一、 实验目的

1). 掌握单周期 CPU 数据通路图的构成、原理及其设计方法；

2). 掌握单周期 CPU 的实现方法，代码实现方法；

3). 认识和掌握指令与 CPU 的关系；

4). 掌握测试单周期 CPU 的方法。

二、 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

1. 算术运算指令

(1). add rd , rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

(2). sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

(3). addiu rt , rs ,immediate

001001	rs(5 位)	rt(5 位)	immediate(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(immediate)；immediate 做符号扩展再参加“与”运算。

2. 逻辑运算指令

(4). andi rt , rs ,immediate

001100	rs(5 位)	rt(5 位)	immediate(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(immediate)；immediate 做 0 扩展再参加“与”运算。

(5). and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

(6). ori rt , rs ,immediate

001101	rs(5 位)	rt(5 位)	immediate(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(immediate)。

(7). or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

3. 移位指令

(8). sll rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] « sa。

4. 比较指令

(9). slti rt, rs,immediate 带符号数

001010	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if GPR[rs] < sign_extend(immediate) GPR[rt] =1 else GPR[rt] = 0。

5. 存储器读/写指令

(10). sw rt , offset (rs) 写存储器

101011	rs(5 位)	rt(5 位)	offset(16 位)

功能：memory[GPR[base] + sign_extend(offset)] ← GPR[rt]。

(11). lw rt , offset (rs) 读存储器

100011	rs(5 位)	rt(5 位)	offset (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(offset)]。

6. 分支指令

(12). beq rs,rt, offset

000100	rs(5 位)	rt(5 位)	offset (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(offset) « 2
else pc ← pc + 4
特别说明：offset 是从 PC+4 地址开始和转移到的指令之间指令条数。offset 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 offset 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

(13). bne rs,rt, offset

000101	rs(5 位)	rt(5 位)	offset (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(offset) « 2
else pc ← pc + 4

(14). bltz rs, offset

000001	rs(5 位)	00000	offset (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (offset) « 2
else pc ← pc + 4。

7. 跳转指令

(15). j addr

000010	addr(26 位)

功能：PC ← {PC[31:28] , addr , 2’ b0}，无条件跳转。
说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

8. 停机指令

(15). halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三、 实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期

CPU 在处理指令时，一般需要经过以下几个步骤：

(1) 取指令 (**IF**)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

(2) 指令译码 (**ID**)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

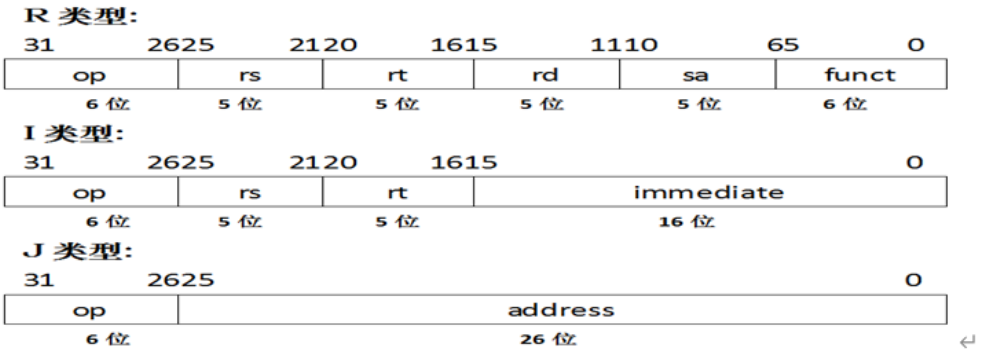
(3) 指令执行 (**EXE**)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问 (**MEM**)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回 (**WB**)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

图 1: 单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中，

- **op**：为操作码；
- **rs**：只读。为第 1 个源操作数寄存器，寄存器地址（编号）是 00000 11111，00 1F；
- **rt**：可读可写。为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；
- **rd**：只写。为目的操作数寄存器，寄存器地址（同上）；
- **sa**：为位移量（shift amt），移位指令用于指定移多少位；
- **funct**：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能与操作码配合使用；
- **immediate**：为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载（Laod）/数据保存（Store）指令的数据地址字节偏移量和分支指令中相对程序计数器（PC）的有符号偏移量；
- **address**：为地址。

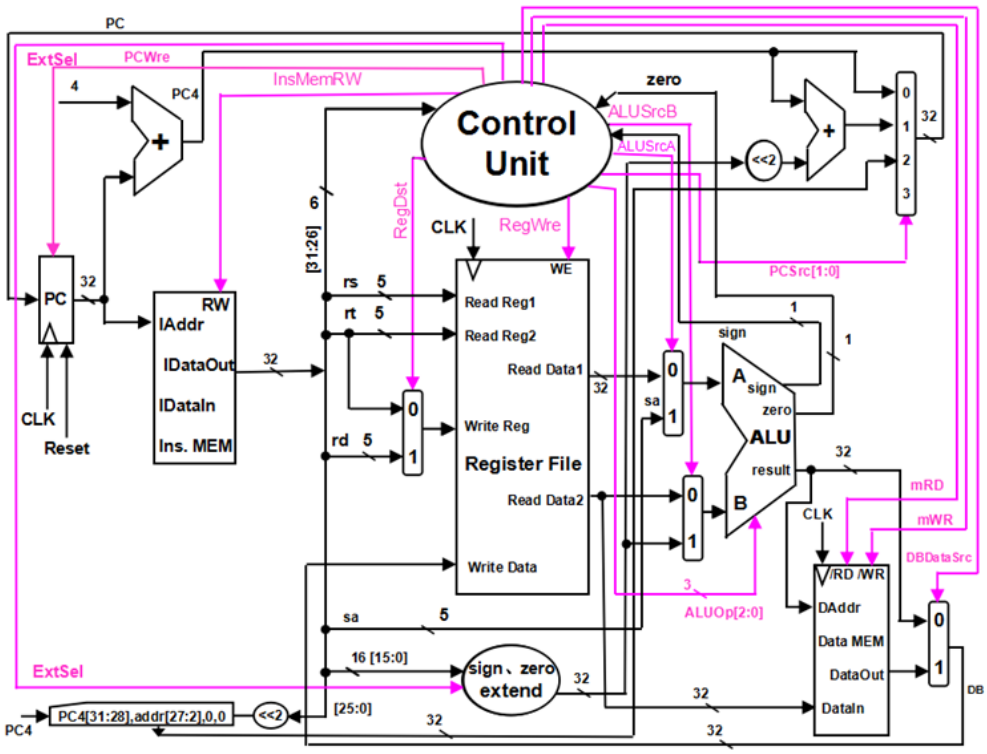


图 2: 单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，先给出寄存器地址，读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

相关部件及引脚说明：

- **Instruction Memory：指令存储器，**
  - Iaddr，指令存储器地址输入端口
  - IDataIn，指令存储器数据输入端口（指令代码输入端口）
  - IDataOut，指令存储器数据输出端口（指令代码输出端口）
  - RW，指令存储器读写控制信号，为 0 写，为 1 读
- **Instruction Memory：指令存储器，**
  - Daddr，数据存储器地址输入端口
  - DataIn，数据存储器数据输入端口
  - DataOut，数据存储器数据输出端口
  - RD，数据存储器读控制信号，为 0 读
  - WR，数据存储器写控制信号，为 0 写

表 1: 控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw	来自移位数 sa，同时，进行 (zero-extend)sa，即 271'b0,sa，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器 (Ins. Data)
mRD	输出高阻态	读数据存储器，相关指令：lw
mWR	无操作	写数据存储器，相关指令：sw
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)immediate（0 扩展），相关指令：addiu、andi、ori	(sign-extend)immediate（符号扩展），相关指令：slti、sw、lw、beq、bne、bltz
PCSrc[1..0]	00：pc←pc+4，相关指令：add、addiu、sub、or、ori、and、andi、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0)； 01：pc←pc+4+(sign-extend)immediate，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1)； 10：pc←(pc+4)[31:28],addr[27:2],2'b00，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择 (000-111)，看功能表	

表 2: ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A - B	减
010	Y = B « A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y = (A < B) ? 1 : 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) ∧ (rega[31] == regb[31])) ∨((rega[31] ==1 ∧ regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

- **Register File：寄存器组**
  - Read Reg1，rs 寄存器地址输入端口
  - Read Reg2，rt 寄存器地址输入端口
  - Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段
  - Write Data，写入寄存器的数据输入端口
  - Read Data1，rs 寄存器数据输出端口
  - Read Data2，rt 寄存器数据输出端口
  - WE，写使能信号，为 1 时，在时钟边沿触发写入
- **ALU：算术逻辑单元**
  - result，ALU 运算结果
  - zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0
  - sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能 (以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表，再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计。

四、 实验设备

PC 机一台，BASYS 3 实验板一块，Xilinx Vivado 开发软件一套。

五、 实验分析与设计

1. 实验分析：

(1) 设计单周期 CPU 时，先利用模块化的思想分别设计各个功能部件，如 PC、ALU 等，在每个文件中根据不同部件的功能编写代码，然后设计总的模块 MonocylicCPU，在其中定义所有元件的连接方式；

(2) 进行烧板时，需要定义 Debouncer 模块消除按下按键时按键会出现人眼无法观测但是系统会检测到的抖动变化，以及定义 SegLED 模块将 CPU 产生的信号转化为对应的 7 段数码管的灯信号，最后建立顶层模块 Basys3 进行烧板。

2. 代码设计：

(1) PC 模块

程序计数器：起始时，PC 中的地址为 0X00000000，每当 CLK 的上升沿或 Reset 的上升沿到来时，PC 值将会作出相应的改变：若 Reset = 1，则将 PC 重新置为 0X00000000；否则若 CLK 上升沿到来且 PCWre 信号为 1，将输出新的地址。

```
module PC(
    input CLK,
    input Reset,
    input PCWre,
    input[31:0] AddrIn,
    output reg[31:0] AddrOut
);
    //initially PC is 0
    initial
    begin
        AddrOut = 0;
    end

    //each time when reset is 1 or CLK posedge comes
    always@(posedge CLK or posedge Reset)
    begin
        if(Reset)
        begin
            AddrOut = 0;
        end
        else if(PCWre)
        begin
            AddrOut = AddrIn;
        end
    end
endmodule
```

(2) InsMEM 模块

指令寄存器：根据实验要求，指令存储器和数据存储器存储单元宽度一律使用 8 位 (即一个字节的存储单位)，因此选用 256 大小的 8 位寄存器数组来存放指令。起始时，调用函数 readmemb 在默认位置打开 instruction.txt 文件，读取二进制指令，并保存在 ins_registers 中，每当 RW 信号

或 IAddr 即输入的地址改变时，则输出放在 ins_registers 对应位置中的指令，这里有两点需要注意的地方：

a. 在 alway@ 列表中需要把 IAddr 信号列入，原因是当 RW 信号持续为高时，若 IAddr 改变，即需要获得 ins_registers 其他位置的指令时，InsMEM 模块将无法识别；

b. 在读入 ins_registers 时，第一条指令的前八位是存放在 ins_registers[0] 中，并依次类推，所以读出数据时，根据图 2 的格式，我们需要将指令的前八位放置于 IDataOut[0]，即进行一定程度上的逆序输出。

```
module InsMEM(
    input[31:0] IAddr,
    input RW,
    output reg[31:0] IDataOut
    );
    //required to be reg[7:0]
    reg[7:0] ins_registers[255:0];
    initial
    begin//remember to reverse the order
        $readmemb("instructions.txt", ins_registers);
    end

    always@(RW or IAddr)
    begin
        if(RW == 0)//we do not have RD signal, so use !RW
        begin
            IDataOut = {ins_registers[IAddr], ins_registers[IAddr+1], ins_registers[IAddr+2],
                ins_registers[IAddr+3]};
        end
    end
endmodule
```

(3) DataMEM 模块

内存：根据实验要求，指令存储器和数据存储器存储单元宽度一律使用 8 位 (即一个字节的存储单位)，因此选用 256 大小的 8 位寄存器数组来存放数据。当 RD 信号或 DAddr 输入地址改变时，进行读内存操作；当 CLK 下降沿到来且 WR 信号为 1 时，进行写内存操作。

```
module DataMEM(
    input[31:0] DAddr,
    input[31:0] DataIn,
    input CLK,
    input RD,
    input WR,
    output reg[31:0] DataOut
    );
    //required to be reg[7:0]
    reg[7:0] registers[255:0];

    always@(RD or DAddr)
    begin
        if(RD)
        begin//big endian
            DataOut = {registers[DAddr + 3], registers[DAddr + 2], registers[DAddr + 1],
                registers[DAddr]};
        end
```

```
    end

    always@(negedge CLK)
    begin
        if(WR)
        begin//big endian
            registers[DAddr + 3] <= DataIn[31:24];
            registers[DAddr + 2] <= DataIn[23:16];
            registers[DAddr + 1] <= DataIn[15:8];
            registers[DAddr] <= DataIn[7:0];
        end
    end
endmodule
```

(4) SignZeroExtend 模块

立即数扩展: 输入 16 位立即数，并根据立即数的最高位 (符号位) 以及 ExtSel 信号进行符号扩展或零扩展。

```
module SignZeroExtend(
    input ExtSel,
    input[15:0] DataIn,
    output[31:0] DataOut
    );
    assign DataOut = {ExtSel&&DataIn[15] ? 16'hffff : 16'h0000, DataIn};
endmodule
```

(5) RegisterFile 模块

寄存器组：根据实验配套的测试指令中得知，本次实验需要用到的寄存器为 12 个，在实现中选择使用 32 大小的数组来实现，且测试指令中其实并没有对需要用到的寄存器进行赋值，因此在初始化时需要对数组赋初始值 0。对于读寄存器组采用持续读入，对于写寄存器则需要满足 CLK 的下降沿到来且 WE 信号为 1.

```
module RegisterFile(
    input[4:0] ReadReg1,
    input[4:0] ReadReg2,
    input[4:0] WriteReg,
    input[31:0] WriteData,
    input CLK,
    input WE,
    output[31:0] ReadData1,
    output[31:0] ReadData2
    );
    //I think 32 is quite enough, considering that test file uses only 10 regs
    reg[31:0] registers[31:0];
    integer i;
    initial
        begin
            for(i = 0; i < 32; i = i + 1)
                registers[i] <= 32'h00000000;
        end

    assign ReadData1 = registers[ReadReg1];
    assign ReadData2 = registers[ReadReg2];
```

```
22
23      always@(negedge CLK)
24      begin
25          if(WE)
26          begin
27              registers[WriteReg] = WriteData;
28          end
29      end
30  endmodule
```

(6) ALU 模块

算术逻辑单元：根据 ALUop 的信号与自定义的信号进行对应，对输入 A、B 进行相应的运算，并将计算结果输出至 result 中，注意根据实验要求，对于移位指令实现应当是 B « A。

```
1   module ALU(
2       input[31:0] A,
3       input[31:0] B,
4       input[2:0] ALUOp,
5       output sign,
6       output zero,
7       output reg[31:0] result
8       );
9       //ALUop ref
10      parameter opADD = 3'b000;//ADD, ADDIU
11      parameter opSUB = 3'b001;//SUB, BEQ, BNE, BLTE
12      parameter opSLL = 3'b010;//SLL
13      parameter opOR =  3'b011;//ORI, OR
14      parameter opAND = 3'b100;//ANDI, AND
15      parameter opSLTU = 3'b101;
16      parameter opSLT = 3'b110;
17      parameter opXOR = 3'b111;
18
19      assign zero = (result == 0);
20      assign sign = result[31];
21
22      always@(*)
23          begin
24              case(ALUOp)
25              opADD:   result =  A + B;
26              opSUB:   result =  A - B;
27              opSLL:   result =  B << A;//sa is connected to input A, so we should write B << A
28              opOR:    result =  A | B;
29              opAND:   result =  A & B;
30              opSLTU:  result =  A < B;
31              ////((rega<regb) && (rega[31] == regb[31] )) ||( ( rega[31] ==1 && regb[31] == 0))) ? 1:0
32              opSLT:   result = (((A < B) && (A[31] == B[31] )) ||( ( A[31] == 1 && B[31] == 0))) ? 1:0;
33              opXOR:   result =  A ^ B;
34              default: result =  0;
35              endcase
36          end
37  endmodule
```

(7) ControlUnit 模块

控制单元：通过解析 opcode(即指令前 6 位) 字段，生成控制各功能器件的信号，对于 **MIPS** 指

令，当 opcode=b000000 时，则需要改为解析 function 字段 (即指令后 6 位)。实现时，首先定义了各个状态常量如 IS_ADD、IS_SUB 等便于理解，然后根据表一中将指令解析结果与生成信号一一对应。此外，对于输出 ALUop 还需要根据解析结果产生相应的 ALU 操作信号。

值得注意的是：对于 IS_BEQ、IS_BNE 以及 IS_BLTZ 状态，在生成 ALUop 信号时，不应将其归入 slt 指令 (比较指令)，而应该归入 sub 指令 (减法指令)，因为这些指令还需要根据 sign 和 zero 信号的结果进行分支跳转的判断。

```
module ControlUnit(
    input zero,
    input sign,
    input[5:0] opcode,  //upper 6 bits of instruction, if opcode == 000000, use func
    input[5:0] func,    //lower 6 bits of instruction
    output PCWre,
    output[1:0] PCSrc,
    output ExtSel,
    output ALUSrcA,
    output ALUSrcB,
    output[2:0] ALUOp,
    output RegWre,
    output RegDst,
    output mRD,
    output mWR,
    output DBDataSrc,
    output InsMemRW
    );
    //assign _opcode = opcode ? opcode : func; if opcode == 000000, then replace opcode with func
    wire[5:0] _opcode = opcode ? opcode : func;
    //opcode ref
    parameter ADD =  6'b100000;
    parameter SUB =  6'b100010;
    parameter ADDIU = 6'b001001;
    parameter ANDI = 6'b001100;
    parameter AND =  6'b100100;
    parameter ORI =  6'b001101;
    parameter OR =   6'b100101;
    parameter SLL =  6'b000000;
    parameter SLTI = 6'b001010;
    parameter SW =   6'b101011;
    parameter LW =   6'b100011;
    parameter BEQ =  6'b000100;
    parameter BNE =  6'b000101;
    parameter BLTZ = 6'b000001;
    parameter J =    6'b000010;
    parameter HALT = 6'b111111;
    //ALUop ref
    parameter opADD = 3'b000;//ADD, ADDIU
    parameter opSUB = 3'b001;//SUB, BEQ, BNE, BLTE
    parameter opSLL = 3'b010;//SLL
    parameter opOR =  3'b011;//ORI, OR
    parameter opAND = 3'b100;//ANDI, AND
    parameter opSLTU = 3'b101;
    parameter opSLT = 3'b110;
    parameter opXOR = 3'b111;
    //state
    wire IS_ADD =    (_opcode == ADD);
    wire IS_SUB =    (_opcode == SUB);
    wire IS_ADDIU =  (_opcode == ADDIU);
```

```
    wire IS_ANDI =   (_opcode == ANDI);
    wire IS_AND =    (_opcode == AND);
    wire IS_ORI =    (_opcode == ORI);
    wire IS_OR =     (_opcode == OR);
    wire IS_SLL =    (_opcode == SLL);
    wire IS_SLTI =   (_opcode == SLTI);
    wire IS_SW =     (_opcode == SW);
    wire IS_LW =     (_opcode == LW);
    wire IS_BEQ =    (_opcode == BEQ);
    wire IS_BNE =    (_opcode == BNE);
    wire IS_BLTZ =   (_opcode == BLTZ);
    wire IS_J =      (_opcode == J);
    wire IS_HALT =   (_opcode == HALT);
    //assign signal
    assign PCWre =   !IS_HALT;
    assign PCSrc[1] = IS_J;
    assign PCSrc[0] = (IS_BEQ && zero) || (IS_BNE && !zero) || (IS_BLTZ && sign);
    assign ExtSel =  IS_SLTI || IS_SW || IS_LW || IS_BEQ || IS_BNE || IS_BLTZ || IS_ADDIU;//IS_ADDIU
        is here
    assign ALUSrcA = IS_SLL;
    assign ALUSrcB = IS_ADDIU || IS_ANDI || IS_ORI || IS_SLTI || IS_SW || IS_LW;
    assign RegWre =  !IS_BEQ && !IS_BNE && !IS_BLTZ && !IS_SW && !IS_HALT;
    assign RegDst =  IS_ADD || IS_SUB || IS_AND || IS_OR || IS_SLL;
    assign mRD =     IS_LW;
    assign mWR =     IS_SW;
    assign DBDataSrc = IS_LW;
    assign InsMemRW = 0;
    assign ALUOp =   (IS_ADD || IS_ADDIU) ? opADD :
                     (IS_ANDI || IS_AND) ? opAND :
                     (IS_ORI || IS_OR) ? opOR :
                     (IS_SLL) ? opSLL :
                     (IS_SUB || IS_BEQ || IS_BNE || IS_BLTZ) ? opSUB : opSLT;
                     /* Here is somthing important, we CAN NOT classify beq, bne, bltz to opSLT(making
                         comparison),
                      * instead we group them in opSUB, the main reason is that for some ouputs, for
                          example PCSrc[0]:
                      * its expression is relevant to both the opcode and the zero bit or sign bit,
                          simply making
                      * comparison between 2 nums will not set zero and sign, so we must use opSUB to
                          change them.
                      */
endmodule
```

(8) MonocylicCPU 模块

单周期 CPU：顶层连接模块，将上述的各个子模块对应的输入输出连接起来，并显示实验中要求的结果如当前 PC 值、ALU 计算结果等。

```
module MonocylicCPU(
    input CLK,
    input Reset,
    output[31:0] CurPC,
    output[31:0] NextPC,
    output[31:0] IDataOut,
    output[31:0] ReadData1,
    output[31:0] ReadData2,
    output[31:0] ALUresult,
    output[31:0] DB
```

```
    );
    wire InsMemRW;
    wire[2:0] ALUOp;
    wire[1:0] PCSrc;
    wire ExtSel;
    wire[31:0] ExtendOut;
    wire sign;
    wire[31:0] ExtAddr;
    wire PCWre;
    wire RegDst;
    wire RegWre;
    wire ALUSrcA;
    wire ALUSrcB;
    wire mRD;
    wire mWR;
    wire DBDataSrc;
    wire[31:0] PC4;
    wire[31:0] PseudoAddr;
    wire[4:0] MUXout_rt_rd;
    wire[31:0] MUXout_rd1_sa;
    wire[31:0] MUXout_rd2_ext;
    wire zero;
    wire[31:0] DataMEMOut;

    PC pc(.CLK(CLK),.Reset(Reset),.PCWre(PCWre),.AddrIn(NextPC),.AddrOut(CurPC));
    AddrAdder pcadder(.Offset(32'h00000004),.OriAddr(CurPC),.OffsetAddr(PC4));
    PseudoAddrAdder pseadder(.PCUp4(PC4[31:28]),.AddrLow26(IDataOut[25:0]),.DstAddr(PseudoAddr));
    InsMEM insmem(.IAddr(CurPC),.RW(InsMemRW),.IDataOut(IDataOut));
    MUX_2to1_5bits mux_rt_rd(.SelectSig(RegDst),.InputA(IDataOut[15:11]),
    .InputB(IDataOut[20:16]),.DataOut(MUXout_rt_rd));
    ControlUnit controlunit(.zero(zero),.sign(sign),.opcode(IDataOut[31:26]),
    .func(IDataOut[5:0]),.PCWre(PCWre),.PCSrc(PCSrc),.ExtSel(ExtSel),.ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),.ALUOp(ALUOp),.RegWre(RegWre),.RegDst(RegDst),.mRD(mRD),.mWR(mWR),
    .DBDataSrc(DBDataSrc),.InsMemRW(InsMemRW));
    ALU alu(.A(MUXout_rd1_sa),.B(MUXout_rd2_ext),.ALUOp(ALUOp),.sign(sign),.zero(zero),
    .result(ALUresult));
    RegisterFile registerfile(.ReadReg1(IDataOut[25:21]),.ReadReg2(IDataOut[20:16]),
    .WriteReg(MUXout_rt_rd),.WriteData(DB),.CLK(CLK),.WE(RegWre),.ReadData1(ReadData1),
    .ReadData2(ReadData2));
    SignZeroExtend szextend(.ExtSel(ExtSel),.DataIn(IDataOut[15:0]),.DataOut(ExtendOut));
    MUX_2to1_32bits mux_rd1_sa(.SelectSig(ALUSrcA),
    .InputA({24'h000000,3'b000,IDataOut[10:6]}),.InputB(ReadData1),.DataOut(MUXout_rd1_sa));
    MUX_2to1_32bits mux_rd2_ext(.SelectSig(ALUSrcB),.InputA(ExtendOut),
    .InputB(ReadData2),.DataOut(MUXout_rd2_ext));
    AddrAdder extendadder(.Offset({ExtendOut[29:0], 2'b00}),.OriAddr(PC4),
    .OffsetAddr(ExtAddr));
    MUX_4to1_32bits mux_nextpc(.SelectSig(PCSrc),.InputA(PC4),.InputB(ExtAddr),
    .InputC(PseudoAddr),.InputD(32'h00000000),.DataOut(NextPC));
    DataMEM datamem(.DAddr(ALUresult),.DataIn(ReadData2),.CLK(CLK),.RD(mRD),
    .WR(mWR),.DataOut(DataMEMOut));
    MUX_2to1_32bits mux_alu_db(.SelectSig(DBDataSrc),.InputA(DataMEMOut),
    .InputB(ALUresult),.DataOut(DB));
endmodule
```

(9) 部分辅助功能器件如二路选择器、四路选择器、地址加法器、伪地址加法器在这里未列出，可在提交代码对应的 source 文件夹中找出。

六、 实验结果

1. 仿真实验

(1) 测试程序段

地址	汇编程序	指令代码				
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码
0x00000000	addiu $1,$0,8	001001	00000	00001	00000000 00001000	= 24010008
0x00000004	ori $2,$0,2	001101	00000	00010	00000000 00000010	= 34020002
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 100000	= 00411820
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101 00000 100010	= 00622822
0x00000010	and $4,$5,$2	000000	00101	00010	00100 00000 100100	= 00A22024
0x00000014	or $8,$4,$2	000000	00100	00010	01000 00000 100101	= 00824025
0x00000018	sll $8,$8,1	000000	00000	01000	01000 00001 000000	= 00084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	000101	00001	01000	11111111 11111110	= 1428FFFE
0x00000020	slti $6,$2,4	001010	00010	00110	00000000 00000100	= 28460004
0x00000024	slti $7,$6,0	001010	00100	00111	00000000 00000000	= 28870000
0x00000028	addiu $7,$7,8	001001	00111	00111	00000000 00001000	= 24E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	000100	00001	00111	11111111 11111110	= 1027FFFE
0x00000030	sw $2,4($1)	101011	00001	00010	00000000 00000100	= AC220004
0x00000034	lw $9,4($1)	100011	00001	01001	00000000 00000100	= 8C290004
0x00000038	addiu $10,$0,-2	001001	00000	01010	11111111 11111110	= 240AFFFE
0x0000003C	addiu $10,$10,1	001001	01010	01010	00000000 00000001	= 254A0001
0x00000040	bltz $10,-2(<0,转 3c)	000001	01010	00000	11111111 11111110	= 0540FFFE
0x00000044	andi $11,$2,2	001100	00010	01011	00000000 00000010	= 304B0002
0x00000048	j 0x00000050	000010	00000	00000	00000000 00010100	= 08000014
0x0000004C	or $8,$4,$2	000000	00100	00010	01000 00000 100101	= 00824025
0x00000050	halt	111111	00000	00000	0000000000000000	= FC000000

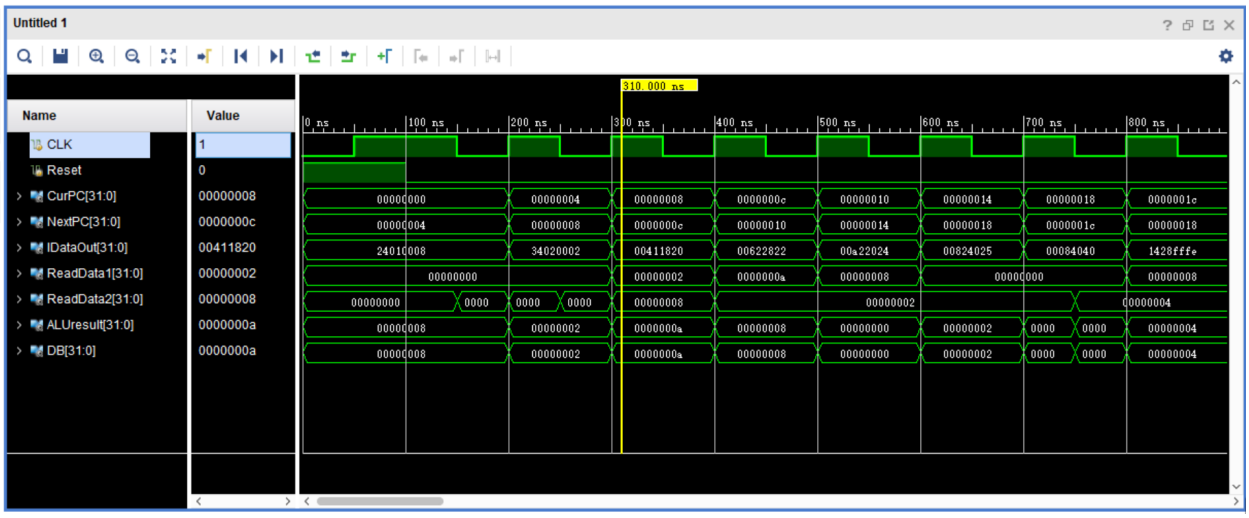
(2) 仿真文件

```
module CPUSim;
    reg CLK;
    reg Reset;
    wire[31:0] CurPC;
    wire[31:0] NextPC;
    wire[31:0] IDataOut;
    wire[31:0] ReadData1;
    wire[31:0] ReadData2;
    wire[31:0] ALUresult;
    wire[31:0] DB;
    MonocylicCPU mcpu(CLK,Reset,CurPC,NextPC,IDataOut,ReadData1,ReadData2,ALUresult,DB);
    initial begin
        CLK = 0;
        Reset = 1; //intitial Reset = 1, and PC = 0X00000000
        #50;
        CLK = !CLK; //negedge
        #50;
        Reset= 0; //clear Reset
```

```
19          forever #50 begin //CLK T = 50
20          CLK = !CLK;
21        end
22    end
23 endmodule
```

(3) 部分仿真实验结果分析

仿真实验结果一

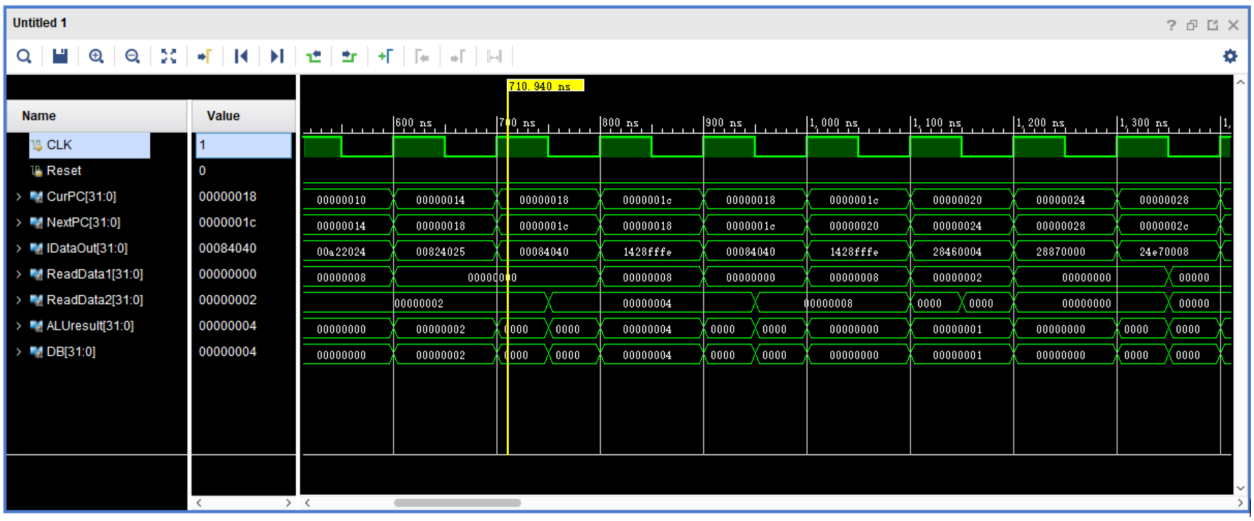


(1) 在 0ns 至 100ns 时，Reset = 1，所以 CurPC(当前 PC 值) 一直为 0X00000000；

(2) 第一个周期到来时，执行指令 **addiu $1,$0,8**，执行立即数加法，将寄存器 $0 的值 (初始化为 0) 与立即数 8 相加的结果存在寄存器 $1 中，图中可以看到此时，ALU 计算结果为 8，DB 总线上的值也为 8；

(3) 当执行地址为 0X00000008 的指令 (即 **add $3,$2,$1**) 时，寄存器堆两个输出分别为 2 和 8(即 $2 和 $1 的值)，相加结果为 0Xa 即十进制中的 10。

仿真实验结果二



(1) 执行地址为 0X14 的指令 (**or $8,$4,$2**) 时，将寄存器 $8 赋值为 2；

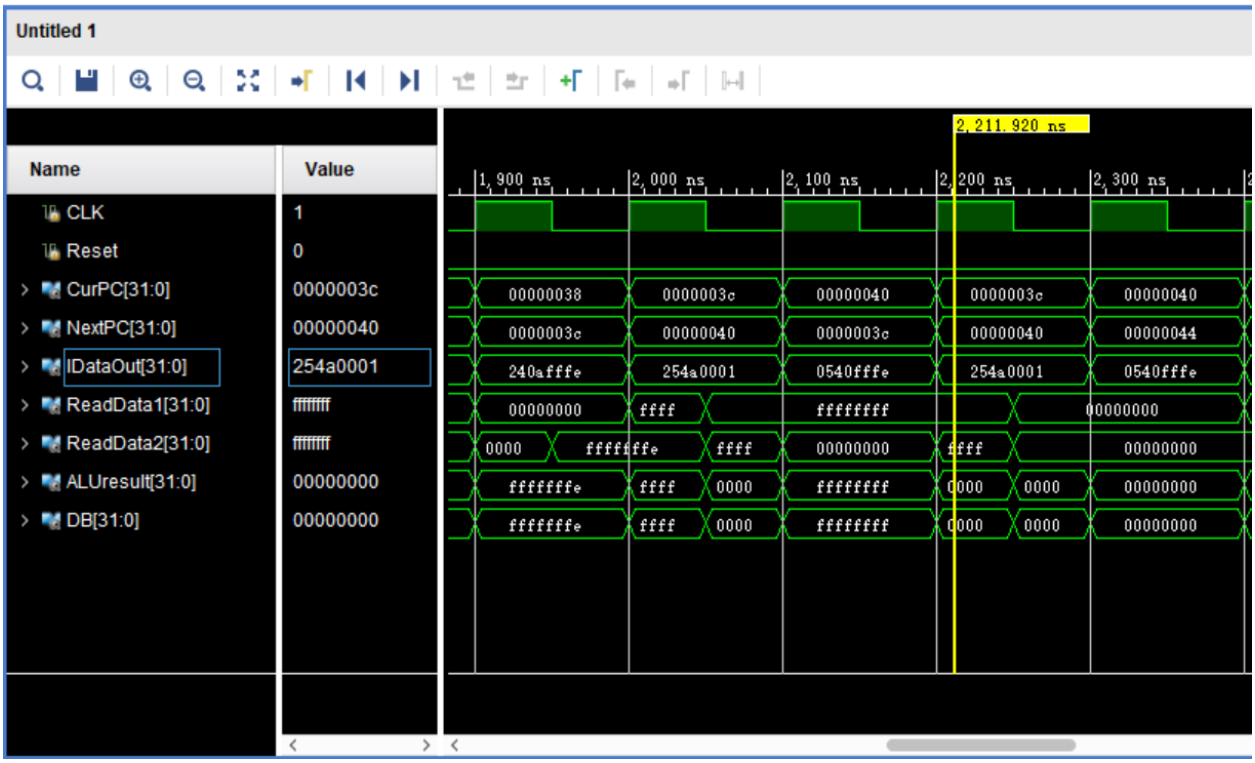
(2) 第一次执行地址为 0X18 的指令 (**or $8,$4,$2**) 时，将寄存器 $8 的值左移一位，即为 4；

(3) 第一次执行地址为 0X1c 的指令 (**bne $8,$1,-2**) 时，将 $1 中的值 (8) 与 $8 中的值 (4) 进行比较，若相等则执行下一条指令，若不等则跳转再次执行 0X18 处的指令，图中可以看出此时的 NextPC 值为 0X18，即进行跳转；

(4) 再次执行地址为 0X18 的指令，将寄存器 $8 的值左移一位，即为 8；

(5) 再次执行地址为 0X1c 的指令，将 $1 中的值 (8) 与 $8 中的值 (8) 进行比较，此时 NextPC 的值为 0X20，即下一条指令。

仿真实验结果三



(1) 执行地址为 0X38 的指令 (**addiu $10,$0,-2**)，将寄存器 $0 的值 (0) 与立即数相加的结果存进寄存器 $10 中，图中可以看出此时 ALU 计算的结果为 0Xfffffffe，即十进制的-2；

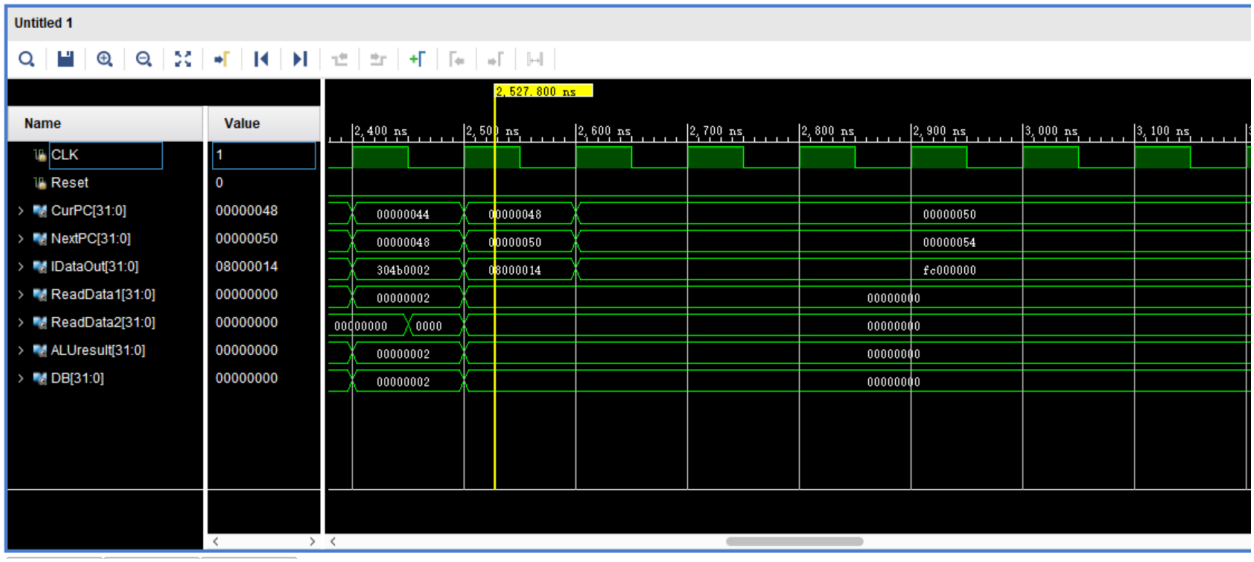
(2) 第一次执行地址为 0X3c 的指令 (**addiu $10,$10,1**)，对寄存器 $10 的值进行立即数 (1) 加法运算，图中可以看出此时 ALU 计算的结果为 0Xffffffff，即十进制的-1；

(3) 第一次执行地址为 0X40 的指令 (**bltz $10,-2**)，将寄存器 $10 中的值 (-1) 与 0 进行比较，若不小于则执行下一条指令，若小于则跳转至地址为 0X3c 处的指令，图中可以看出此时 NextPC 的值为 0X3c，即进行跳转；

(4) 再次执行地址为 0X3c 的指令，对寄存器 $10 的值进行立即数 (1) 加法运算，此时 $10 中的值为 0X0；

(5) 再次执行地址为 0X40 的指令，将 $10 中的值 (0) 与 0 进行比较，此时 NextPC 的值为 0X44，即下一条指令。

仿真实验结果四



(1) 执行地址为 0X44 的指令 (**andi $11,$2,2**)，将寄存器 $2 中的值 (2) 与立即数相与的结果存在寄存器 $11 中，图中可以看出此时 ALU 结果为 2；

(2) 执行地址为 0X48 的指令 (**j 0x00000050**)，无条件跳转至地址为 0X50 处的指令，图中可以看出此时 NextPC 的值为 0X50；

(3) 执行地址为 0X50 的指令 (**halt**)，即停机，PC 的值保持不变，图中可以看出在之后的周期中，PC 的值一直保持为 0X50，不发生改变。

2. 烧写实验

(1) 实验说明

(1) **显示说明**: 指令存储器中的指令地址范围：0~255；数据存储器中的数据地址范围：0~255。也就是只使用低 8 位。

(2) **开关说明**: 开关 SW_in (SW15、SW14) 状态情况如下。显示格式：左边两位数码管 BB : 右边两位数码管 BB。以下是数码管的显示内容。

- SW_in = 00：显示当前 PC 值: 下条指令 PC 值
- SW_in = 01：显示 RS 寄存器地址:RS 寄存器数据
- SW_in = 10：显示 RT 寄存器地址:RT 寄存器数据
- SW_in = 11：显示 ALU 结果输出:DB 总线数据
- 复位信号 (reset) 接开关 SW0，按键 (单脉冲) 接按键 BTNR

(3) **数码管信号说明**：

- 7 段数码管的位控信号 AN3-AN0，每组编码中只有一位为 0(亮) ，其余都是 1(灭)

- 七段数码显示器编码与引脚对应关系为 (左到右，高到低)：七段共阳极数码管 → 1gfedcba; 七段共阴极数码管 → 0gfedcba
- 必须有足够的刷新频率，频率太高或太低都不成，系统时钟必须适当分频，否则效果达不到

(2) 设计思路

(1) 实现 CPU 在板上运行需要两个时钟信号，CPU 工作时钟和 Basys3 板系统时钟。CPU 工作时钟即为按键，是 CPU 正常工作时钟信号，按键必须进行消抖处理；Basys3 板系统时钟即为板提供的正常工作时钟信号，即为 100MHZ。Basys3 板系统时钟信号引脚对应管脚 W5。

(2) 每个按键周期，4 个数码管都必须刷新一次。数码管位控信号 AN3-AN0 是 1110、1101、1011、0111，为 0 时点亮该数码管，当然，还应该为数码管各位“1gfedcba”引脚输出信号，最高位为“1”。比如，“当前 PC 值”低 8 位中的高 4 位和低 4 位，必须经下页转换后送给数码管各引脚。

(3) 显示模块设计大概分为 4 个部分：

a. 对 Basys3 板系统时钟信号进行分频，分频的目的用于计数器；

b. 生成计数器，计数器用于产生 4 个数。这 4 数用于控制 4 个数码管；

c. 根据计数器产生的数生成数码管相应的位控信号（输出）和接收 CPU 来的相应数据；

d. 将从 CPU 接收到的相应数据转换为数码管显示信号，再送往数码管显示 (输出)。

(3) 代码实现

(1) SegLED 模块

七段数码显示器信号转换 (共阳极): 将输入的信号转换为对应的七段数码管亮灭的信号

```
module SegLED(
    input[3:0] Store,
    input Reset,
    output reg[7:0] Out
);
    always@(Store or Reset)begin
        if(Reset==1)begin
            Out= 8'b11111110;
        end
        else begin
            case(Store)
                4'b0000 : Out = 8'b11000000; //0
                4'b0001 : Out = 8'b11111001; //1
                4'b0010 : Out = 8'b10100100; //2
                4'b0011 : Out = 8'b10110000; //3
                4'b0100 : Out = 8'b10011001; //4
                4'b0101 : Out = 8'b10010010; //5
                4'b0110 : Out = 8'b10000010; //6
                4'b0111 : Out = 8'b11011000; //7
                4'b1000 : Out = 8'b10000000; //8
                4'b1001 : Out = 8'b10010000; //9
                4'b1010 : Out = 8'b10001000; //A
                4'b1011 : Out = 8'b10000011; //b
                4'b1100 : Out = 8'b11000110; //C
                4'b1101 : Out = 8'b10100001; //d
                4'b1110 : Out = 8'b10000110; //E
```

```
            4'b1111 : Out = 8'b10001110; //F
        endcase
    end
end
endmodule
```

(2) Debouncer 模块

按键消抖: 消除按下按键时按键会出现人眼无法观测但是系统会检测到的抖动变化。自定义一个 LASTING_TIME，每当检测到 CLK 上升沿到来时检测一次当前电平信号并记录，同时计数器开始计数，只有在计数器达到 LASTING_TIME(即电平信号维持一段自定义的时间) 后，才会对电平信号取反，否则若在计数器达到 LASTING_TIME 前，电平信号发生改变，则将计数器清零，不对电平进行操作。

```
module Debouncer(
    input CLK,
    input key_in,
    output key_out
    );
    parameter LASTING_TIME = 20000;//count if exceed 20000
    reg[21:0] count_low;
    reg[21:0] count_high;
    reg key_out_reg;

    always@(posedge CLK)
    begin
        count_low <= !key_in ? 0 : count_low + 1;
        count_high <= key_in ? 0 : count_high + 1;
        if(count_high == LASTING_TIME)
            key_out_reg <= 1;
        else if(count_low == LASTING_TIME)
            key_out_reg <= 0;
    end

    assign key_out = !key_out_reg;
endmodule
```

(3) Selector 模块

数码管位选择信号生成: 与 Basys3 相应引脚相连接，实现数码管显示位的选择。

```
module Selector(
    input CLK,
    input Reset,
    output reg[3:0] AN
);
parameter SETTIME = 100000;
reg[16:0] counter;

initial begin
    counter <= 0;
    AN      <= 4'b0111;
end

always@(posedge CLK)
```

```
    begin
    if(Reset == 1)
    begin
        counter <= 0;
        AN     <= 4'b0000;
    end else
    begin
        counter <= counter + 1;
        if(counter == SETTIME)
            begin
            counter <= 0;
            case(AN)
                4'b1110:  begin
                    AN <= 4'b1101;
                end
                4'b1101:  begin
                    AN <= 4'b1011;
                end
                4'b1011:  begin
                    AN <= 4'b0111;
                end
                4'b0111:  begin
                    AN <= 4'b1110;
                end
                4'b0000:  begin
                    AN <= 4'b0111;
                end
            endcase
        end
    end
end
endmodule
```

(4) 约束文件

对 Basys3 进行烧板时，需要将信号与引脚一一对应起来，以便使用 Basys3 提供的时钟信号以及使用其按键控制程序的执行

```
set_property IOSTANDARD LVCMOS33 [get_ports {AN[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {AN[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {Out[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports Button]
set_property IOSTANDARD LVCMOS33 [get_ports CLK]
set_property IOSTANDARD LVCMOS33 [get_ports Reset]
set_property PACKAGE_PIN U2 [get_ports {AN[0]}]
set_property PACKAGE_PIN U4 [get_ports {AN[1]}]
set_property PACKAGE_PIN V4 [get_ports {AN[2]}]
```

```
set_property PACKAGE_PIN W4 [get_ports {AN[3]}]
set_property PACKAGE_PIN T1 [get_ports {SW[0]}]
set_property PACKAGE_PIN R2 [get_ports {SW[1]}]
set_property PACKAGE_PIN W7 [get_ports {Out[0]}]
set_property PACKAGE_PIN W6 [get_ports {Out[1]}]
set_property PACKAGE_PIN U8 [get_ports {Out[2]}]
set_property PACKAGE_PIN V8 [get_ports {Out[3]}]
set_property PACKAGE_PIN U5 [get_ports {Out[4]}]
set_property PACKAGE_PIN V5 [get_ports {Out[5]}]
set_property PACKAGE_PIN U7 [get_ports {Out[6]}]
set_property PACKAGE_PIN V7 [get_ports {Out[7]}]
set_property PACKAGE_PIN T17 [get_ports Button]
set_property PACKAGE_PIN W5 [get_ports CLK]
set_property PACKAGE_PIN V17 [get_ports Reset]
```

(5) Basys3 模块

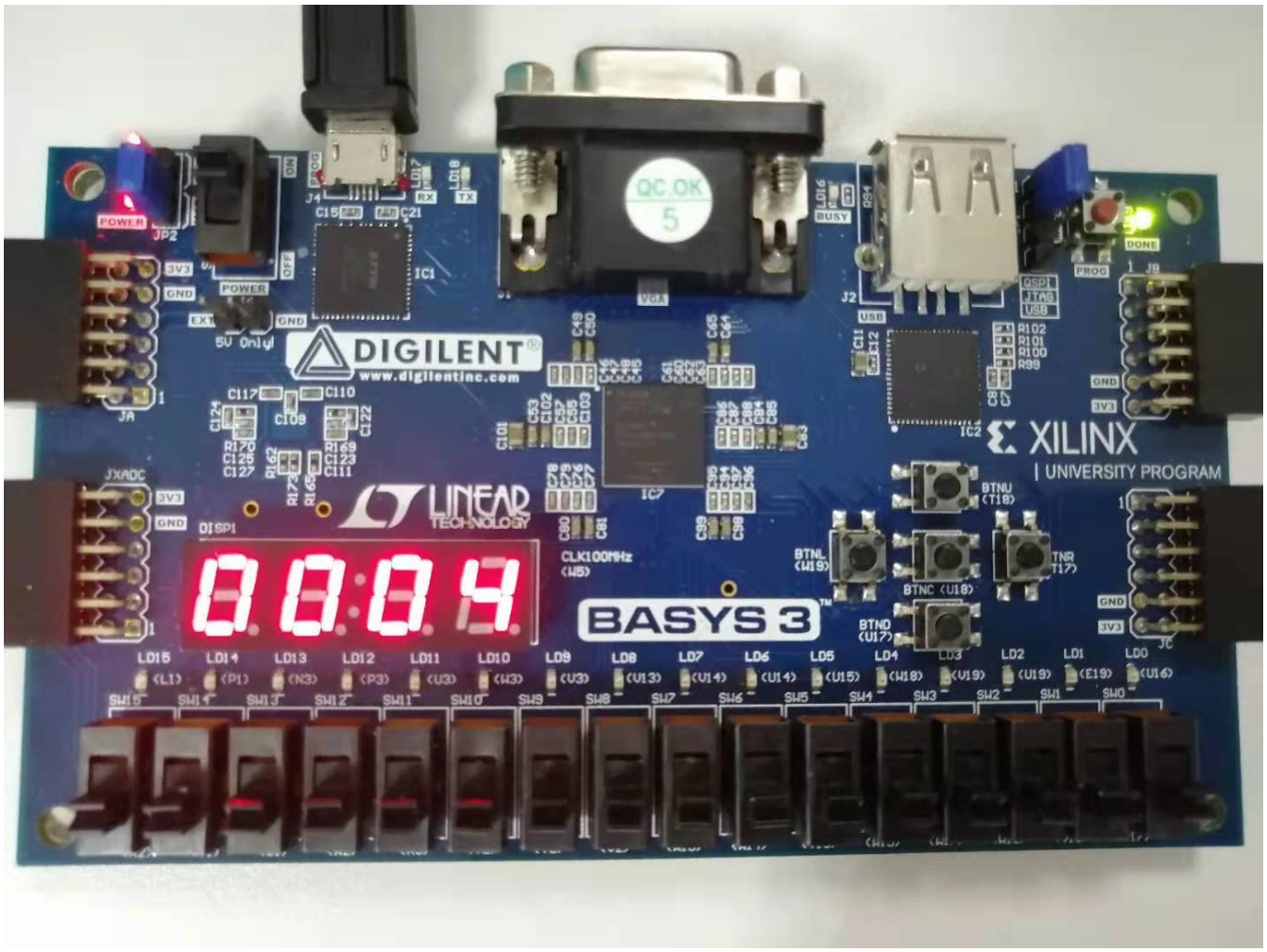
烧板顶层文件: 定义烧板实验中各个子模块的连接方式，通过利用 Debouncer 生成对应的 MyCLK 和 MyReset 信号，消除抖键干扰；利用 Selector 模块以及 SegLED 模块控制四个七段数码管对应位的显示。

```
module Basys3(
    input CLK,
    input[1:0] SW,      //select which signal to show
    input Reset,
    input Button,       //pulse
    output[3:0] AN,     //7seg_led select signal
    output[7:0] Out     //what digits to show
);
    wire[31:0] ALUresult;
    wire[31:0] CurPC;
    wire[31:0] DB;
    wire[31:0] ReadData1;
    wire[31:0] ReadData2;
    wire[31:0] Instruction;
    wire MyCLK;
    wire MyReset;
    reg[3:0] Store ;    //record what to show
    wire[31:0] NextPC;

    //debouncer
    Debouncer CLKDebouncer(CLK,Button,MyCLK);
    Debouncer ResetDebouncer(CLK,Reset,MyReset);
    //cpu
    MonocylicCPU mcpu(MyCLK,MyReset,CurPC,NextPC,Instruction,ReadData1,ReadData2,ALUresult,DB);
    //select signal for 7seg_led
    Selector selector(CLK,MyReset,AN);
    //7seg_led
    SegLED led(Store,MyReset,Out);

    always@(MyCLK)begin
        case(AN)
            4'b1110:  begin
                case(SW)
                    2'b00: Store <= NextPC[3:0];
                    2'b01: Store <= ReadData1[3:0];
                    2'b10: Store <= ReadData2[3:0];
                    2'b11: Store <= DB[3:0];
```

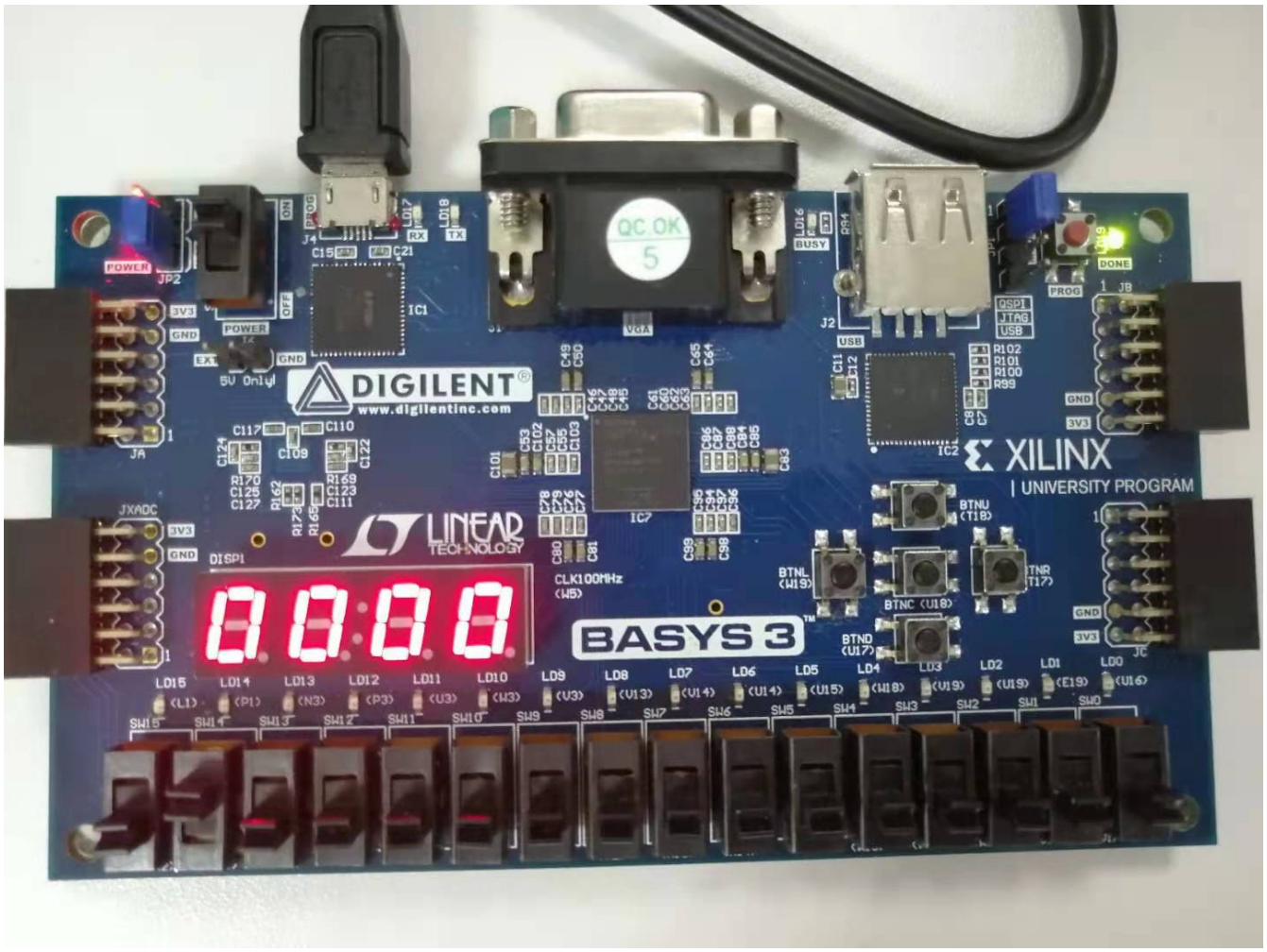
```
            endcase
        end
        4'b1101:  begin
            case(SW)
                2'b00: Store <= NextPC[7:4];
                2'b01: Store <= ReadData1[7:4];
                2'b10: Store <= ReadData2[7:4];
                2'b11: Store <= DB[7:4];
            endcase
        end
        4'b1011:  begin
            case(SW)
                2'b00: Store <= CurPC[3:0];
                2'b01: Store <= Instruction[24:21];
                2'b10: Store <= Instruction[19:16];
                2'b11: Store <= ALUresult[3:0];
            endcase
        end
        4'b0111 : begin
            case(SW)
                2'b00: Store <= CurPC[7:4];
                2'b01: Store <= {3'b000,Instruction[25]};
                2'b10: Store <= {3'b000,Instruction[20]};
                2'b11: Store <= ALUresult[7:4];
            endcase
        end
    endcase
  end
endmodule
```

(4) 部分烧板实验结果分析

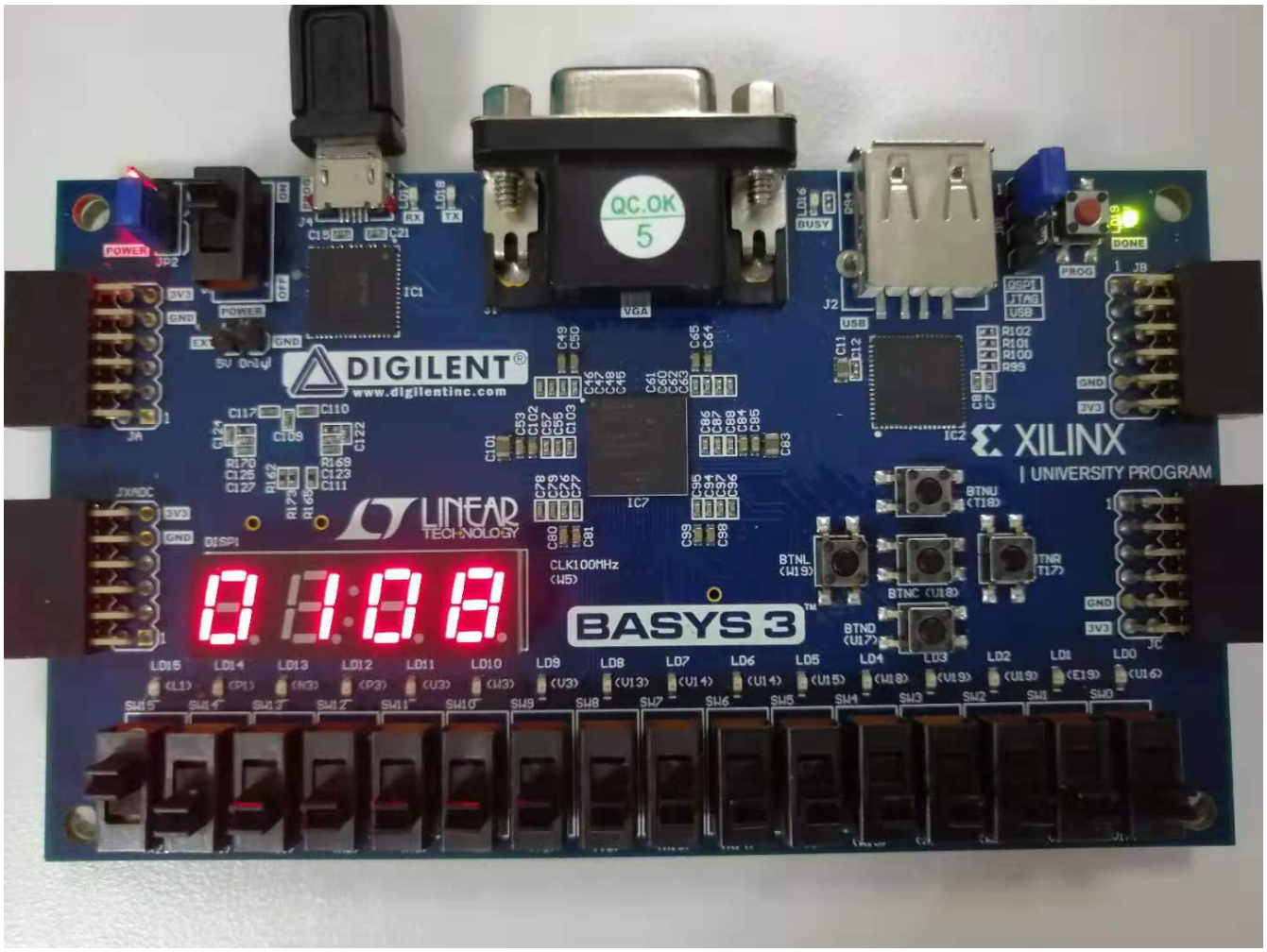
(1) 执行地址为 0X00 的指令 (**addiu $1,$0,8**)



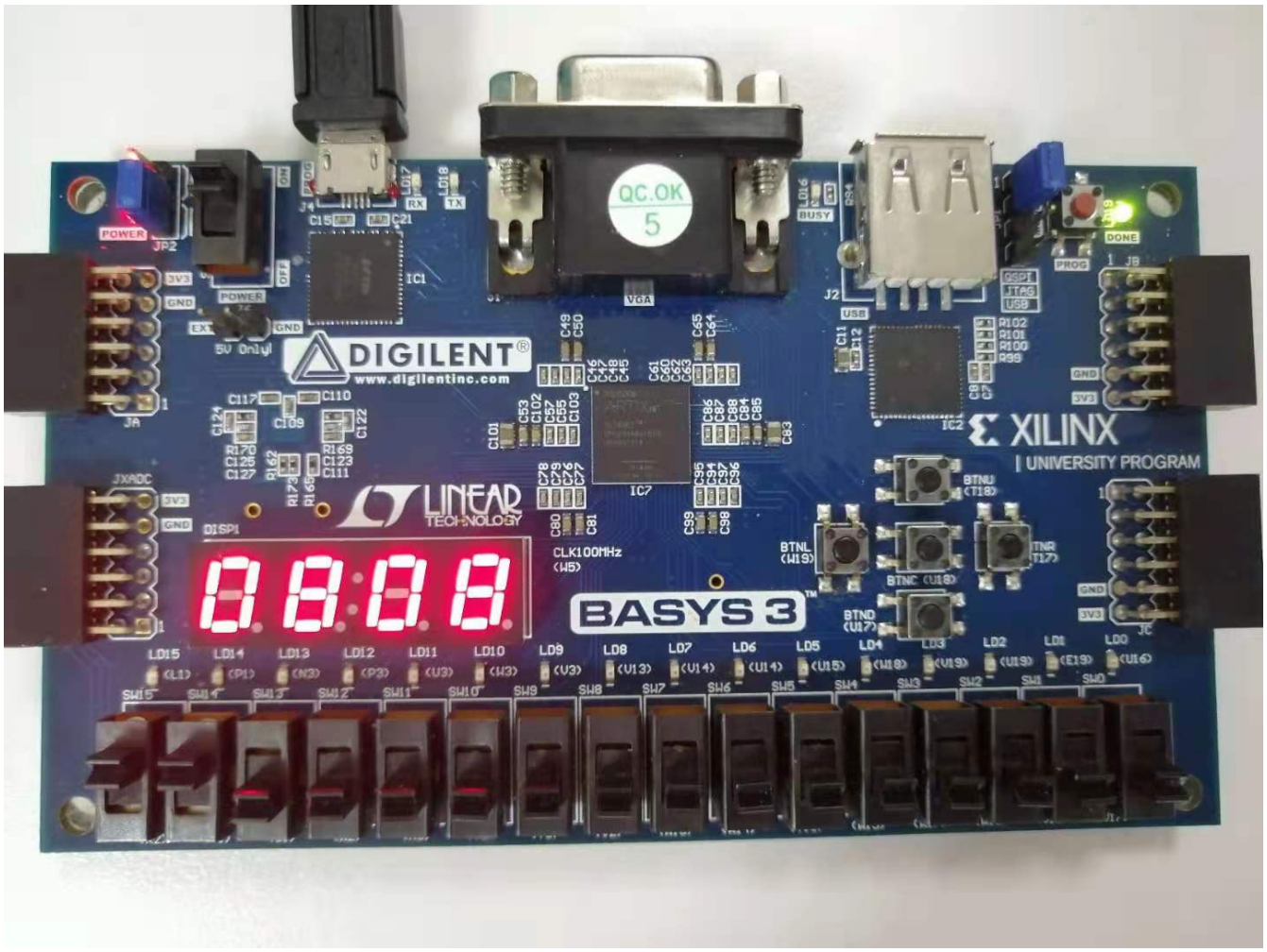
此时，SW = 00，数码管显示为：当前 PC 值: 下条指令 PC 值，即 0X00 和 0X04



此时，SW = 01，数码管显示为：RS 寄存器地址:RS 寄存器数据，即寄存器 $0 的值为 0

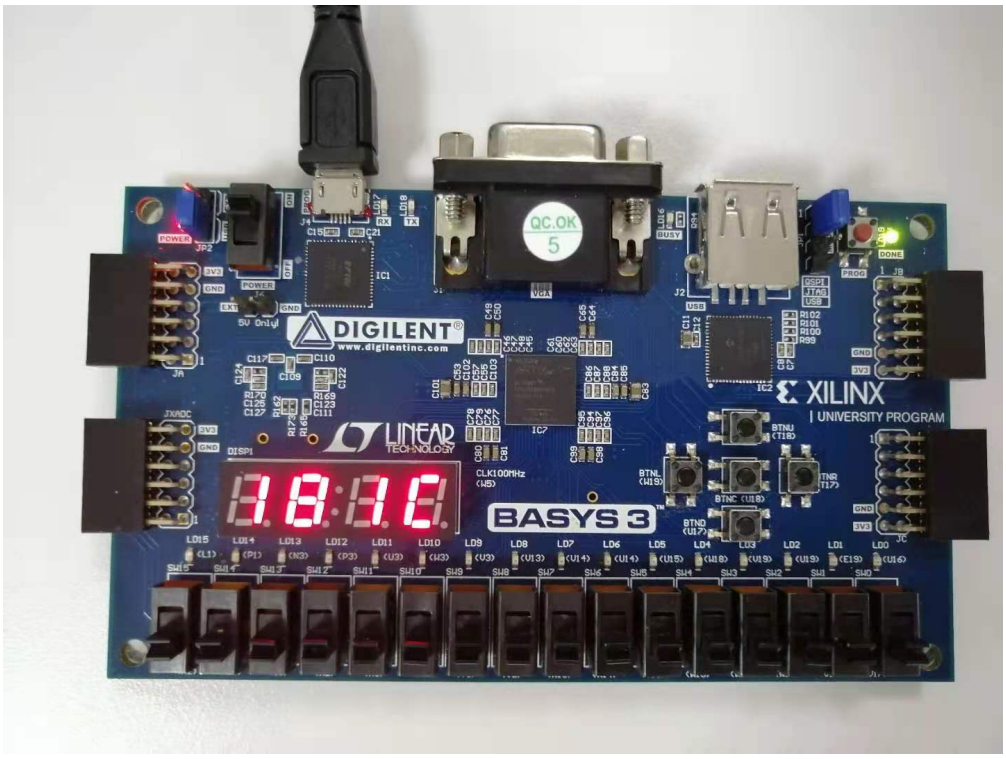


此时，SW = 10，数码管显示为：RT 寄存器地址:RT 寄存器数据，即寄存器 $1 的值为 8

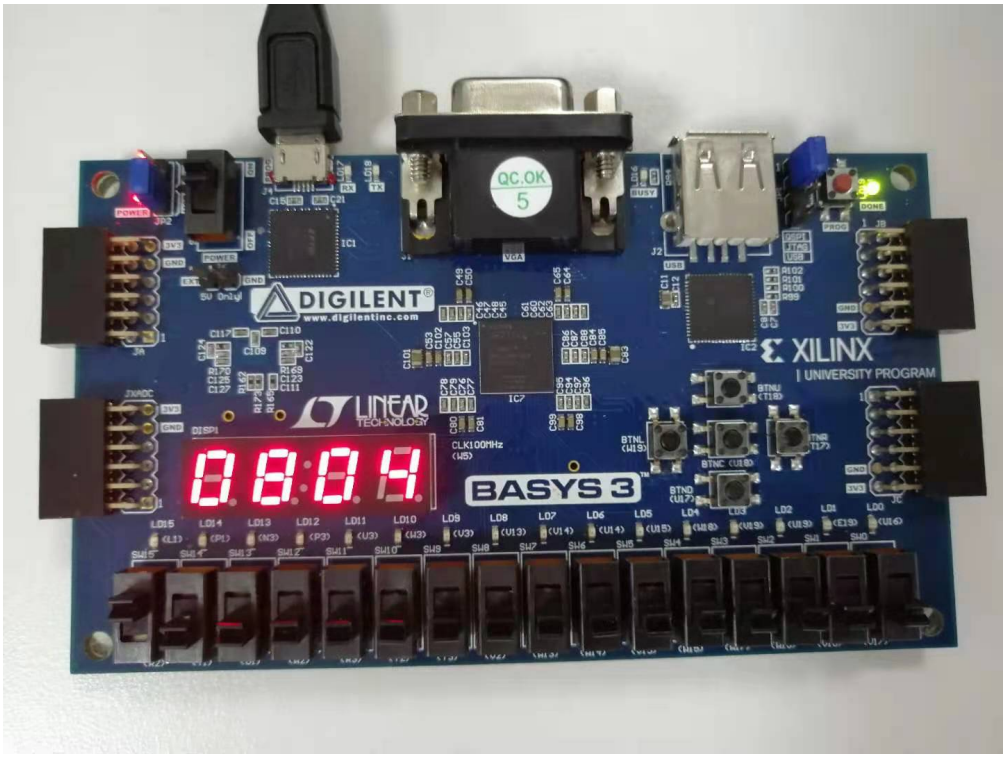


此时，SW = 11，数码管显示为：ALU 结果输出:DB 总线数据，即 ALU 计算结果为 8，数据总线上的值为 8

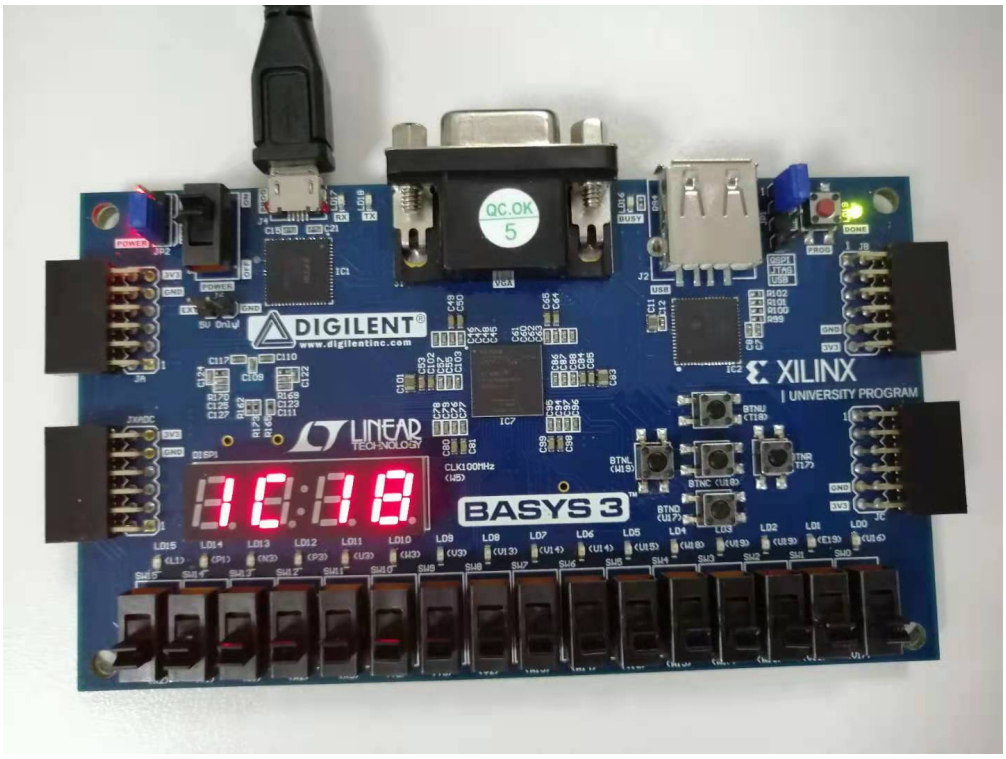
(2) 执行地址为 0X18 以及 0x1c 的指令



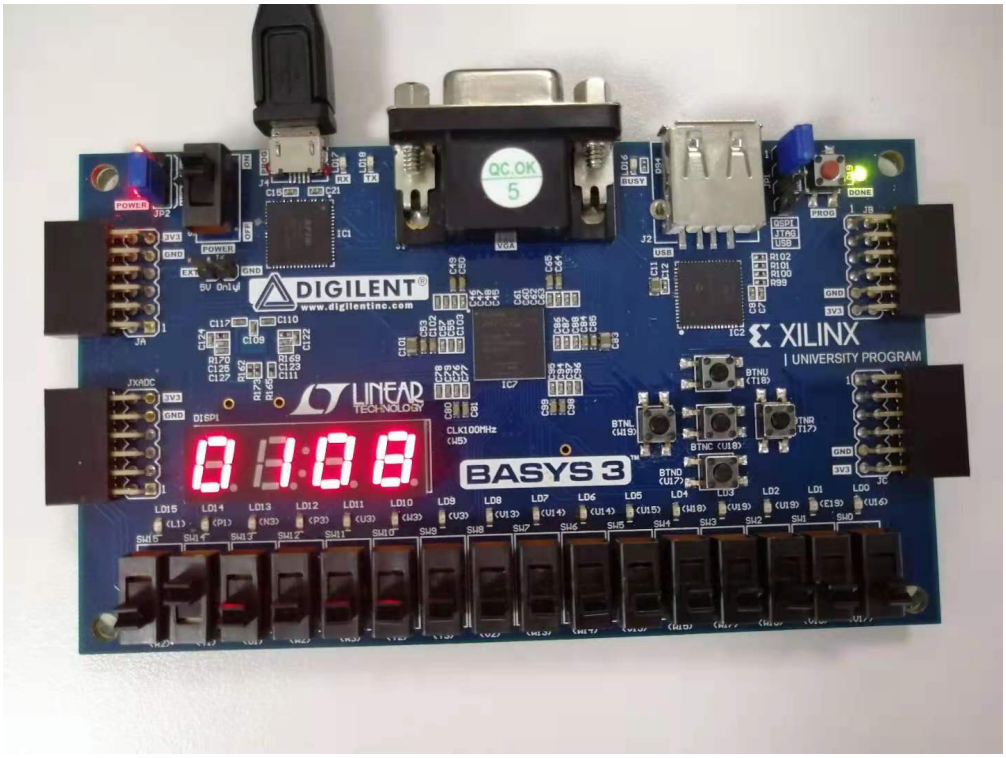
第一次执行地址为 0X18 的指令 (**sll $8,$8,1**)，此时，SW = 00，数码管显示为：当前 PC 值: 下条指令 PC 值，即 0X18 和 0X1c



此时，SW = 10，数码管显示为：RT 寄存器地址:RT 寄存器数据，即寄存器 $8 的值为 4

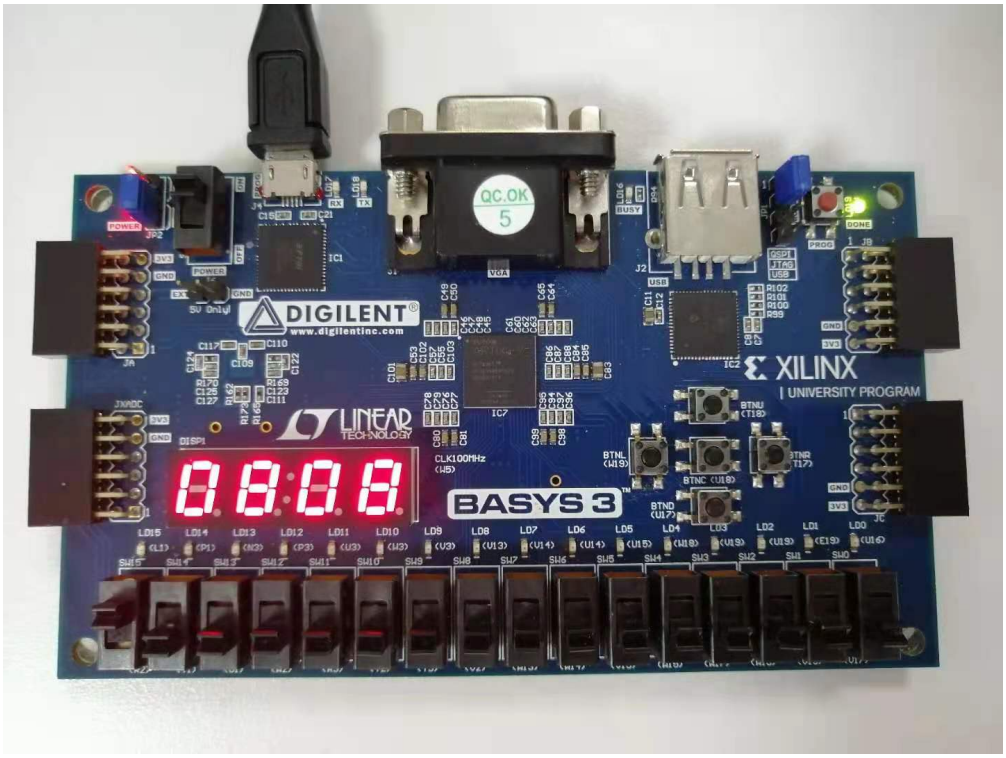


第一次执行地址为 0X1c 的指令 (**bne $8,$1,-2**)，此时，SW = 00，数码管显示为：当前 PC 值:下条指令 PC 值，即 0X1c 和 0X18

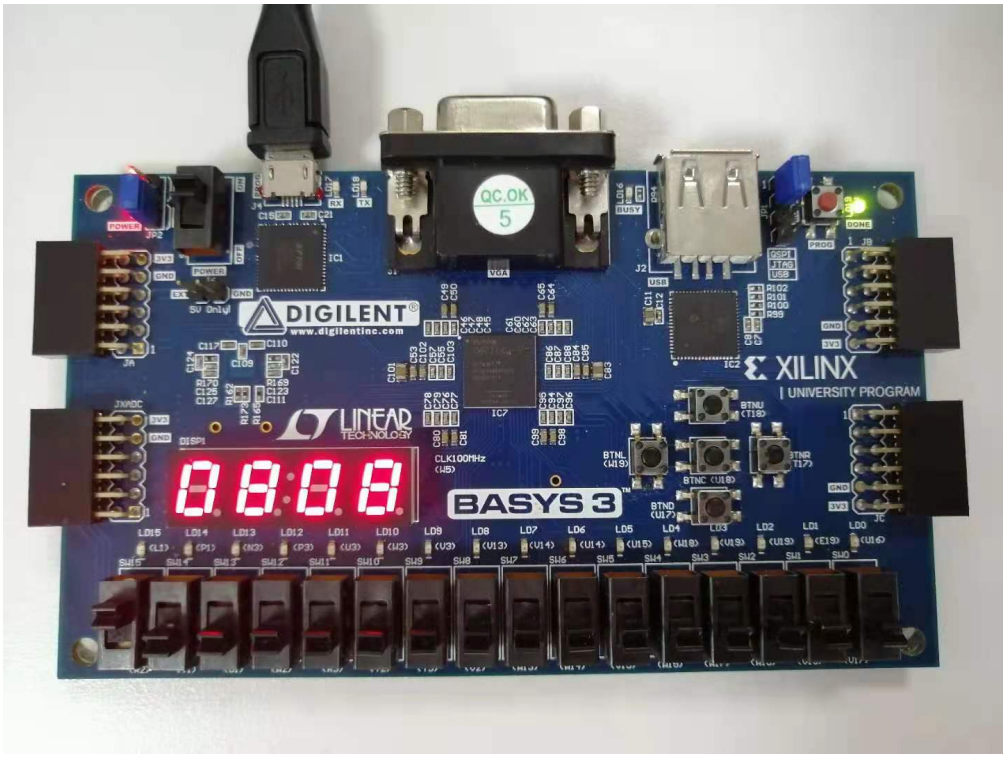


此时，SW = 01，数码管显示为：RS 寄存器地址:RS 寄存器数据，即寄存器 $1 的值为 8，所以

下一条指令地址应为 0X18

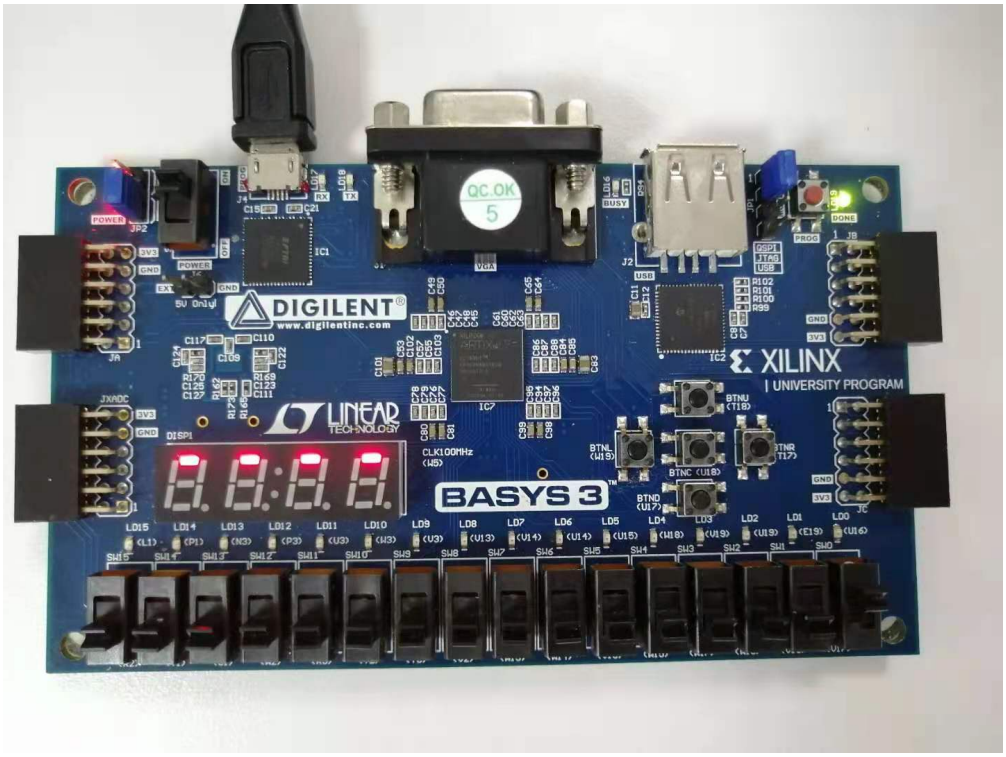


再次执行地址为 0X18 的指令 (**sll $8,$8,1**)，此时，SW = 01，数码管显示为：RS 寄存器地址:RS 寄存器数据，寄存器 $8 的值左移一位，即为 8

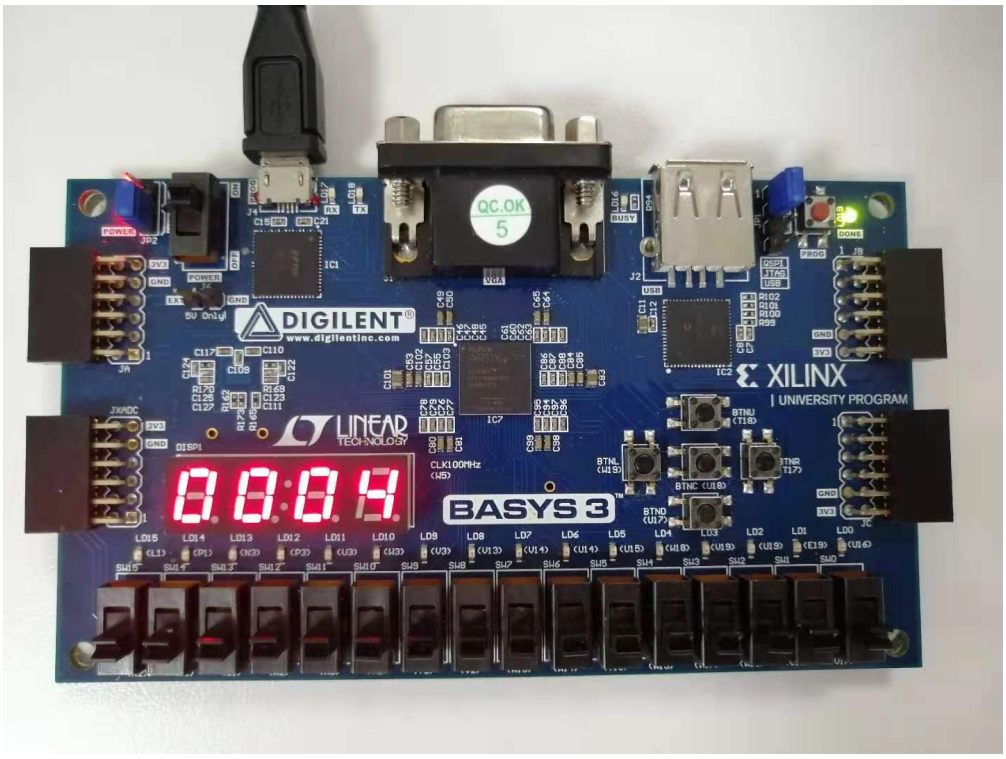


再次执行地址为 0X1c 的指令 (**bne $8,$1,-2**)，此时，SW = 00，数码管显示为：当前 PC 值: 下条指令 PC 值，因为寄存器 $8 与寄存器 $1 的值均为 8，所以下一条地址为 0X20

(3) Reset 信号测试



在执行上述指令的基础上，打开端口数位 V17 的开关，使 Reset 信号为高，此时程序暂停执行，显示自定义的 Reset 状态



关闭端口数位 V17 的开关，使 Reset 信号为低，此时程序回复到开始时候状态，SW = 00，数码管显示为：当前 PC 值: 下条指令 PC 值，即 0X00 和 0X04

(4) 其余结果因报告篇幅有限，其余实验结果与课上展示给助教，这里不作展示。

七、 实验心得

本次实验可以说是几年大学学习以来最令我崩溃的实验 (虽然感觉下一个流水线 cpu 会更令我崩溃)，从看明白实验的要求，再着手设计，再开始踩 verilog 语法上的坑，接着看资料烧板，每一步对于我来说都是不小的挑战，总的来说最困难的还是 verilog 语法上的问题，特此记录一下

(1) 编写代码时，以为 verilog 的语法和 c 非常的相像，于是写出了类似 assign signal = !signal 的语句，然后 vivado 就开始报错说程序中某个信号有双驱动，然后查看顶层文件也没有发现双驱动的情况，后来利用 vivado 中生成可视电路的功能才发现，就是上述的语句出现了问题，在 verilog 中 assign 赋值相当于在器件间相互连接，这里的语句相当于把 signal 取反再次连接上 signal，因此产生双驱动的问题，解决方法是多声明一个 wire negsignal 变量，再 assign negsignal = !signal;

(2) 编写代码时，因为粗心，在声明变量时声明方式为 wire RW，但实际使用时，却把 RW 写成 Rw，但是 vivado 在编译时并没有报错，而是直接忽略继续执行，而原来的 RW 则一直维持在高阻态，所以后来的信号都无法发生改变，不太理解为什么 vivado 对于这种行为并没有采取更严格的处理，找这个 bug 的时候真的抓狂;

(3) 在烧板的时候，因为缺少烧板的经验 (因为疫情原因我们的数字电路课程并没有进行过电路板上的实验)，不知道 sim 文件与 design 中的顶层文件与 synthesis 的关系，导致没办法得出正确的结果，后来寻找高年级同学问他们拿到了 Basys3 走马灯实验的资料认真地看了一遍才明白其中的关系，所以说计组实验真的要有一些前导实验做基础才能少走弯路;

(4) 在实验的过程中还遇到了诸如在 run simulation 时需要添加指令-mode out_of_context，而在 run synthesis 时需要再次删掉的一些直至目前解决了但并没有完全理解的问题，幸好在课程群中有助教和同学们的帮助才能更快的解决。

总的来说，从这次实验中，我了解到了模块化编程带来的易于 debug 与分析的优点，以及 verilog 语法上的一些坑，更把之前数字电路课程上缺失的烧板经验狠狠地补了回来，最重要的是明白了硬件设计过程的困难之处，希望这些经验能使我在下一个流水线 cpu 实验中更容易地发现错误以及有一个更平和的心态进行实验，真的这次实验做到最后情绪起伏有点太大了，其实大家都是在这个坑里面慢慢摸索的，慢慢探索总能找到答案的。