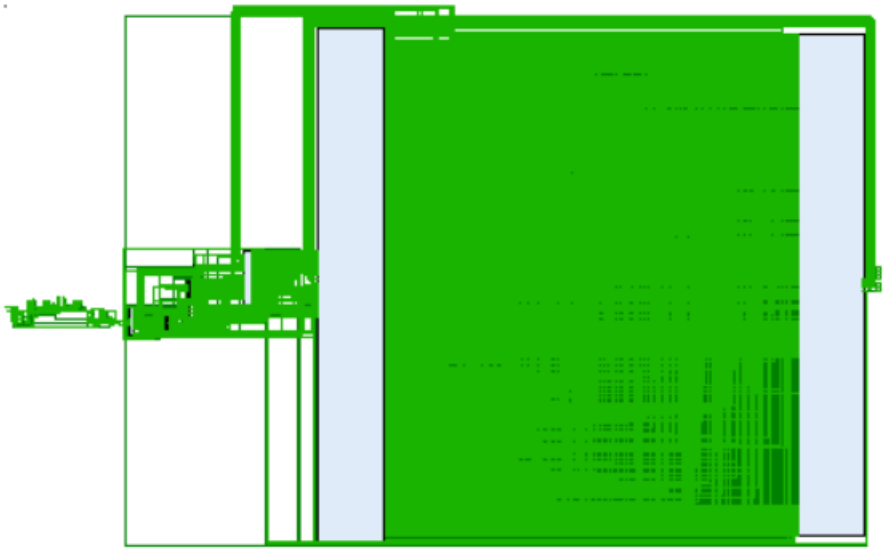



时序裕量为负值，说明时钟周期设置偏短。

最短时钟周期 Tmin = 11.53ns

最高时钟频率 fmax = 86.7MHz

**Schematic：**

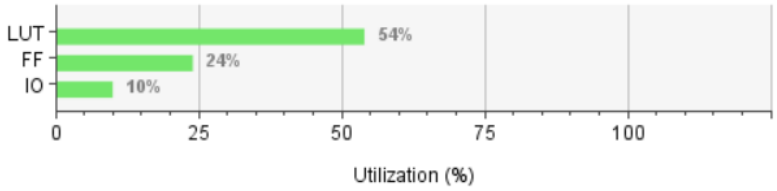


## （二）逻辑资源占用情况

处理器的逻辑资源占用情况如下：

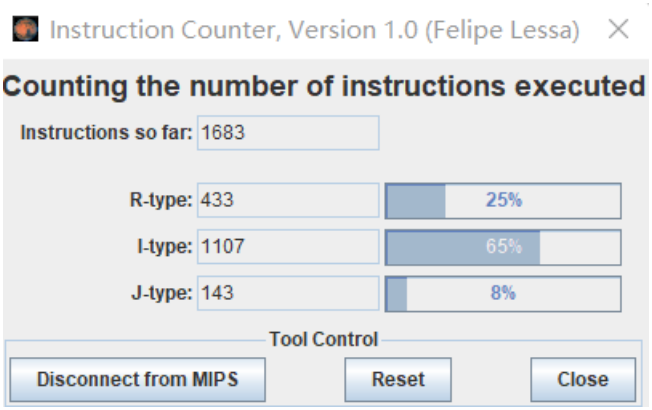
| Name | Slice LUTs (20800) | Slice Registers (41600) | F7 Muxes (16300) | F8 Muxes (8150) | Slice (8150) | LUT as Logic (20800) | LUT Flip Flop Pairs (20800) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|---|
| CPU | 11309 | 9883 | 2425 | 1048 | 4800 | 11309 | 4104 | 21 | 2 |
| AluC (ALUControl) | 13 | 5 | 0 | 0 | 16 | 13 | 0 | 0 | 0 |
| DMem (DataMemory_n... | 4376 | 8211 | 2168 | 1048 | 3855 | 4376 | 0 | 0 | 0 |
| EXMEM (RegEX_MEM) | 5212 | 285 | 0 | 0 | 2213 | 5212 | 239 | 0 | 0 |
| IDEX (RegID_EX) | 868 | 156 | 1 | 0 | 335 | 868 | 32 | 0 | 0 |
| IFID (RegIF_ID) | 65 | 65 | 0 | 0 | 47 | 65 | 0 | 0 | 0 |
| MEMWB (RegMEM_WB) | 123 | 104 | 0 | 0 | 86 | 123 | 1 | 0 | 0 |
| regPC (RegTemp) | 35 | 32 | 0 | 0 | 30 | 35 | 0 | 0 | 0 |
| Rfile (RegisterFile) | 576 | 992 | 256 | 0 | 587 | 576 | 0 | 0 | 0 |

| Resource | Utilization | Available | Utilization % |
|---|---|---|---|
| LUT | 11309 | 20800 | 54.37 |
| FF | 9883 | 41600 | 23.76 |
| IO | 21 | 210 | 10.00 |

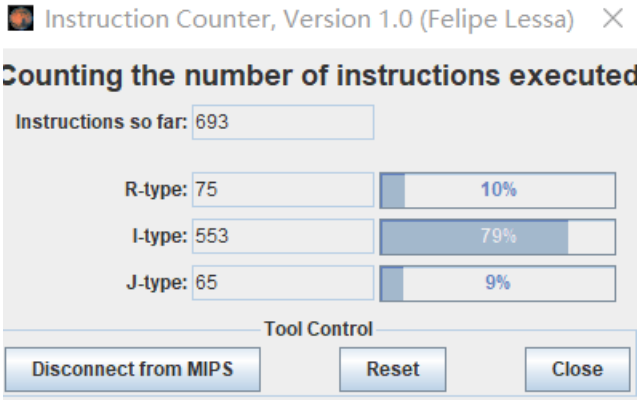


# （三）处理器 CPI 计算

**指令数：**



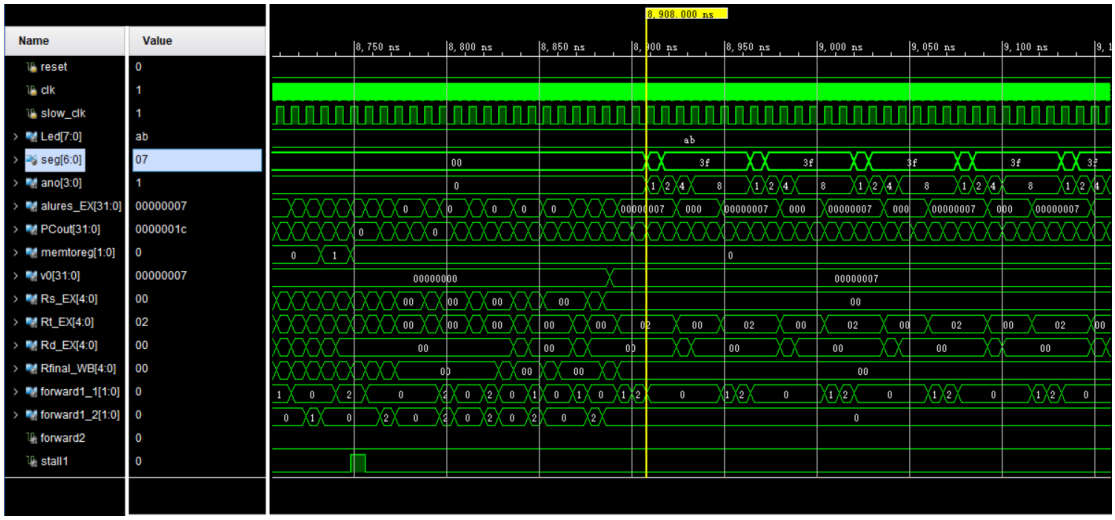

程序指令数计算：1683 ，考虑到前期使用了较多指令进行文件读取，而在仿真时文件读取直接通过初始化完成，故需要减去一部分指令。
注释掉 jal 到字符串匹配的指令后再计算指令数如下：




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

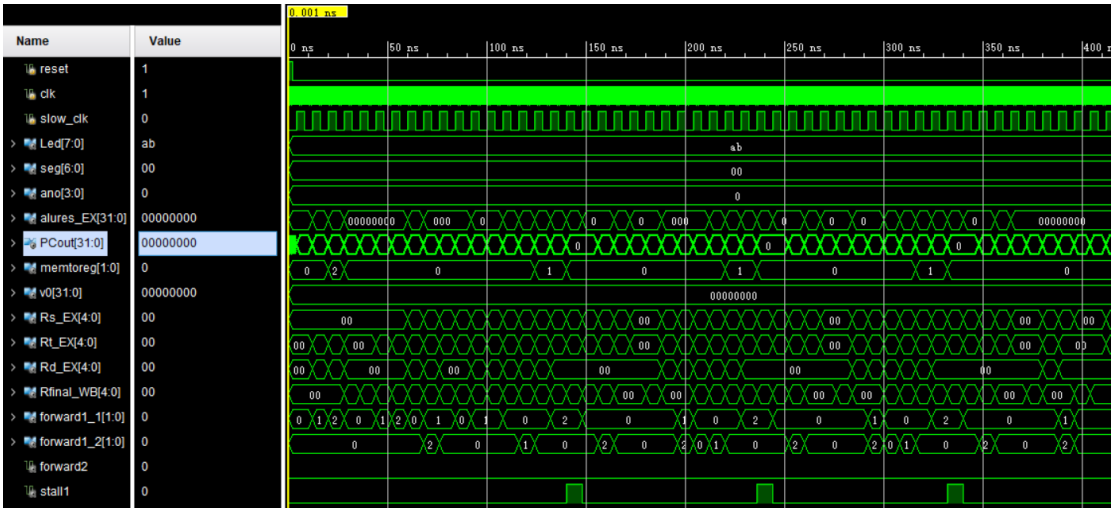
仿真测试直接使用最终要跑的 brute-force 字符串匹配代码。再字符串匹配代码中存在大量的数据冒险和控制冒险，如果能得到正确的结果，可以认为处理器工作正常。

测试语句：

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

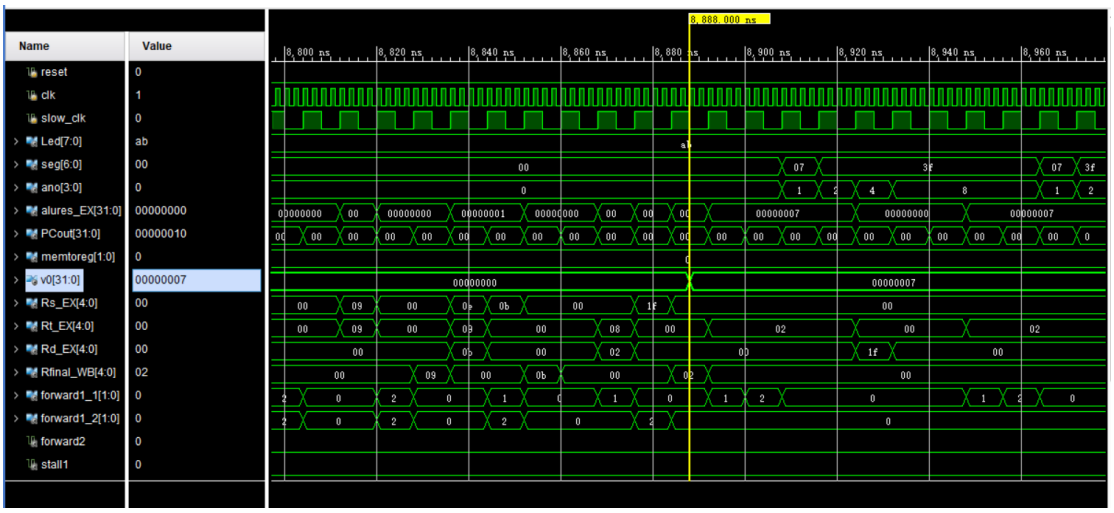
Pattern: can

仿真运行过程如下：



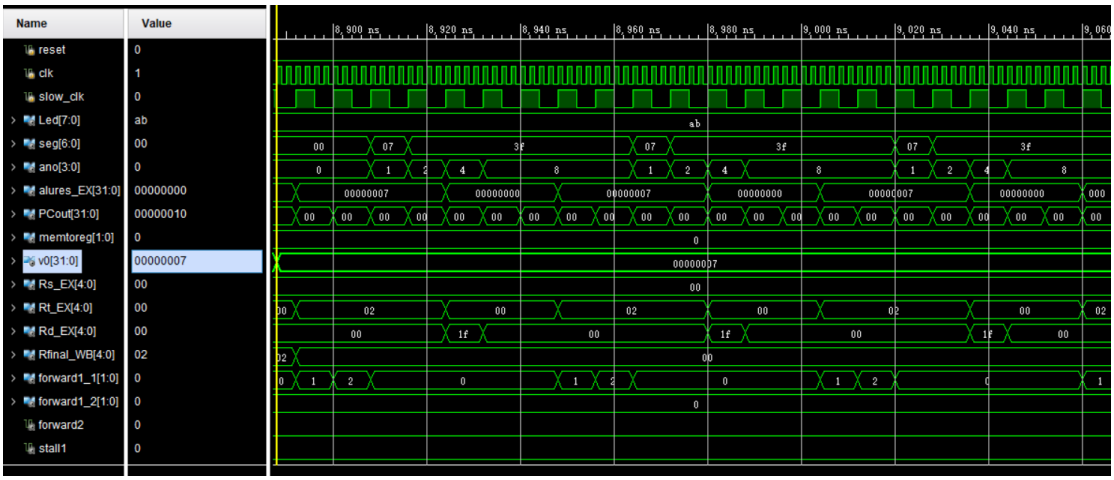

在结果尚未计算完成时，seg 和 ano 的值不会发生改变。（seg 对应 BCD 所显示的数字）

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

| 指令格式表 | | | | | | |
|---|---|---|---|---|---|---|
| | OpCode [5:0] | Rs [4:0] | Rt [4:0] | Rd [4:0] | Shamt [4:0] | Funct [5:0] |
| | OpCode [5:0] | Rs [4:0] | Rt [4:0] | imm/offset [15:0] | | |
| | OpCode [5:0] | target [25:0] | | | | |
| R 型算术指令 | | | | | | |
| add | 0 | rs | rt | rd | 0 | 0x20 |
| addu | 0 | rs | rt | rd | 1 | 0x21 |
| sub | 0 | rs | rt | rd | 2 | 0x22 |
| subu | 0 | rs | rt | rd | 3 | 0x23 |
| and | 0 | rs | rt | rd | 0 | 0x24 |
| or | 0 | rs | rt | rd | 1 | 0x25 |
| xor | 0 | rs | rt | rd | 2 | 0x26 |
| nor | 0 | rs | rt | rd | 3 | 0x27 |
| slt | 0 | rs | rt | rd | 0 | 0x2a |
| sltu | 0 | rs | rt | rd | 0 | 0x2b |
| sll | 0 | 0 | rt | rd | shamt | 0 |
| srl | 0 | 0 | rt | rd | shamt | 0x02 |
| sra | 0 | 0 | rt | rd | shamt | 0x03 |
| I 型算术指令 | | | | | | |
| lui | 0x0f | 0 | rt | imm | | |
| addi | 0x08 | rs | rt | imm | | |
| addiu | 0x09 | rs | rt | imm | | |
| andi | 0x0c | rs | rt | imm | | |
| sltiu | 0x0b | rs | rt | imm | | |
| 存取指令 | | | | | | |
| lw | 0x23 | rs | rt | offset | | |
| sw | 0x2b | rs | rt | offset | | |
| 分支指令 | | | | | | |
| beq | 0x04 | rs | rt | offset | | |
| bne | 0x05 | rs | rt | offset | | |
| blez | 0x06 | rs | 0 | offset | | |
| bgtz | 0x07 | rs | 0 | offset | | |
| bltz | 0x01 | rs | 0 | offset | | |
| 跳转指令 | | | | | | |
| j | 0x02 | target | | | | |
| jal | 0x03 | target | | | | |
| jr | 0 | rs | 0 | | | 0x08 |
| jalr | 0 | rs | 0 | rd | 0 | 0x09 |

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

# 附录二 指令说明表

| 指令说明表 | |
|---|---|
| 指令 | 说明 |
| R 型算术指令 | |
| add | R[rd] = R[rs] + R[rt] |
| addu | R[rd] = R[rs] + R[rt] |
| sub | R[rd] = R[rs] - R[rt] |
| subu | R[rd] = R[rs] - R[rt] |
| and | R[rd] = R[rs] & R[rt] |
| or | R[rd] = R[rs] \| R[rt] |
| xor | R[rd] = R[rs] ^ R[rt] |
| nor | R[rd] = ~(R[rs] \| R[rt]) |
| slt | R[rd] = (R[rs]± < R[rt]±) ? 1 : 0 |
| sltu | R[rd] = (R[rs]ø < R[rt]ø) ? 1 : 0 |
| sll | R[rd] = R[rt] << shamt |
| srl | R[rd] = R[rt]ø >> shamt |
| sra | R[rd] = R[rt]± >> shamt |
| I 型算术指令 | |
| lui | R[rt] = {imm, 16'b0} |
| addi | R[rt] = R[rs] + SignExtImm |
| addiu | R[rt] = R[rs] + SignExtImm |
| andi | R[rt] = R[rs] & ZeroExtImm |
| sltiu | R[rt] = (R[rs]ø < SignExtImmø) ? 1 : 0 |
| 存取指令 | |
| lw | R[rt] = M[R[rs] + SignExtImm] |
| sw | M[R[rs] + SignExtImm] = R[rt] |
| 分支指令 | |
| beq | if (R[rs] == R[rt]) PC += 4 + (offset << 2) |
| bne | if (R[rs] != R[rt]) PC += 4 + (offset << 3) |
| blez | if (R[rs] <= 0) PC += 4 + (offset << 4) |
| bgtz | if (R[rs] > 0) PC += 4 + (offset << 5) |
| bltz | if (R[rs] < 0) PC += 4 + (offset << 6) |
| 跳转指令 | |
| j | PC = {PC[31:28], target << 2} |
| jal | R[31] = PC + 4; PC = {PC[31:28], target << 2} |
| jr | PC = R[rs] |
| jalr | R[31] = PC + 4; PC = R[rs] |

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