Vivado 实验三：单周期 CPU 设计实验

姓名： 李泽桐 学号： 18340100

一、实验目的

1. 掌握单周期 CPU 的设计方法；
2. 通过实践加深对单周期各模块及其整体工作的原理的理解。

二、实验要求

1. 在 vivado 上设计单周期 CPU；
2. 利用设计的单周期 CPU 跑冒泡排序；
3. 将设计的 CPU 烧到开发板上，实现输入数据排序并输出显示。

三、实验内容

在 vivado 上采用 Verilog 硬件描述语言，设计实现单周期 CPU 各模块并整合形成完整的单周期 CPU，并自行编写适用于自己设计的指令集的冒泡排序程序使其在自己设计的单周期 CPU 上运行成功，还有自行编写开发板输入输出数据所需要的模块，将单周期 CPU 与输入输出数据模块烧录到开发板上跑冒泡排序程序。

四、实验原理

单周期 CPU，顾名思义，是指在一个时钟周期内完成一整条指令，即从取指、译指到执行，要求在一个周期内完成。

在一个时钟周期内，CPU 各个模块所需要的功能由这个周期内所要执行的指令确定，因此，需要控制电路来针对每一条指令输出对应的控制（选择）信号，使每个模块在当前周期执行相应的功能，这样单周期 CPU 就可以运行了。

事实上，就 CPU 而言，基本上可以分为取指、译指和执行（具体又有运算、访存和写回等动作）这几个环节，这样在 CPU 设计中必不可少的就有程序计数器（Program Counter）、存储器（Memory）、寄存器堆（Register File）、算术逻辑单元（Arithmetic Logical Unit）还有控制电路（Control Circuit）。在以上的基础之上，由于单周期 CPU 中一条指令只在一个周期内完成，因此存储器（Memory）又需要裂解为指令存储器（Instruction Memory）和数据存储器（Data Memory）。为了能够实现每个模块的功能选择（或者输入选择），还需要一些额外的多路选择器（Multiplexer），并且，由于单周期 CPU 每个指令的时钟周期只有一个，因此，还需要额外的加法器（Adder）帮助实现 PC+4 等其他类似的操作。

因此，完成以上模块的构建及整合即可完成单周期 CPU 的设计。

五、实验过程与结果

（一）设计计划

设计单周期 CPU 的步骤如下：

1. 明确自己想要实现的所有指令，做成指令集；
2. 自顶向下，先画出大致的单周期的数据通路图（按上面提及的模块先粗略构建）；
3. 根据指令集以及粗略的数据通路图，初步设计好各个模块的接口；
4. 根据各个模块的接口以及各条指令运行所需的控制信号整理出控制电路的具体实现；
5. 根据粗略的数据通路图连接好各个模块，完成顶层设计；
6. 测试指令执行情况，及时改错并稍做优化；

注：实验内容的条理性和美观性将影响实验报告的分数。对实验结果是否拍照不作要求，重点在于实验内容的描述和关键代码的解释。

7.用自己设计的指令集中拥有的指令编写冒泡排序程序并转化为机器代码;
8.在CPU上运行程序查看波形，若不成功，需要继续debug;
9.烧板。

（二）设计过程

按照上述设计的计划的步骤逐步进行。

1.指令集的设计

为了完成最终能够跑起冒泡排序程序的目标,以及避免需要将汇编语言转化为机器代码的繁琐,我决定在MIPS指令集中选取最终编写冒泡排序程序需要用到的24条指令，这样一来，也方便使用Mars软件将汇编语言转化为机器代码。我所选取的这24条指令如下:

add	addi	addiu	addu
and	andi	or	ori
slt	slti	sltiu	sltu
sll	srl	sub	subu
beq	bne	j	jal
jr	sw	lw	halt

其中，halt为自行设计的停机指令，以便程序的运行得以终止。

指令的格式与设计与MIPS完全一致，即以上指令中有R型、I型、J型，具体格式可以参考MIPS Reference Data，下图（图1）只是部分:

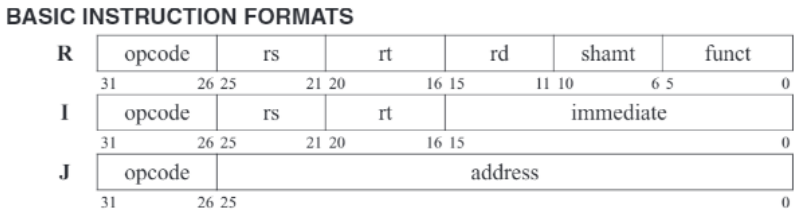


图1

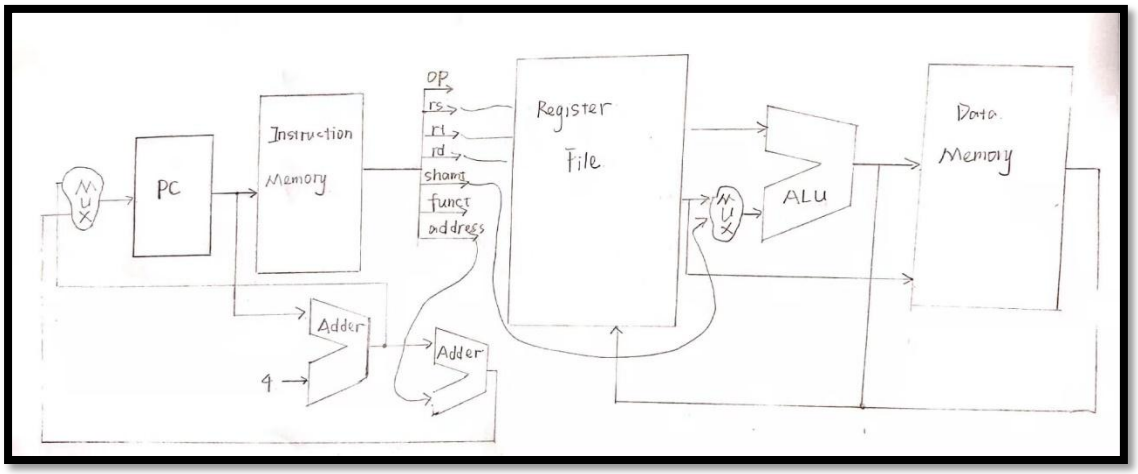
具体的各个指令如下:

指令名	指令具体格式
add	R[rd]=R[rs]+R[rt]
addi	R[rt]=R[rs]+SignExtImm
addiu	R[rt]=R[rs]+SignExtImm
addu	R[rd]=R[rs]+R[rt]
and	R[rd]=R[rs]&R[rt]
andi	R[rt]=R[rs]&ZeroExtImm
or	R[rd]=R[rs]|R[rt]
ori	R[rt]=R[rs]|ZeroExtImm
slt	R[rd]=(R[rs]<R[rt])?1:0
slti	R[rt]=(R[rs]<SignExtImm)?1:0
sltiu	R[rt]=(R[rs]<ZeroExtImm)?1:0
sltu	R[rd]=(R[rs]<R[rt])?1:0
sll	R[rd]=R[rt]<<shamt
srl	R[rd]=R[rt]>>shamt
sub	R[rd]=R[rs]-R[rt]
subu	R[rd]=R[rs]-R[rt]
beq	if(R[rs]==R[rt]) PC=PC+4+BranchAddr

bne	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr
j	PC=JumpAddr
jal	R[31]=PC+8 PC=JumpAddr
jr	PC=R[rs]
sw	M[R[rs]+SignExtImm]=R[rt]
lw	R[rt]=M[R[rs]+SignExtImm]
halt	停机

（续上页表）

2.绘制大概的数据通路



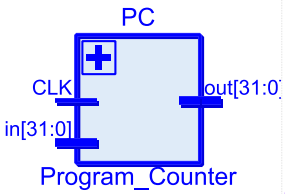
在草稿上粗略画一张数据通路，以便后面开始搭建。

3.设计各个模块及其接口

按照上面所述，我们需要构建的模块有程序计数器（Program Counter）、指令存储器（Instruction Memory）、寄存器堆（Register File）、算术逻辑单元（Arithmetic Logical Unit）、数据存储器（Data Memory）、加法器（Adder）、多路选择器（Multiplexer）。下面逐一构造。

a） 程序计数器

如图 2 所示，Program Counter 的接口有时钟输入 CLK、一个输入且为 32 位的 in、一个输出且为 32 位的 out。Program Counter 主要就是用来输出下一条指令的地址的。在时钟上升沿到来时，Program Counter 会更新其输出的指令地址。在我编写的代码中，有一个较为多余的变量 counter，这是之前编写时的一些随意与不认真造成的，没有做修改，但是之后做多周期 CPU 时代码风格会稍微改善，此处就暂时不做处理了。

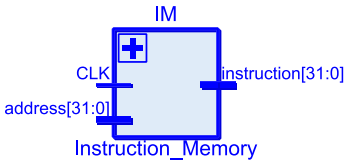



```
module Program_Counter(
    input [31:0] in,
    input CLK,
    output reg [31:0] out
    );
    reg [31:0] counter;

    initial
    begin
        #100 counter<=0;
    end

    always@(posedge CLK)
    begin
        counter<=in;
    end

    always@(*)
    begin
        out<=counter;
    end

endmodule
```

图 2

b） 指令存储器

如图 3 所示，Instruction Memory 的接口有时钟输入 CLK、指令地址输入且为 32 位的 address、指令输出且为 32 位的 instruction_out。这里面我的每一个存储单元的大小是 1Byte，也就是 8bit，这是为了与书中 MIPS CPU 的设计保持一致，不另起炉灶。在这里也只申请了 0-ffh 这么多的空间，这是受制于后面烧板时出现的资源问题，事实上，这么多的存储单元已经远远满足冒泡排序指令的存储了。

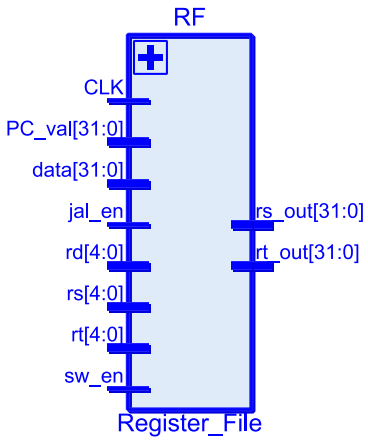



```
module Instruction_Memory(
    input [31:0] address,
    input CLK,
    output reg [31:0] instruction
    );
    reg [7:0] IM[0:8'hff];
    integer i;

    initial
    begin
        #150;
        $readmemh("E:/try/final_test.dat",IM,0,103);
    end

    always@(*)
    begin
        instruction[31:24]=IM[address];
        instruction[23:16]<=IM[address+1];
        instruction[15:8]<=IM[address+2];
        instruction[7:0]<=IM[address+3];
    end

endmodule
```

图 3

c） 寄存器堆

如图 4 所示，Register File 的接口有时钟输入 CLK、Program Counter 的输出值且为 32 位的 PC_val（用于 jal）、数据输入且为 32 位的 data、判断是否为 jal 指令的跳转 jal_en（jal enable）、对应 rd 寄存器的编号输入且为 5 位的 rd、对应 rs 寄存器的编号输入且为 5 位的 rs、对应 rt 寄存器的编号输入且为 5 位的 rt、允许写入寄存器堆的信号 sw_en、rs 寄存器中的数据输出且为 32 位的 rs_out、rt 寄存器中的数据输出且为 32 位的 rt_out。当然，在这里面的 rs、rt、rd 寄存器编号不是一定是指令中对应的 rs、rt、rd，这里只是为了命名的方便而已。在这个设计当中，我定义了 32 个寄存器，分别对应于 MIPS 中所设计的寄存器，且只有当时钟上升沿到来以及允许输入信号 sw_en 为 1 时才允许将 data 输进寄存器堆。在这里，PC_val 与 sw_en 其实都是为指令 jal 服务，由于事先设计并未考虑到 jal 特殊性，在后来为了实现 jal 只能修改，所以是临时添加上去的，这样“打补丁”的行为不可取，但现在暂时未作改动，可以在多周期 CPU 制作时吸取经验加以改进。

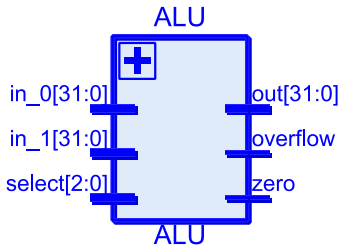



```
module Register_File(
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd,
    input [31:0] data,
    input sw_en,
    input [31:0] PC_val,
    input jal_en,
    input CLK,
    output reg [31:0] rs_out,
    output reg [31:0] rt_out
    );
    reg [31:0] register[0:31];
    integer i;

    initial
    begin
        for(i=0;i<32;i=i+1)
            register[i]<=0;
    end

    always@(posedge CLK)
    begin
        if(sw_en==1)
            register[rd]<=data;
        if(jal_en==1)
            register[31]<=PC_val;
    end

    always@(*)
    begin
        rs_out<=register[rs];
        rt_out<=register[rt];
    end

endmodule
```

图 4

d） 算术逻辑单元

如图 5 所示，Arithmetic logical unit 的接口有第一个输入且为 32 位的 in_0、第二个输入且为 32 位的 in_1、ALU 功能选择信号且为 3 位的 select、运算结果输出且为 32 位的 out、溢出标志位 overflow、判断运算结果是否为 0 的位 zero。事实上，这里的 ALU 大部分功能都是依赖于 Verilog 语言本身，也就是说，这里的加、减、与、或、有符号比较小于置 1、无符号比较小于置 1、左移、右移这 8 个 ALU 功能全部是直接用 Verilog 语言实现而没有自己设计门电路（当然该课程实验后面有要求实现加法器和乘法器，也算弥补了这一缺陷）。正如刚才所列举的，这个 ALU 总共有 8 个功能选择，故需要 3 位的 select 作为选择信号。其中加减乘除以及左右移位的功能就不再赘述，其中有有符号比较置 1 和无符号比较置 1，分别对应于指令 slt 和 sltu。除了这些运算方面的功能之外，还有 0 标志位和溢出标志位，作用也是相当重要。

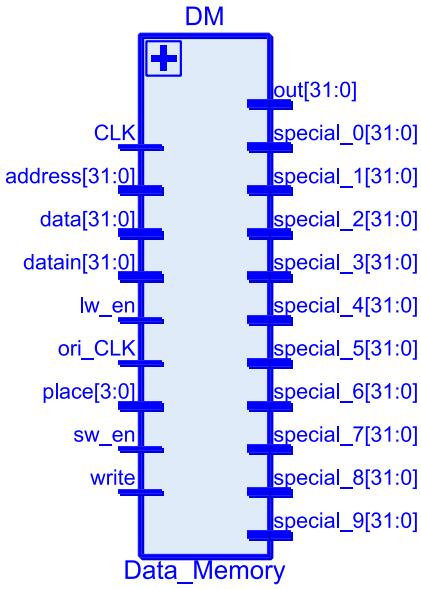


```
module ALU(
    input [31:0] in_0,
    input [31:0] in_1,
    input [2:0] select,
    output reg [31:0] out,
    output reg zero,
    output reg overflow
    );
    reg carry1,carry2;
    reg [31:0] temp1;
    reg [30:0] temp2;

    always@(*)
    begin
        case(select)
            3'b000:out<=in_0+in_1;
            3'b001:out<=in_0-in_1;
            3'b010:out<=in_0&in_1;
            3'b011:out<=in_0|in_1;
            3'b100:
                begin
                    if(in_0[31:31]==0&&in_1[31:31]==0)
                        out<=(in_0<in_1)?1:0;
                    else if(in_0[31:31]==1&&in_1[31:31]==1)
                        out<=(in_0<in_1)?1:0;
                    else if(in_0[31:31]==0&&in_1[31:31]==1)
                        out<=0;
                    else if(in_0[31:31]==1&&in_1[31:31]==0)
                        out<=1;
                end
            3'b101:out<=(in_0<in_1)?1:0;
            3'b110:out<=in_1<<in_0;
            3'b111:out<=in_1>>in_0;
            default:out<=0;
        endcase
        if(select==3'b000)
    begin
        {carry1,temp1}<=in_0+in_1;
        {carry2,temp2}<=in_0[30:0]+in_1[30:0];
        overflow<=((carry1==1&&carry2==0)||(carry1==0&&carry2==1))?1:0;
    end
    else if(select==3'b001)
    begin

        {carry1,temp1}<=in_0-in_1;
        {carry2,temp2}<=in_0[30:0]-in_1[30:0];
        overflow<=((carry1==1&&carry2==0)||(carry1==0&&carry2==1))?1:0;
    end
    zero<=(in_0-in_1==0)?1:0;
end

endmodule
```

图 5

e）数据存储器

如图 6 所示,Data Memory 的接口有时钟输入 CLK、数据地址输入且为 32 位的 address、CPU 写入数据且为 32 位的 data、开发板的开关输入数据且为 32 位的 datain、允许数据读出的控制信号位 lw_en、原始时钟输入 ori_CLK（original CLK 意指原始时钟输入，是前面 CLK 分频之前的 100MHz 时钟，后面会有解释，这是数码管显示使用到的引脚）、开发板数码管显示数据在存储器的地址且为 4 位的 place、允许 CPU 数据写入的控制信号位 sw_en、开发板写入存储器的确认按钮 write、CPU 请求地址的数据输出且为 32 位的 out、与开发板显示存储器 10 个数（10 个存储地址的数据）的引脚即 10 个 32 位的输出引脚（special_0-special_9）。事实上，这不是一块简单的数据存储器，由于后续需要从开发板上获取数据的输入并要输出到数码管进行显示，因此，ori_CLK 的引入是为了解决开发板的读入和读出数据即显示的问题，而引脚 datain 即是开发板读入数据的引脚，place 是开发板选择查看或修改的存储器地址，write 是开发板确定输入的按钮，special_0-special_9 是数据输出到数码管的引脚。由于这一部分跟 CPU 的设计关系不大，此处不做详细介绍。对于数据存储器本身，当时钟上升沿到来时，data 引脚数据会被写入数据存储器。




```
module Data_Memory(
    input [31:0] address,
    input [31:0] data,
    input sw_en,
    input lw_en,
    input ori_CLK,
    input CLK,
    output [31:0] out,
    output [31:0] special_0,//特例
    output [31:0] special_1,//特例
    output [31:0] special_2,//特例
    output [31:0] special_3,//特例
    output [31:0] special_4,//特例
    output [31:0] special_5,//特例
    output [31:0] special_6,//特例
    output [31:0] special_7,//特例
    output [31:0] special_8,//特例
    output [31:0] special_9,//特例
    input [31:0] datain,
    input [3:0] place,
    input write
    );
    reg [7:0] DM[0:8'hff];
    reg addr=0;
    reg i=0;
    reg j=0;
    reg k=0;
    reg l=0;
    reg detect_write=0;
    reg detect_CLK=0;

    initial
    begin
        $readmemh("E:/try/final_sample.dat",DM,0,39);
    end

    assign out[31:24]=DM[address];
    assign out[23:16]=DM[address+1];
    assign out[15:8]=DM[address+2];
    assign out[7:0]=DM[address+3];

    assign special_0={DM[addr+0],DM[addr+1],DM[addr+2],DM[addr+3]};
    assign special_1={DM[addr+4],DM[addr+5],DM[addr+6],DM[addr+7]};
    assign special_2={DM[addr+8],DM[addr+9],DM[addr+10],DM[addr+11]};
    assign special_3={DM[addr+12],DM[addr+13],DM[addr+14],DM[addr+15]};
    assign special_4={DM[addr+16],DM[addr+17],DM[addr+18],DM[addr+19]};
    assign special_5={DM[addr+20],DM[addr+21],DM[addr+22],DM[addr+23]};
    assign special_6={DM[addr+24],DM[addr+25],DM[addr+26],DM[addr+27]};
    assign special_7={DM[addr+28],DM[addr+29],DM[addr+30],DM[addr+31]};
    assign special_8={DM[addr+32],DM[addr+33],DM[addr+34],DM[addr+35]};
    assign special_9={DM[addr+36],DM[addr+37],DM[addr+38],DM[addr+39]};

endmodule
```

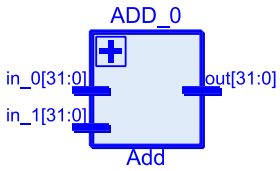
```
always@(posedge ori_CLK)
begin
    i<=write;
    detect_write<=(~write)&i;
end

always@(posedge ori_CLK)
begin
    j<=CLK;
    k<=j;
    detect_CLK<=(~k)&j;
end

always@(posedge ori_CLK)
begin
    if(detect_CLK==1)
    begin
        if(sw_en==1)
        begin
            DM[address]<=data[31:24];
            DM[address+1]<=data[23:16];
            DM[address+2]<=data[15:8];
            DM[address+3]<=data[7:0];
        end
    end
    else if(detect_write==1)
    begin
        DM[place*4]<=datain[31:24];
        DM[place*4+1]<=datain[23:16];
        DM[place*4+2]<=datain[15:8];
        DM[place*4+3]<=datain[7:0];
    end
end
```

图 6

f） 加法器

如图 7 所示,Adder 的接口有第一个输入且为 32 位的 in_0、第二个输入且为 32 位的 in_1、

结果输出且为 32 位的 out。实际上，Adder 只是 ALU 的一个功能而已，受制于单周期，CPU 不能在一个时钟周期内既执行指令要求的运算，又做 PC+4 之类的计算，因此 Adder 可以分担 ALU 一部分工作，从而使单周期 CPU 得以运行。

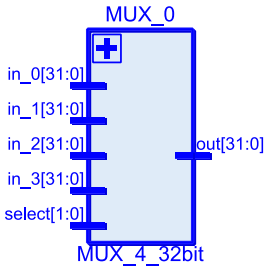



```
module Add(
    input [31:0] in_0,
    input [31:0] in_1,
    output reg [31:0] out
    );

    always@(*)
    begin
        out<=in_0+in_1;
    end

endmodule
```

图 7

g） 多路选择器

如图 8 所示，Multiplexer 的接口有 4 个输入、一个输出还有一个选择信号。事实上，在这个单周期 CPU 中需要的多路选择器不止一个，也不止 4 选 1 这一种多路选择器，其实还有 2 选 1 等等，这里仅展示 4 选 1 作为代表。




```
module MUX_4_32bit(
    input [31:0] in_0,
    input [31:0] in_1,
    input [31:0] in_2,
    input [31:0] in_3,
    input [1:0] select,
    output reg [31:0] out
);

    always@(*)
    begin
        case(select)
            2'b00:out<=in_0;
            2'b01:out<=in_1;
            2'b10:out<=in_2;
            2'b11:out<=in_3;
            default:out<=0;
        endcase
    end

endmodule
```

图 8

到此，各模块及其接口就设计好了。

4. 控制电路的实现

这一部分，照理来说是应该需要写真值表来完成信号的构建，但是由于在 Verilog 语言中，自行构建门电路过于繁琐，且有 if..else..语句可以使用，控制电路的构建就转变为了通过判断该条指令来得到具体的控制信号，因此，我们可以列出指令与控制信号的对应的表格如下：

指令名	类型	OPCODE	FUNCT	PC select	rd selest	fir input select	sec input select	ALU func select	sign select	reg data select	reg sw en	DM sw en	DM lw en	jalen
add	R	00h	20h	00b	0b	0b	0b	000b	1b	0b	1b	0b	0b	0b
addi	I	08h		00b	1b	0b	1b	000b	0b	0b	1b	0b	0b	0b
addiu	I	09h		00b	1b	0b	1b	000b	1b	0b	1b	0b	0b	0b
addu	R	00h	21h	00b	0b	0b	0b	000b	1b	0b	1b	0b	0b	0b
and	R	00h	24h	00b	0b	0b	0b	010b	1b	0b	1b	0b	0b	0b
andi	I	0ch		00b	1b	0b	1b	010b	1b	0b	1b	0b	0b	0b
or	R	00h	25h	00b	1b	0b	0b	011b	1b	0b	1b	0b	0b	0b
ori	I	0dh		00b	1b	0b	1b	011b	1b	0b	1b	0b	0b	0b
slt	R	00h	2ah	00b	0b	0b	0b	100b	1b	0b	1b	0b	0b	0b
slti	I	0ah		00b	1b	0b	1b	100b	0b	0b	1b	0b	0b	0b
sltiu	I	0bh		00b	1b	0b	1b	101b	1b	0b	1b	0b	0b	0b
sltu	R	00h	2bh	00b	0b	0b	0b	101b	1b	0b	1b	0b	0b	0b
sll	R	00h	00h	00b	0b	1b	0b	110b	1b	0b	1b	0b	0b	0b
srl	R	00h	02h	00b	0b	1b	0b	111b	1b	0b	1b	0b	0b	0b
sub	R	00h	22h	00b	0b	0b	0b	001b	1b	0b	1b	0b	0b	0b
subu	R	00h	23h	00b	0b	0b	0b	001b	1b	0b	1b	0b	0b	0b
beq	I	04h		01b(zero=1)	1b	0b	0b	001b	0b	0b	0b	0b	0b	0b
bne	I	05h		01b(zero=0)	1b	0b	0b	001b	0b	0b	0b	0b	0b	0b
j	J	02h		10b	1b	0b	0b	000b	1b	0b	0b	0b	0b	0b
jal	J	03h		10b	1b	0b	0b	000b	1b	0b	0b	0b	0b	1b
jr	R	00h	08h	11b	1b	0b	0b	000b	1b	0b	0b	0b	0b	0b
sw	I	2bh		00b	1b	0b	1b	000b	1b	0b	0b	1b	0b	0b
lw	I	23h		00b	1b	0b	1b	000b	1b	1b	1b	0b	1b	0b
halt	自创	ffh	ffh											

通过以上表格可以用 if..else..这样的逻辑直接编写出控制电路。

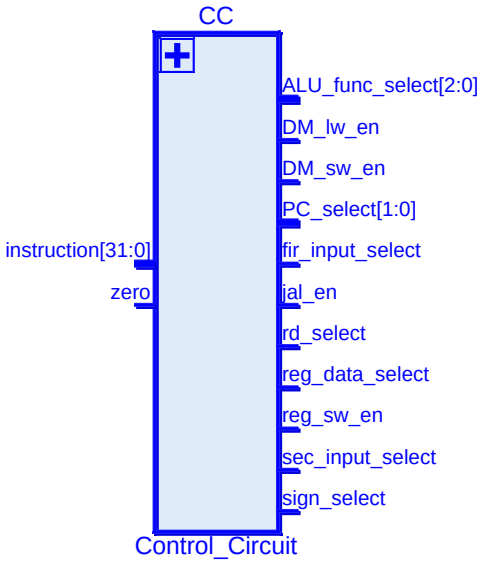
而至于上述的每一个控制信号其名称及作用如下表所示：

控制信号名	状态 0	状态 1
rd_delect	表示指令最后写回寄存器的编号是指令中的 rd 寄存器	表示指令最后写回寄存器的编号是指令中的 rt 寄存器，比如指令 addi、addiu
fir_input_select	表示 ALU 所接受的第一个输入来自前面寄存器堆的 rs 寄存器	表示 ALU 所接受的第一个输入来自指令中的立即数（无符号），比如指令 sll、srl
sec_input_select	表示 ALU 所接受的第二个输入来自前面寄存器堆的 rt 寄存器	表示 ALU 所接受的第二个输入来自指令中的立即数（有无符号还得看 sign_select）
sign_select	表示获得有符号扩展的立即数	表示获得无符号扩展的立即数
reg_data_select	表示写回寄存器堆的数据来源是 ALU 的运算结果	表示写回寄存器堆的数据来源是数据存储器，即指令 lw
reg_sw_en	表示禁止写入寄存器堆的信号	表示允许写入寄存器堆的信号
DM_sw_en	表示禁止写入数据存储器的信号	表示允许写入数据存储器的信号，即指令 sw
DM_lw_en	表示禁止从数据存储器获得数据	表示允许从数据存储器获得数据，即指令 lw
jal_en	表示 jal 跳转信号无效	表示 jal 跳转信号有效

（续表）

控制信号名	各状态意义
PC_select	00 表示 PC+4 作为 PC 输入（对应大多数指令）；01 表示 PC+4+BranchAddr 作为 PC 输入（对应指令 beq、bne 等）；10 表示 JumpAddr 作为 PC 输入（对应指令 j、jal 等）；11 表示 R[rs]作为 PC 输入（对应指令 jr）
ALU_func_select	000 表示 ALU 做加法；001 表示做减法；010 表示做与；011 表示做或；100 表示 slt 和 slti；101 表示 sltu 和 sltiu；110 表示做左移；111 表示做右移

控制电路模块如图 9 所示，Control Circuit 的接口有指令的输入即 32 位的 instruction、零标志 zero（用于判断 beq、bne 等指令所需要的信号）、各个控制信号的输出。




```
//fir_input_select
if(instruction[31:26]==8'h00&&(instruction[5:0]==8'h00||instruction[5:0]==8'h02))
    fir_input_select<=1;
else
    fir_input_select<=0;
//sec_input_select
if(instruction[31:26]==8'h08||instruction[31:26]==8'h09||instruction[31:26]==8'h0c||instruct
    sec_input_select<=1;
else
    sec_input_select<=0;
//ALU_funct_select
if(instruction[31:26]==8'h00&&instruction[5:0]==8'h02)
    ALU_func_select<=8'b111;
else if(instruction[31:26]==8'h00&&instruction[5:0]==8'h00)
    ALU_func_select<=8'b110;
else if((instruction[31:26]==8'h00&&instruction[5:0]==8'h2b)||instruction[31:26]==8'h0b)
    ALU_func_select<=8'b101;
else if((instruction[31:26]==8'h00&&instruction[5:0]==8'h2a)||instruction[31:26]==8'h0a)
    ALU_func_select<=8'b100;
else if((instruction[31:26]==8'h00&&instruction[5:0]==8'h25)||instruction[31:26]==8'h0d)
    ALU_func_select<=8'b011;
else if((instruction[31:26]==8'h00&&instruction[5:0]==8'h24)||instruction[31:26]==8'h0c)
    ALU_func_select<=8'b010;
else if((instruction[31:26]==8'h00&&instruction[5:0]==8'h22)||(instruction[31:26]==8'h00&&i
    ALU_func_select<=8'b001;
else
    ALU_func_select<=8'b000;
```

图 9

由于控制电路的代码冗长且较为单一（只是利用 if..else..语句传送出控制信号），故此处只是给出一两段代码供参考，其余的就不再赘述了。

5. 顶层设计

综合上面各个模块（包括单独讨论的控制电路），再结合之前粗略的数据通路图，即可编写顶层文件，将各个模块联系起来，由于顶层模块也只是连接起各个模块的引脚，因此代码就不再展示，这里只展示最终形成的 RTL 图，如图 10 所示，这其中的电路较为复杂，其实是包含了开发板数据输入和输出所需要包含的一些其他模块，关于这些模块如 Decomposition、Number_7seg、display、Top_for_display 会在后面稍微提及，因为这一块内容与 CPU 的设计关系不大，且有许多小问题，这里就不详细展开了。

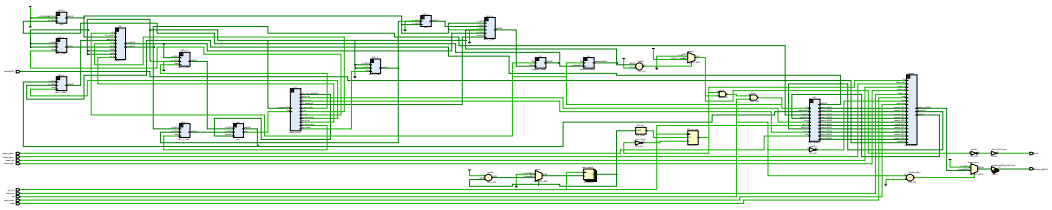


图 10

至此，整体的单周期 CPU 设计完毕，下面可以开始测试。

6. 逐条指令进行测试

这一部分由于当时在做测试时没有做下记录（主要是因为指令数量比较多，且每条指令需要输入不同数据测试，数据量繁杂，不易记录），因而没有数据可以展示。其实像指令 sub 和 subu 在做测试时，要考虑到负数的问题，还有像跳转指令的测试等都不便以数据形式展示，这样就导致在测试时需要多组数据测试，而且在测试的时候不能仅仅说有一组数据成功就盲目跳过（我在这里吃了很多亏，往往有些数据就是在错的设计上跑出对的结果，但实际上还有许多潜在问题）。那下面我们就分别从 R、I、J 这三种类型指令中各挑取一条指令详细讲解一下其在这个单周期 CPU 中的大概运行过程，具体的波形及解释在后面完成冒泡排序后会有提及。

对于 R 型指令，我们以 add rd,rs,rt 为例。首先从 Program Counter 开始，它会输出 add 这一条指令的地址到 Instruction Memory，Instruction Memory 接收到地址后输出 add 这条指令的机器代码，这些代码进入到控制电路，生成了控制信号，有 PC_select=00b（意味着下一个输入 Program Counter 数据的是 PC+4）、rd_select=0b（意味着 ALU 运算结果写回编号 rd 寄存器）、fir_input_select=0b（意味着 ALU 第一个输入是 rs 寄存器）、sec_input_select=0b（意味着 ALU 第二个输入是 rt 寄存器）、ALU_func_select=000b（意味着 ALU 使用加法功能）、sign_select=1b（意味着无符号扩展，但在这里此条指令不起作用）、reg_data_select=0b（意味着写回寄存器的值来源于 ALU 运算结果）、reg_sw_en=1b（意味着允许写回寄存器）、DM_sw_en=0b（意味着禁止写回数据存储器）、DM_lw_en=0b（意味着禁止从数据存储器读出数据）、jal_en=0b（意味着不是执行跳转 jal 信号）。指令除了送到控制电路外，还有依照其具体格式的分解后送到寄存器堆和 ALU，并完成从寄存器堆读出数据在 ALU 完成运算操作，最后 ALU 运算结果写回寄存器堆。

对于 I 型指令，我们以 addi rt,rs,SignExtImm 为例。首先从 Program Counter 开始，它会输出 addi 这一条指令的地址到 Instruction Memory，Instruction Memory 接收到地址后输出 addi 这条指令的机器代码，这些代码进入到控制电路，生成了控制信号，有 PC_select=00b（意味着下一个输入 Program Counter 数据的是 PC+4）、rd_select=1b（意味着 ALU 运算结果写回编号 rs 寄存器）、fir_input_select=0b（意味着 ALU 第一个输入时 rs 寄存器）、sec_input_select=1b（意味着 ALU 的二个输入是有符号扩展的立即数）、ALU_func_select=000b（意味着 ALU 使用加法功能）、sign_select=0b（意味着对前面所说的输入 ALU 第二个输入做有符号扩展）、reg_data_select=0b（意味着写回寄存器的值来源于 ALU 运算结果）、reg_sw_en（意味着允许写回寄存器）、DM_sw_en=0b（意味着禁止写回数据存储器）、DM_lw_en=0b（意味着禁止从数据存储器读出数据）、jal_en=0b（意味着不是执行跳转 jal 信号）。指令除了送到控制电路外，还有依照其具体格式的分解后送到寄存器堆和 ALU 并完成从寄存器堆读出数据在 ALU 完成运算操作，最后 ALU 运算结果写回寄存器堆。

对于 J 型指令，我们以 j JumpAddr 为例。首先从 Program Counter 开始，它会输出 j 这一条指令的地址到 Instruction Memory，Instruction Memory 接收到地址后输出 j 这条指令的机器代码，这些代码进入到控制电路，生成了控制信号，有 PC_select=10b（意味着

下一个输入 Program Counter 数据的是指令中的 JumpAddr)、rd_select=1b（对于此条跳转指令无意义)、fir_input_select=0b（对于此条跳转指令无意义)、sec_input_select=0b（对于此条跳转指令无意义)、ALU_func_select=000b（对于此条跳转指令无意义)、sign_select=1b（对于此条跳转指令无意义)、reg_data_select=0b（对于此条跳转指令无意义)、reg_sw_en=0b（对于此条跳转指令无意义)、DM_sw_en=0b（对于此条跳转指令无意义)、DM_lw_en=0b（对于此条跳转指令无意义)、jal_en=0b（对于此条跳转指令无意义)。对于这条指令，由于是无条件跳转，因此当控制电路识别出来后，CPU 中所有的与写操作有关系的信号都被置为无效，PC 则变为 JumpAddr，实现跳转。

7.汇编写冒泡排序程序并转为机器代码

由于前面的设计完全是按照 MIPS 的指令格式完成,因此我们可以借助 Mars 写汇编再转为机器代码，而不必手动转化为机器代码。由于在之前的实验当中已经编写过该程序，下面只做一些小的修改，编写的冒泡排序如下：

```
next:   la $a0,0
        addi $a1,$zero,10
        jal bubble
        li $v0,10
        syscall

bubble: addi $t0,$zero,1        # 置flag为1
        add $t2,$a1,$zero       # 置num为size
while:  addi $t9,$t0,-1         # $t9=flag-1，判断flag=1
        bne $t9,$zero,A         #
        addi $t0,$zero,0        # 置flag为0
        addi $t1,$zero,0        # 置i为0
for:    addi $t9,$t2,-1         # $t9=num-1，判断i<num-1
        slt $t8,$t1,$t9         #
        beq $t8,$zero,B         #
        sll $t9,$t1,2           # i左移2位，即*4
        add $t8,$t9,$a0         # $t8是arr[i]地址
        lw $t9,0($t8)           # $t9=arr[i]
        lw $t7,4($t8)           # $t7=arr[i+1]
        slt $t6,$t7,$t9         # 判断arr[i]>arr[i+1]
        beq $t6,$zero,C         #
        sw $t7,0($t8)           # arr[i+1]->arr[i]
        sw $t9,4($t8)           # arr[i]->arr[i+1]
        addi $t0,$zero,1        # 置flag为1
C:      addi $t1,$t1,1          # i++
        j for                   #
B:      addi $t2,$t2,-1         # num--
        j while
A:      jr $ra
```

汇编后得到机器代码如下：

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24040000	addiu $4,$0,0x00000000	2: next: la $a0,0
	0x00400004	0x2005000a	addi $5,$0,0x0000000a	3: addi $a1,$zero,10
	0x00400008	0x0c100005	jal 0x00400014	4: jal bubble
	0x0040000c	0x2402000a	addiu $2,$0,0x0000000a	5: li $v0,10
	0x00400010	0x0000000c	syscall	6: syscall
	0x00400014	0x20080001	addi $8,$0,0x00000001	8: bubble: addi $t0,$zero,1 # 置flag为1
	0x00400018	0x00a05020	add $10,$5,$0	9: add $t2,$a1,$zero # 置num为size
	0x0040001c	0x2119ffff	addi $25,$8,0xffffffff	10: while: addi $t9,$t0,-1 # $t9=flag-1，判断flag=1
	0x00400020	0x17200012	bne $25,$0,0x00000012	11: bne $t9,$zero,A #
	0x00400024	0x20080000	addi $8,$0,0x00000000	12: addi $t0,$zero,0 # 置flag为0
	0x00400028	0x20090000	addi $9,$0,0x00000000	13: addi $t1,$zero,0 # 置i为0
	0x0040002c	0x2159ffff	addi $25,$10,0xffffffff	14: for: addi $t9,$t2,-1 # $t9=num-1，判断i<num-1
	0x00400030	0x0139c02a	slt $24,$9,$25	15: slt $t8,$t1,$t9 #
	0x00400034	0x1300000b	beq $24,$0,0x0000000b	16: beq $t8,$zero,B #
	0x00400038	0x0009c880	sll $25,$9,0x00000002	17: sll $t9,$t1,2 # i左移2位，即*4
	0x0040003c	0x0324c020	add $24,$25,$4	18: add $t8,$t9,$a0 # $t8是arr[i]地址
	0x00400040	0x8f190000	lw $25,0x00000000($24)	19: lw $t9,0($t8) # $t9=arr[i]
	0x00400044	0x8f0f0004	lw $15,0x00000004($24)	20: lw $t7,4($t8) # $t7=arr[i+1]
	0x00400048	0x01f9702a	slt $14,$15,$25	21: slt $t6,$t7,$t9 # 判断arr[i]>arr[i+1]
	0x0040004c	0x11c00003	beq $14,$0,0x00000003	22: beq $t6,$zero,C #
	0x00400050	0xaf0f0000	sw $15,0x00000000($24)	23: sw $t7,0($t8) # arr[i+1]->arr[i]
	0x00400054	0xaf190004	sw $25,0x00000004($24)	24: sw $t9,4($t8) # arr[i]->arr[i+1]
	0x00400058	0x20080001	addi $8,$0,0x00000001	25: addi $t0,$zero,1 # 置flag为1
	0x0040005c	0x21290001	addi $9,$9,0x00000001	26: C: addi $t1,$t1,1 # i++
	0x00400060	0x0810000b	j 0x0040002c	27: j for #
	0x00400064	0x214affff	addi $10,$10,0xffffffff	28: B: addi $t2,$t2,-1 # num--
	0x00400068	0x08100007	j 0x0040001c	29: j while
	0x0040006c	0x03e00008	jr $31	30: A: jr $ra

此处我们不再对这个程序进行详细解释（之前实验已经叙述过了），但是我们要清楚这个程序是不能够直接导入到 CPU 中运行的，这是由于原来程序中的系统调用 syscall 在我们设计的 CPU 是用不了的，而我们还要在这里面加入我们设计的停机指令，在调整代码之后，还要注意到 j、jal 这类指令中的 JumpAddr 指的是绝对地址，因此我们还需要修改原来程序中的 j 和 jal 指令内容。具体修改过程有一些繁琐，这里就直接贴出修改后的代码：

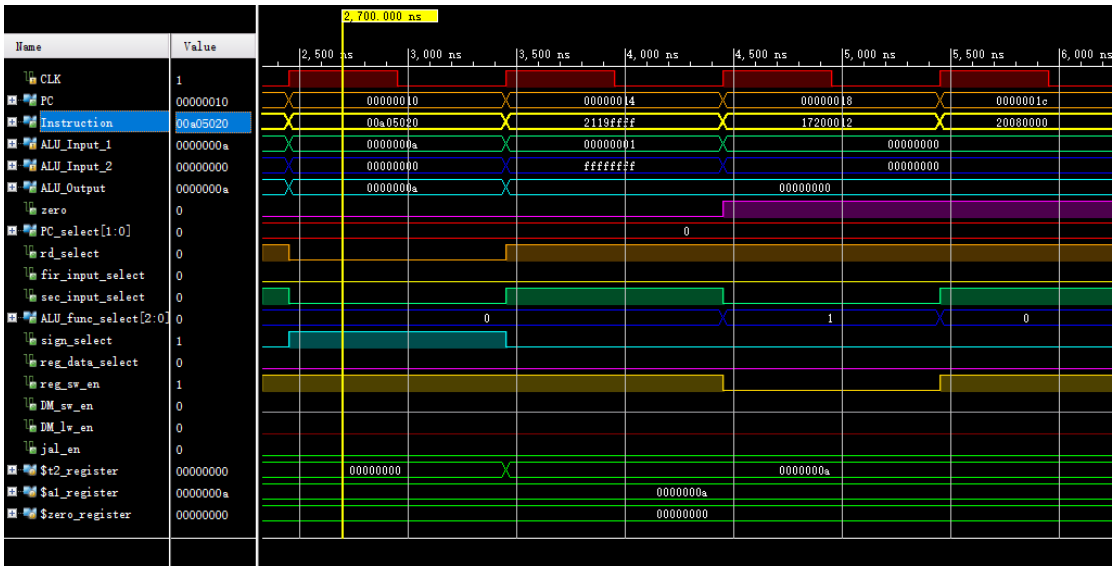
```
1  2005000a        addi $a1,$zero,10
2  0c000003        jal bubble
3  ffffffff        halt
4                bubble:
5  20080001        addi $t0,$zero,1    # 置flag为1
6  00a05020        add $t2,$a1,$zero   # 置num为size
7                while:
8  2119ffff        addi $t9,$t0,-1     # $t9=flag-1，判断flag=1
9  17200012        bne $t9,$zero,A     #
10 20080000        addi $t0,$zero,0    # 置flag为0
11 20090000        addi $t1,$zero,0    # 置i为0
12               for:
13 2159ffff        addi $t9,$t2,-1     # $t9=num-1，判断i<num-1
14 0139c02a        slt $t8,$t1,$t9     #
15 1300000b        beq $t8,$zero,B     #
16 0009c880        sll $t9,$t1,2       # i左移2位，即*4
17 0324c020        add $t8,$t9,$a0     # $t8是arr[i]地址
18 8f190000        lw $t9,0($t8)       # $t9=arr[i]
19 8f0f0004        lw $t7,4($t8)       # $t7=arr[i+1]
20 01f9702a        slt $t6,$t7,$t9     # 判断arr[i]>arr[i+1]
21 11c00003        beq $t6,$zero,C     #
22 af0f0000        sw $t7,0($t8)       # arr[i+1]->arr[i]
23 af190004        sw $t9,4($t8)       # arr[i]->arr[i+1]
24 20080001        addi $t0,$zero,1    # 置flag为1
25               C:
26 21290001        addi $t1,$t1,1      # i++
27 08000009        j for               #
28               B:
29 214affff        addi $t2,$t2,-1     # num--
30 08000005        j while
31               A:
32 03e00008        jr $ra
```

8. 仿真查看波形

经过上述步骤的工作，我们有了单周期 CPU 及冒泡排序程序，接下来我们将冒泡排序程序导入到 CPU 查看仿真结果。

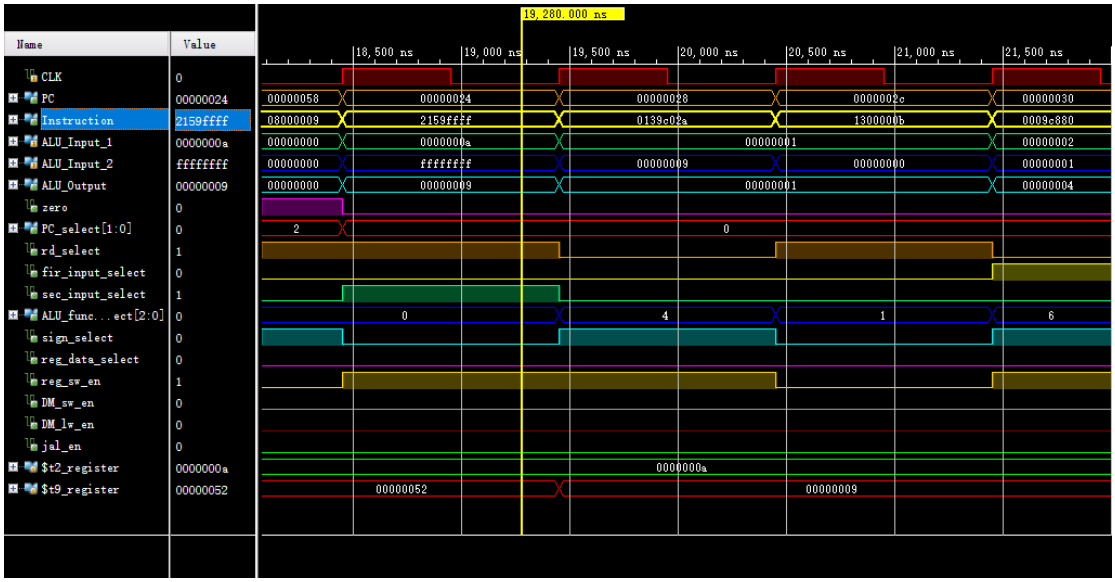
首先，我们先紧接着第 6 步测试各条指令时提到的三种指令 R、I、J 型，分别以 add、addi、j 为例子，下面直接看波形。

对于 add，我们看指令 add $t2,$a1,$zero，其机器代码为 00a05020。这里$t2 寄存器在 CPU 中编号为 10，$a1 寄存器在 CPU 中编号为 5，而$zero 寄存器在 CPU 编号为 0。波形如下：



在最左边可以看到当我们执行 00a05020 这条指令时，ALU_func_select 为 00，此时选择 ALU 功能为加法，且 reg_sw_en 为 1，此时允许结果写回寄存器堆，ALU 的输入输出是符合预期的。我们可以看到$a1=0ah，$zero=00h，两者相加确实得到下一个时钟周期的$t2=0ah。指令的具体运行过程可以参考步骤 6 测试指令中的说明。

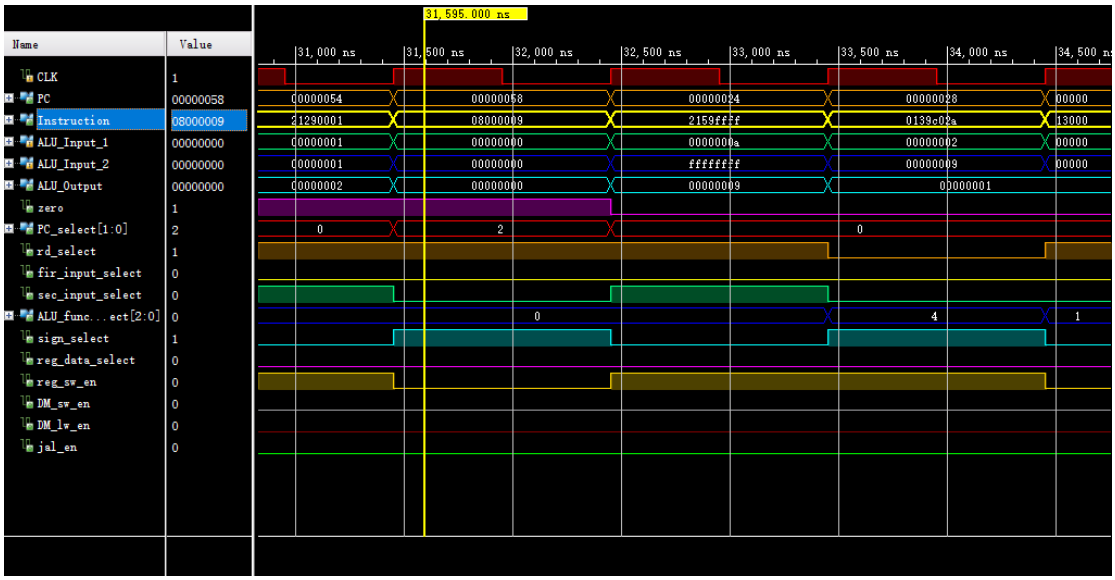
对于 addi，我们看指令 addi $t9,$t2,-1，其机器代码为 2159ffff。这里$t9 寄存器在 CPU 中编号为 25，$t2 寄存器在 CPU 中编号为 10。波形如下：



在最左边可以看到当我们执行 2159ffff 这条指令时，ALU_func_select 为 00，此时选择 ALU 功能为加法，且且 reg_sw_en 为 1，此时允许结果写回寄存器堆，ALU 的输入输出是符合预期的。与上述的 add 不同的是，addi 在此处的 rd_select 为 1，意味着 ALU 运算结果写回寄

存器 rt，还有 addi 在此处的 sec_input_select 为 1，意味着 ALU 的第二个输入来自于指令中的立即数，以及 addi 在此处的 sign_select 为 0，意味着送进 ALU 的立即数需要进行有符号扩展。这前面这些不同就展示了 addi 于 add 的区别。指令的具体运行过程可以参考步骤 6 测试指令中的说明。

对于 j，我们看指令 j for（for 是标识符，看上面写的程序就清楚这里的意思了），其机器代码为 08000009。这里 for 的地址是 00000009，而且我们查询刚才的程序知道 00000009 的指令是 addi $t9, $t2, -1，机器代码是 2159ffff，等会只需要查看 j 执行后下一个时钟周期指令是否为 2159ffff。波形如下：



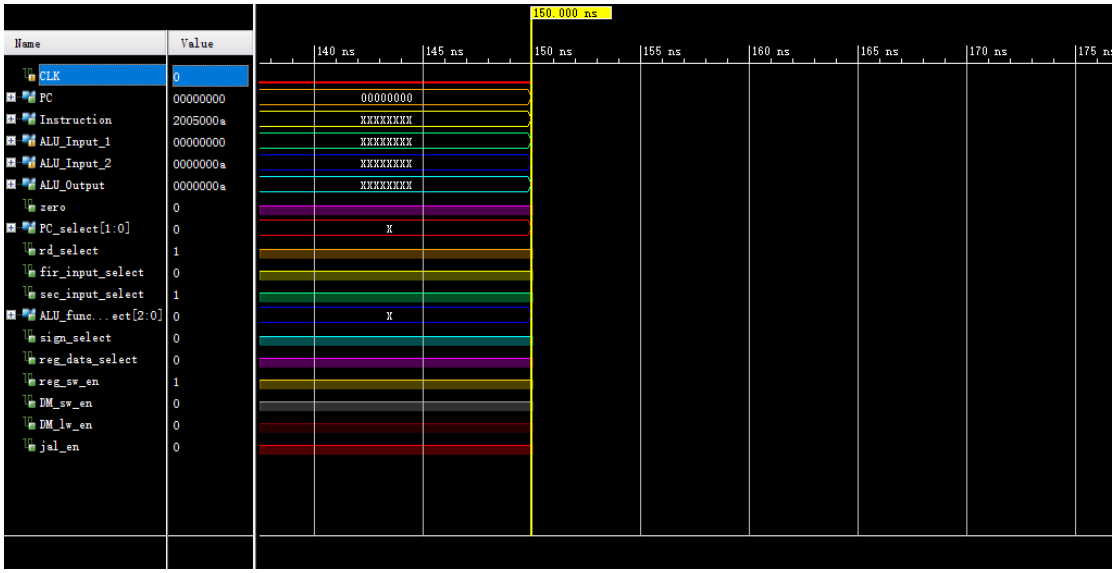
可以看到，在指令 j 执行后，下一个时钟周期指令变为 2159ffff，符合预期。至于下面的控制信号，事实上，在 j 执行时，所有有关写的操作，如写寄存器堆或写数据存储器都为无效，这样就满足控制要求了，其余的控制信号为何并没有太大关系。指令的具体运行过程可以参考步骤 6 测试指令中的说明。

在看完这些指令具体执行的波形后，我们接下来可以查看单周期 CPU 跑冒泡排序程序的结果。我们需要先明确一下，我所编写的冒泡排序程序跑的是对 10 个数进行排序，且我导入排序的 10 个数据（16 进制）如下：

```
1  00000052
2  00000096
3  00000001
4  00000000
5  00000256
6  00000063
7  00000074
8  00001111
9  000000ff
10 01234678
```

接下来我们开始看波形。

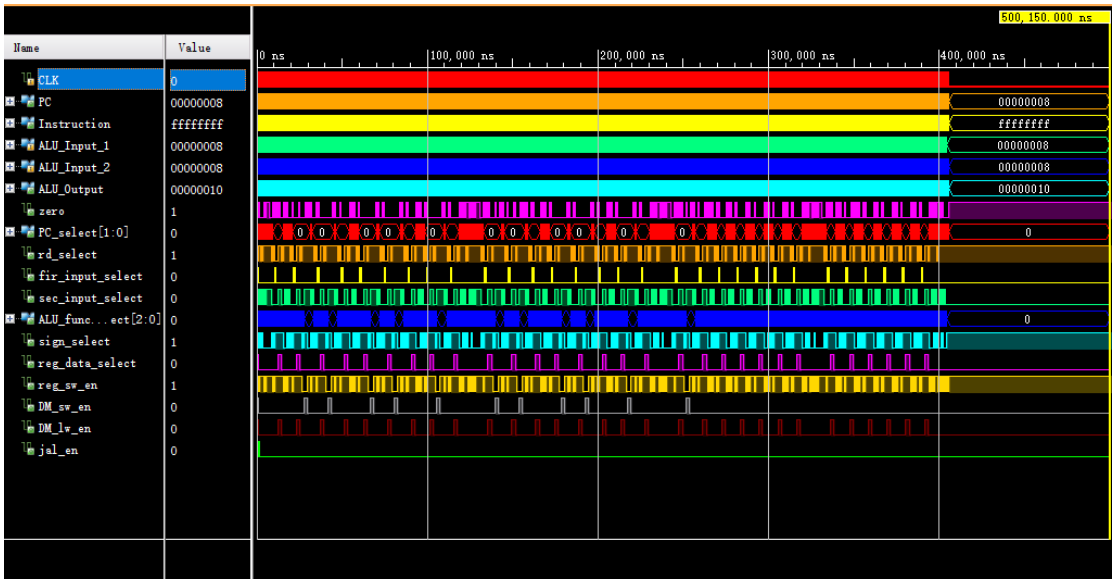
刚开始运行 150ns，这是数据初始化阶段，为了看看我们的数据和指令是否成功导入到了 CPU 中，波形全为 X，表示未初始化，波形以及导入的指令数据如下：



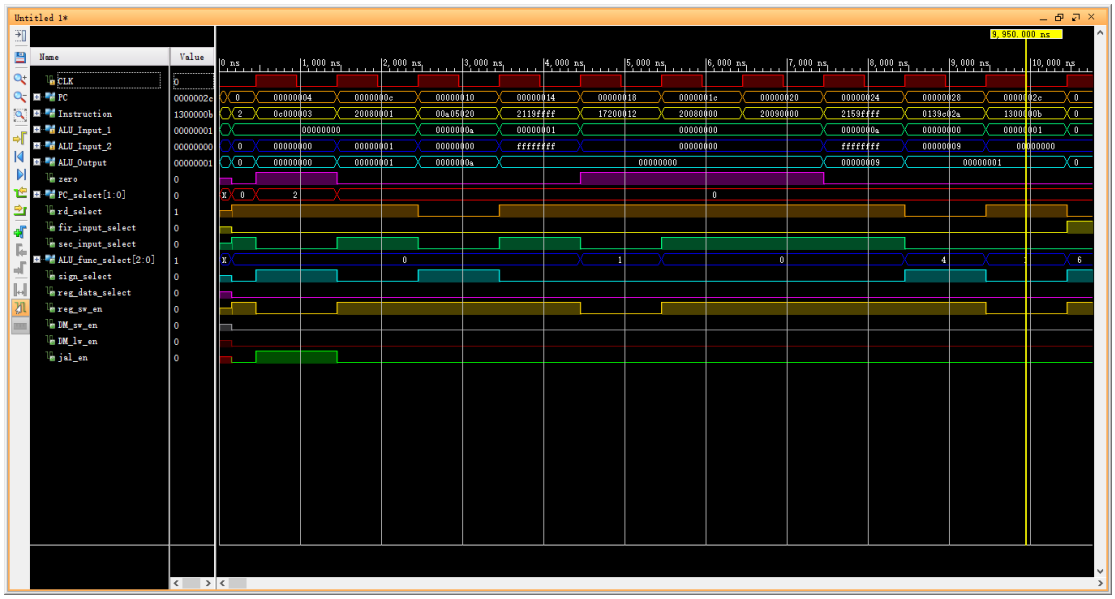
Name	Value	Data ..
address[31:0]	00000000	Array
CLK	0	Logic
instruction...	2005000a	Array
IM[0:255][7:0]	20,05,00,...	Array
[0][7:0]	20	Array
[1][7:0]	05	Array
[2][7:0]	00	Array
[3][7:0]	0a	Array
[4][7:0]	0c	Array
[5][7:0]	00	Array
[6][7:0]	00	Array
[7][7:0]	03	Array
[8][7:0]	ff	Array
[9][7:0]	ff	Array
[10][7:0]	ff	Array
[11][7:0]	ff	Array
[12][7:0]	20	Array
[13][7:0]	08	Array
[14][7:0]	00	Array

Name	Value	Data ..
address[31:0]	0000000a	Array
data[31:0]	00000000	Array
sw_en	0	Logic
lw_en	0	Logic
ori_CLK	1	Logic
CLK	0	Logic
out[31:0]	00010000	Array
special_0[3...	00000052	Array
special_1[3...	00000096	Array
special_2[3...	00000001	Array
special_3[3...	00000000	Array
special_4[3...	00000256	Array
special_5[3...	00000063	Array
special_6[3...	00000074	Array
special_7[3...	00001111	Array
special_8[3...	000000ff	Array
special_9[3...	01234678	Array
datain[31:0]	00000000	Array
place[3:0]	0	Array

（左为指令存储器，右为数据存储器，其中指令存储器是 8 位 1 个单元，前面已经说明过了）可以看到指令与数据已经导入，下面可以继续运行 500000ns，此时程序就运行完了，再看看运行完毕的整体波形图，如下：



再看看局部波形图，如下：



首先看到整体波形图中指令最后全变为 ffffffff，说明停机指令确实起作用了，中间波形错落有致，实际上也是完成了我们所导入的程序，下面看看数据存储器中的数据是否被排好序，结果图如下：

Name	Value	Data ..
address[31:0]	00000010	Array
data[31:0]	00000008	Array
sw_en	0	Logic
lw_en	0	Logic
ori_CLK	1	Logic
CLK	0	Logic
out[31:0]	00000074	Array
special_0[3...	00000000	Array
special_1[3...	00000001	Array
special_2[3...	00000052	Array
special_3[3...	00000063	Array
special_4[3...	00000074	Array
special_5[3...	00000096	Array
special_6[3...	000000ff	Array
special_7[3...	00000256	Array
special_8[3...	00001111	Array
special_9[3...	01234678	Array
datain[31:0]	00000000	Array
place[3:0]	0	Array

可以看到，对比前面导入时的情况，我们可以确定，冒泡排序在单周期 CPU 上运行成功。

9. 烧板

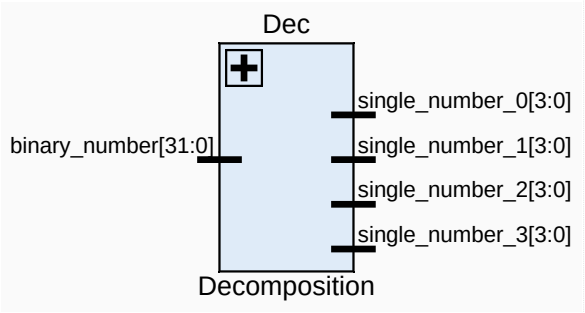
在 vivado 上完成仿真后，按照实验要求，还需要将单周期 CPU 烧录到开发板上，个人认为,这一部分是整个实验中最难的部分。烧录到开发板上,要求用开关和按钮来实现输入，用数码管显示来实现输出。下面从输入和输出两个方面完成。

先解决输入的问题。在开发板上有若干个开关和 5 个按钮，我们考虑开关当作二进制输入（开关共 16 个，只用其中 13 个来作为输入，剩下 3 个作为其他控制用途），5 个按钮有上下左右中之分，可以让上下按钮控制当前地址的数据加一和减一，让左右按钮控制地址加一和减一（也就是可以通过左右按钮切换地址查看数据），让中间的按钮表示确认按钮（也就是当你按下时，确认当前要输入的数据）。这一部分只需要在原来的顶层文件中加入这些相关的引脚，并在前述的 Data Memory 上做一些改动即可，由于这与 CPU 设计关系不大，就不在此赘述相关代码的编写了。

再解决输出的问题。开发板上有数码管可以使用，我们可以在数码管上显示我们将要排序的 10 个数据,可以选择循环显示或者用左右按钮切换显示(这个选择就占用了 1 个开关，剩下的 2 个按钮，1 个用于控制排序开始，还有 1 个用于选择用开关输入数据还是按钮加减来输入数据），而为了让数码管能够实现我们需要实现的功能，我又编写了几个小的模块来实现数码管的功能，分别有 Decomposition（将要显示的数字分解为 4 个个位数的数字）、Num_7seg(将数字转化为对应的 7 段码)、Display(接受 7 段码实现在数码管上显示的模块)、Top_for_Display（对前面三个模块以及一些按钮操作的封装）。

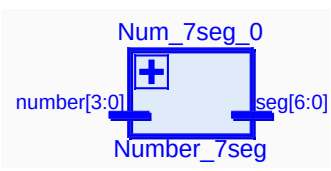
下面看模块的接口与代码(与 CPU 设计关系不大,在此就不赘述其功能是如何实现的了)。

第一个是 Decomposition:



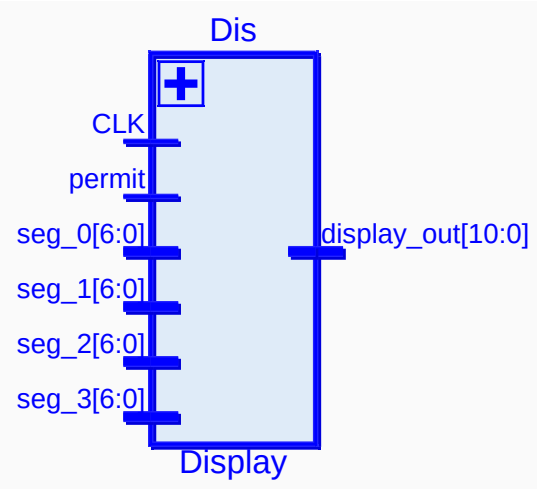

```
module Decomposition(
    input [31:0] binary_number,
    output reg [3:0] single_number_0,
    output reg [3:0] single_number_1,
    output reg [3:0] single_number_2,
    output reg [3:0] single_number_3
    );

    always@(*)
    begin
        single_number_0<=binary_number%10;
        single_number_1<=binary_number/10%10;
        single_number_2<=binary_number/100%10;
        single_number_3<=binary_number/1000%10;
    end

endmodule
```

第二个是 Num_7seg:



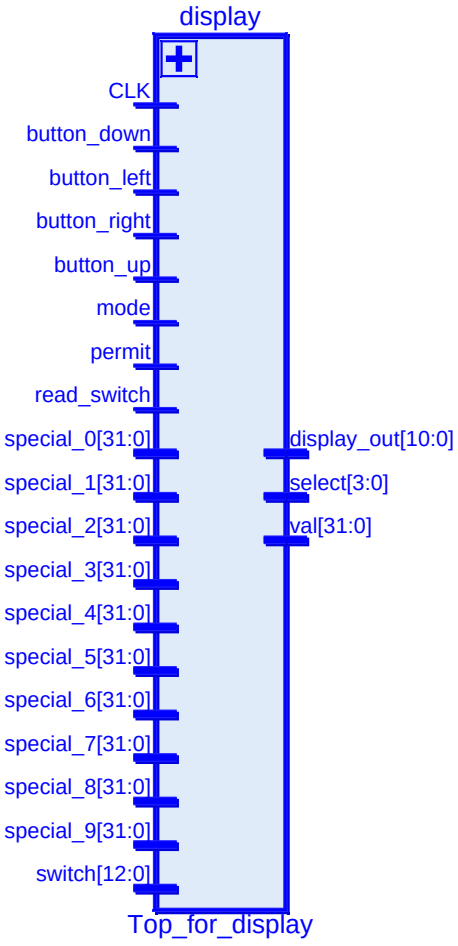

```
module Number_7seg(
    input [3:0] number,
    output reg [6:0] seg
    );

    always@(*)
    begin
        case(number)
            4'b0000:seg<=7'b0000001;
            4'b0001:seg<=7'b1001111;
            4'b0010:seg<=7'b0010010;
            4'b0011:seg<=7'b0000110;
            4'b0100:seg<=7'b1001100;
            4'b0101:seg<=7'b0100100;
            4'b0110:seg<=7'b0100000;
            4'b0111:seg<=7'b0001111;
            4'b1000:seg<=7'b0000000;
            4'b1001:seg<=7'b0000100;
            default:seg<=7'b1111111;
        endcase
    end

endmodule
```

第三个是 Display:



代码过长就不附了。

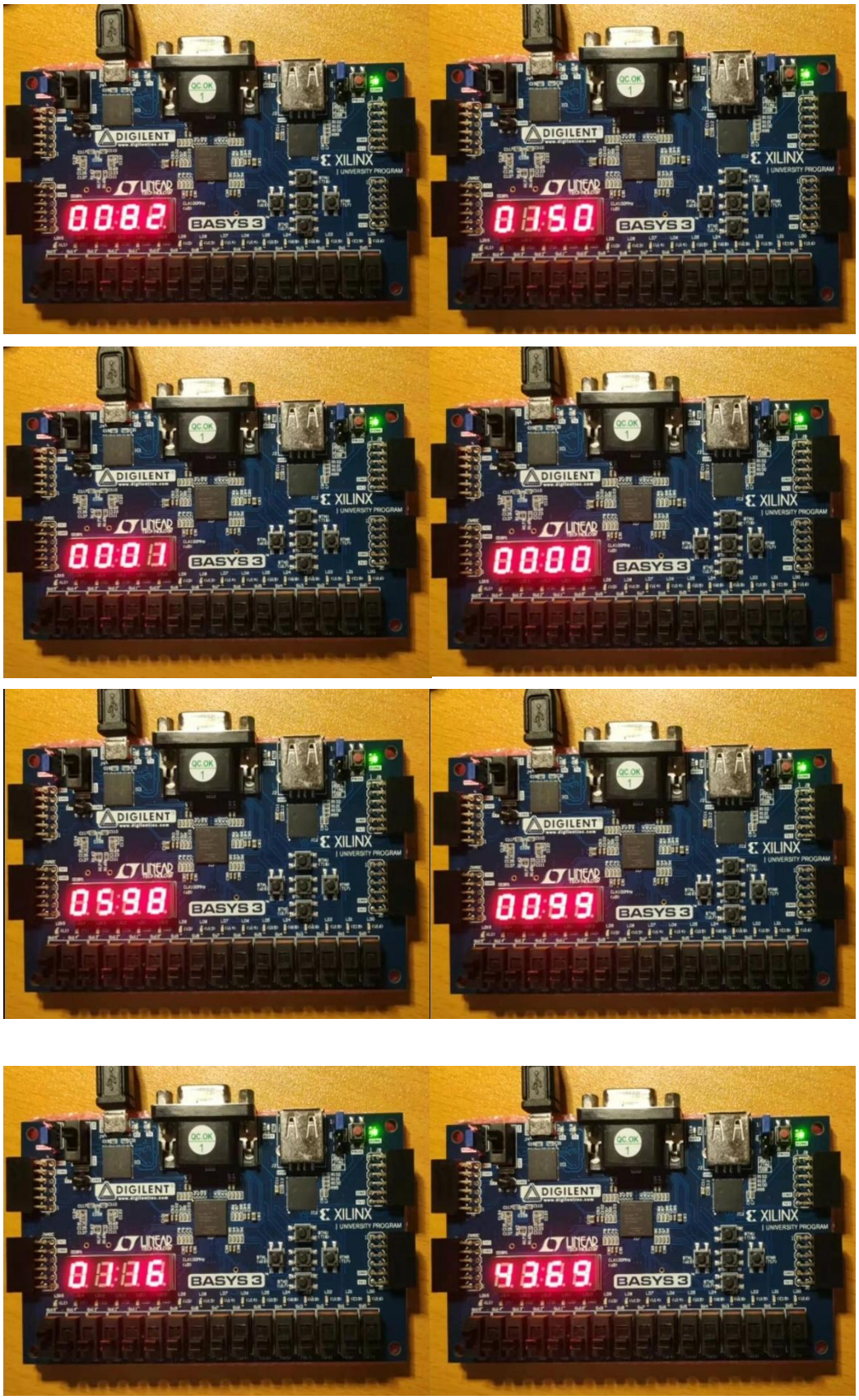
第四个是 Top_for_display:

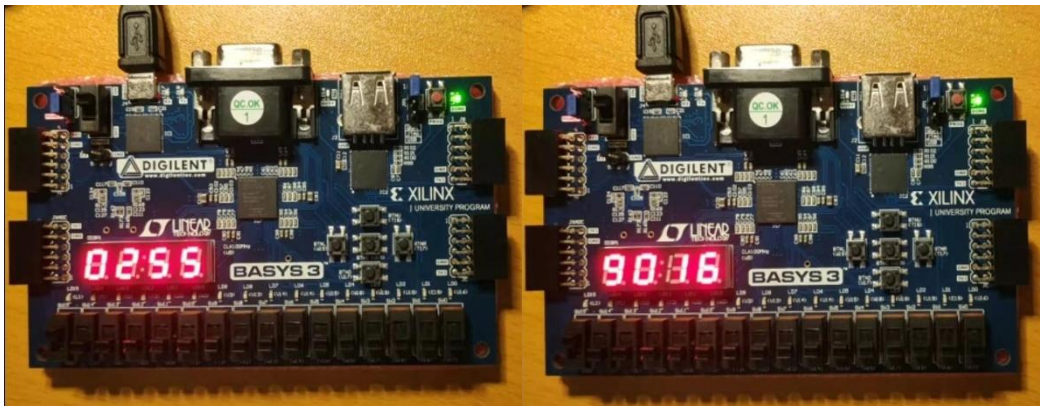


代码过长就不附了。

以上就是实现烧板额外编写的模块了，实际上在原来 CPU 的顶层文件还需要做修改以便融入输入输出模块，关于这其中设计遇到的问题及解决会在最后的总结中提及。

接下来可以看看在开发板上的结果，由于在报告难以展示运行效果，故这里仅展示几张图片，具体的运行效果在课堂检查中已检查过，若有需要，可以在这份报告的附件（即演示视频）中查看。


DIGILENT
XILINX
UNIVERSITY PROGRAM
BASYS 3

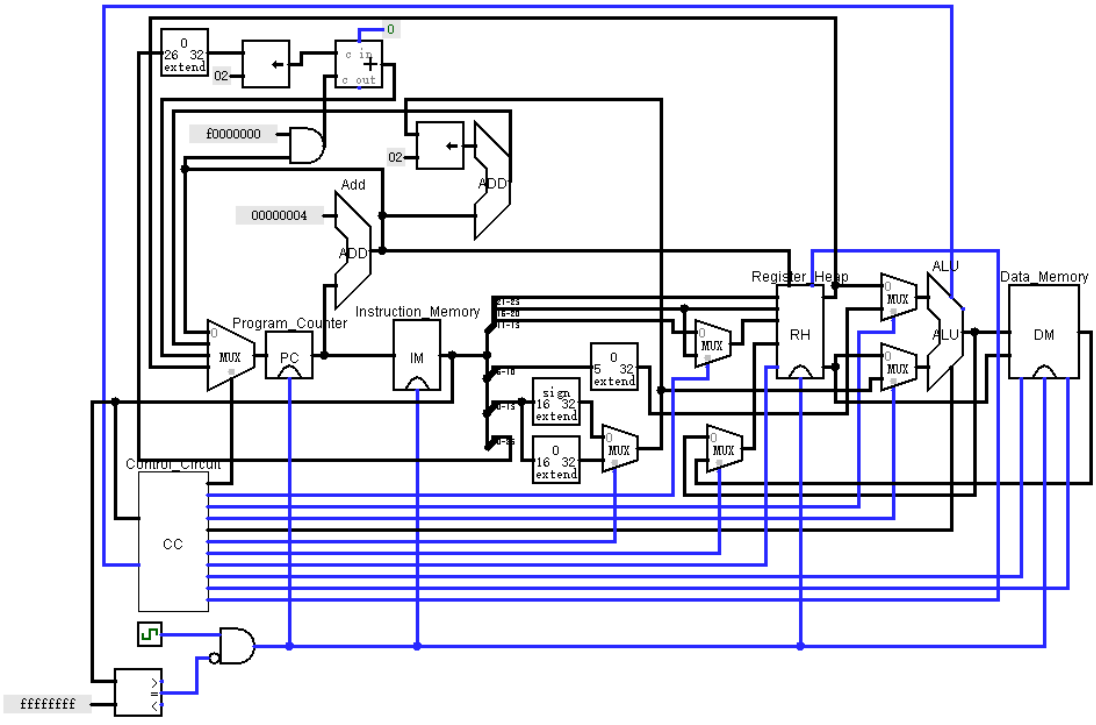


至此，单周期 CPU 设计完毕，冒泡排序及开发板输入输出设计成功。

六、实验总结

这个最后的总结部分会分为两个部分，一个是我在设计过程中遇到的问题（若已解决会有相关的解决办法），另外一个是对该设计各个不足的总结。

在这次设计中，因为在实验的布置之前我就已经着手开始做了，但最开始并非使用 vivado 及 Verilog 语言编写，而是先采用的 logisim 直接拼接部件实现，所以在“写”这个单周期 CPU 之前，我先在 logisim 上“画”出来了，如下图：



图中 CPU 在 logisim 上仿真是正确的。但是后来在 vivado 上编写单周期 CPU 与这个“画”CPU 的过程不大相同。由于是用 Verilog 编写，打代码不如“画”CPU 来的具象，故打代码编写时虽然思路非常清晰，但是出现的 bug 会更多，且更加难以找到，加上本身对 vivado 调试十分陌生，这无疑大大加大了设计难度。因此，除去基本使用问题（即对 Verilog 不熟悉不会使用）后，我遇到的第一个问题就是**调试**。由于一开始并不知道 vivado 有仿真这个功能，之前一向都是人肉 debug，效率低下。但知道有仿真后，遇到的问题又来了，我**所编写的代码无法仿真**，一开始是由于没有写测试代码 testbench（事实上一开始并不知道这需要写测试代码），在写了测试代码之后，每次启动仿真都会使电脑崩溃，后来经过好长一段

时间查询才发现在代码编写中，诸如“2’ha”这样的代码是错误的，应该改为“4’ha”，在单引号前代表的应该是后面数据的二进制的位数,而我在定义存储器时就出现大量这样的错误，而 vivado 实际上是有给出 warning，但没有给 error，还自行优化了，但一旦进入仿真，就会访问未定义的内存，直接导致电脑卡死崩溃，修正代码错误之后，问题解决。但是在进入仿真后，**指令与数据始终不能导入指令存储器和数据存储器**，经过对编译信息的查询，以及在 xilinx 上搜索才得知导入的数据文件，其路径上不能有反斜杠。上述问题解决之后，剩下的 debug 都只是对代码的修修补补，最后仿真成功，冒泡排序成功运行。但是在烧录到开发板上又遇到了许多的问题。在最初，我只准备实现数据在数码管上循环显示，但是**烧录到板上时，数据出现了始终无法读进开发板上的问题**，在经过长时间的搜索后才找到解决方法，即在使用系统命令 readmemh（或 readmemb）时，最后两个参数（即数据存放的起始地址和终止地址）不要省略，以及在从数据存储器读出数据时其索引不能只用常数表示（比如说，我想读 mem[10]的内容，我应该写成 reg addr=0;mem[addr+10]=…;），通过这两个方法,我就成功地把数据读进了开发板并完成了排序。在完成显示,又有要求实现输入，在这个过程中遇到的一个问题是**如何检测除了时钟的上升沿之外同时检测按钮的上升沿**（用于检测按钮的按下，一旦按下表示确认数据写入数据存储器），对于这个问题，我上网查询就找到解决办法，写了一小段检测上升沿的代码就解决了（其实就是增加一个变量存储目标的前一个状态即可）。以上就是我遇到问题并解决的过程。

下面是此次设计的不足之处。事实上最明显的就是***各变量和各控制信号的命名问题***，由于在设计这个单周期 CPU 时没有参考别的资料，只是粗略看过 MIPS Reference Data 和课本中最开始的一张数据通路之后便自己开始着手做，正是这个原因，我的所有命名都是自己想出来的，与大众化的命名不同从而导致代码可读性极差。另外，设计中还存在的问题就是***最初设计时没有考虑周全***，这有一部分原因是设计之前参考资料过少，还有一部分就是自己的大意，比如说 jal 指令在最初设计中并没有涉及，导致后面快到跑冒泡排序时才意识到自己的程序用到了这条指令，后来便是直接在原代码上直接添加修补，导致代码十分“丑陋”和“随意”，而且代码中基本没有注释，可读性十分差，再加上后来又添加的为了在开发板上实现输入输出的代码，使得整个设计十分混乱，虽然最终可以运行成功，但源代码不堪入目。最后还有***停机指令的设计问题***,这个不足是在设计多周期 CPU 之后才发现的,原因在于，我的停机指令的工作原理是直接让时钟不再跳动，直接置零，这确实也能做到停机，但与实际上而言这不太合理，事实上停机应该只是让 Program Counter 停止计数而已，在多周期 CPU 设计时这一点将会被改进的。

总而言之，此次单周期 CPU 设计让我熟悉了 CPU 的设计方法及流程，还有 vivado 的用法以及烧板的流程，其中遇到的许多问题让我耗费了大量时间，但是这对后面的多周期 CPU 设计会有许多帮助，除此之外，我也会慢慢改善代码风格，争取在下一次的多周期 CPU 设计中做的更好一点。