湖南大學

简易计算机系统

综合设计报告

题　　目：	简易计算机系统
学生姓名：	刘大卫
学生学号：	201826010215
专业班级：	软件 1802
完成时间：	2019.12.20

一、 设计目的

完整、连贯地运用《数字逻辑》所学到的知识，熟练掌握 EDA 工具基本使用方法，为 学习好后续《计算机原理》课程做铺垫。

二、 设计内容

（一） 按照给定的数据通路、数据格式和指令系统，使用 EDA 工具设计一台用硬连线逻辑控制的简易计算机；

（二） 要求灵活运用各方面知识，使得所设计的计算机具有较佳的性能；

（三） 对所设计计算机的性能指标进行分析，整理出设计报告。

三、 详细设计

（一） 设计的整体架构

表 1 指 令 系 统 表

汇编符号	功能	编码
MOV R1，R2	（R2）→ R1	1111 R1 R2
MOV M，R2	（R2）→（C）	1111 11 R2
MOV R1，M	((C)) →R1	1111 R1 11
ADD R1，R2	（R1）+（R2）→ R1	1001 R1 R2
SUB R1，R2	（R1）-（R2）→ R1	0110 R1 R2
OR R1，R2	（R1）∨（R2）→ R1	1011 R1 R2
NOT R1	/（R1）→ R1	0101 R1 XX
RSR R1	（R1）循环右移一位→ R1	1010 R1 00
RSL R1	（R1）循环左移一位→ R1	1010 R1 11
JMP add	add → PC	0001 00 00，address
JZ add	结果为 0 时 add → PC	0001 00 01，address
JC add	结果有进位时 add → PC	0001 00 10，address
IN R1	（开关 7-0）→ R1	0010 R1 XX
OUT R1	（R1）→ 发光二极管 7-0	0100 R1 XX
NOP	（PC）+1 → PC	0111 00 00
HALT	停机	1000 00 00

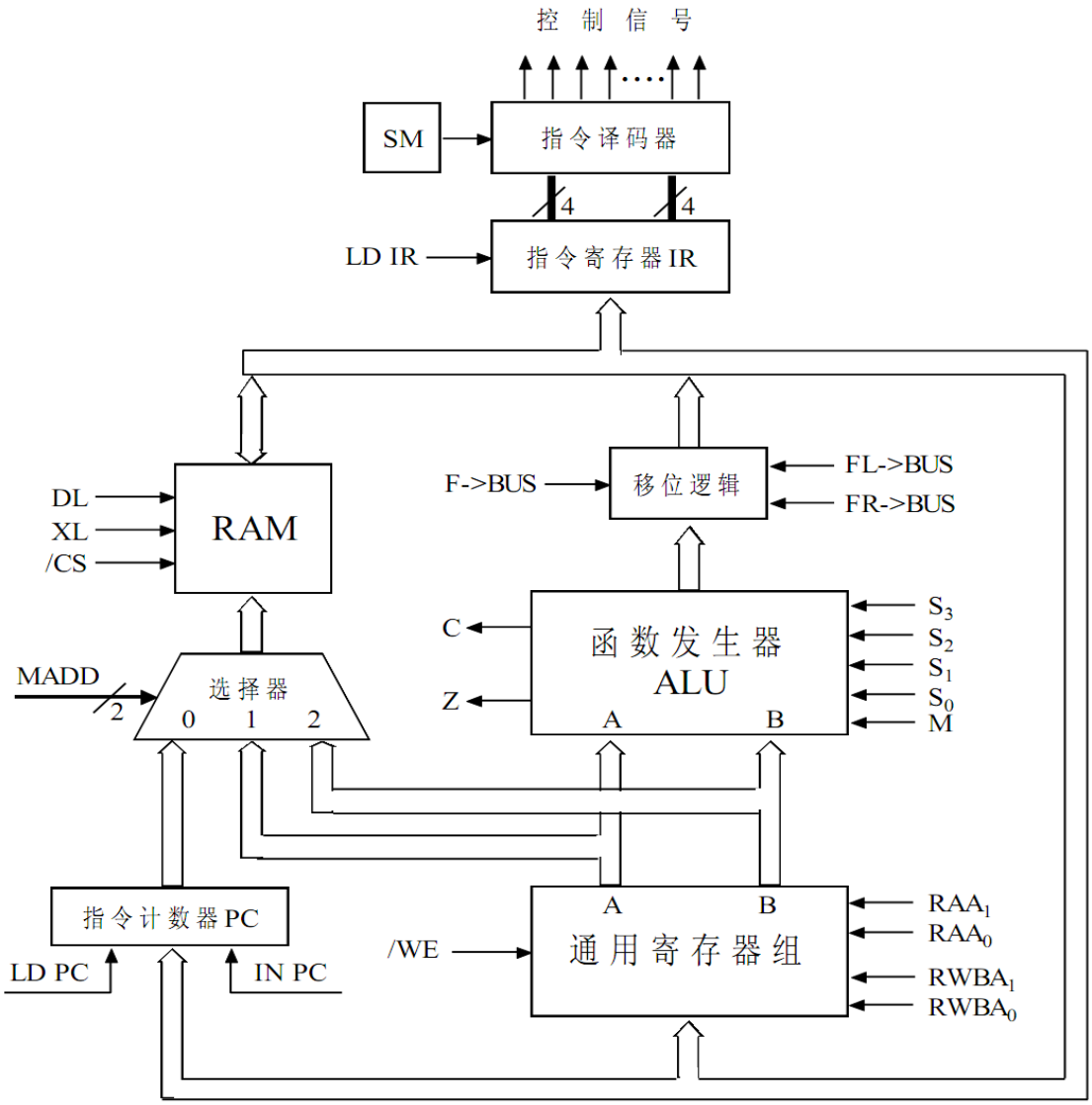


表 2　基本控制信号及功能表

序号	信号	功能
1	IN PC	与 LD PC 配合使用，为 1 时 PC 加 1 计数，为 0 时加载 BUS 上的数据。
2	LD PC	当 IN PC=1 允许对 PC 加 1 计数，否则允许把 BUS 上的数据打入 PC。
3	LD IR	允许把 BUS 上的数据打入指令寄存器 IR。
4	/WE	允许把 BUS 上的数据打入通用寄存器组，低电平有效。
5	F→BUS	ALU 的运算结果通过移位门直接送到总线 BUS 的对应位。
6	FL→BUS	ALU 的运算结果通过移位门左移一位送到总线 BUS，且 F7 送 C_f。
7	FR→BUS	ALU 的运算结果通过移位门右移一位送到总线 BUS，且 F0 送 C_f。
8	/CS	允许访问存储器，低电平有效。
9～10	MADD	存储器 RAM 地址来源。0：指令计数器，1：通用寄存器 A 口，2：B 口。
11	DL	读存储器 RAM。
12	XL	写存储器 RAM。
13	M	M=1，表示 ALU 进行逻辑运算操作。
14～17	S_3～S_0	使 ALU 执行各种运算的控制位。
18	HALT	此位为“1”时停机，下次输入人工操作。

设计的大体架构与上述图片中的大体相同。只是在少数的部件中进行了修改使之可以在学习板上运行。具体的修改可见各模块具体实现。

（二） 各模块具体实现

1. 寄存器组+ALU+移位逻辑

1） 寄存器组

我们首先对寄存器进行设计，其接口设计、功能实现和仿真如下：

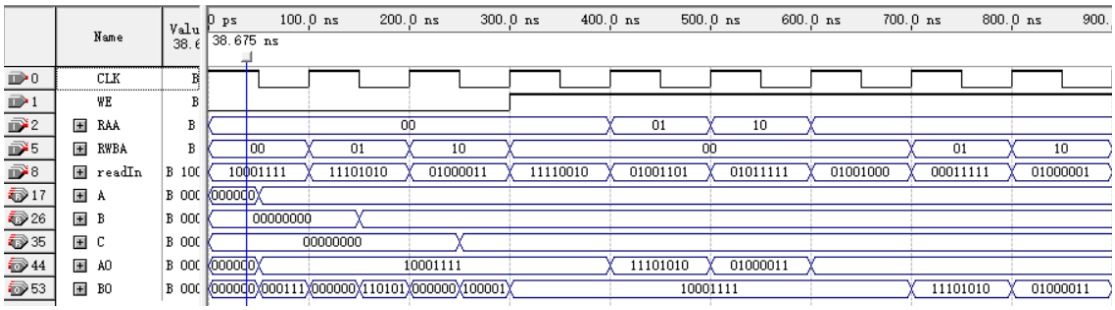
```
entity regGroup is
    port(
        CLK, WE : in std_logic;
        RAA, RWBA : in std_logic_vector(1 downto 0);
        readIn : in std_logic_vector(7 downto 0);
        AO, BO : out std_logic_vector(7 downto 0)
    );
end regGroup;
```

CLK 为时钟信号；WE 为写控制信号；RAA、RWBA 为输出控制信号，同时 RWBA 还为写入地址控制信号；readIn 为输入的数据；AO、BO 为输出信号。

```
architecture behave of regGroup is
signal A, B, C : std_logic_vector(7 downto 0);
begin
    process(CLK, WE, RAA, RWBA, readIn)
    begin
        -- RAA inout, AO output
        case RAA is
            when "00" =>
                AO <= A;
            when "01" =>
                AO <= B;
            when "10" =>
                AO <= C;
            when others =>
                NULL;
        end case;
        -- RWBA inout, BO output
        case RWBA is
            when "00" =>
                BO <= A;
            when "01" =>
                BO <= B;
            when "10" =>
                BO <= C;
            when others =>
                NULL;
        end case;
        -- write in it
        if WE = '0' and (CLK'event and CLK = '0') then
            case RWBA is
                when "00" =>
                    A <= readIn;
                when "01" =>
                    B <= readIn;
                when "10" =>
                    C <= readIn;
                when others =>
                    NULL;
            end case;
        end if;
    end process;
end;
```

功能设计如上，通过 RAA、RWBA 控制输出，当写入控制信号为 1 并且为时钟

下降沿时，进行写入操作，写入地址为 RWBA 指定的地址。



仿真结果如上，完成预计的指令。

2） ALU

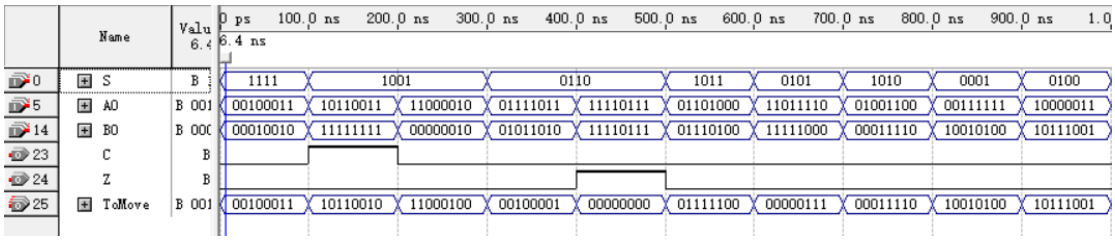
接下来进行 ALU 的设计，其接口设计、功能实现和仿真如下：

```
entity ALU is
    port(
        S : in std_logic_vector(3 downto 0);
        AO, BO : in std_logic_vector(7 downto 0);
        C, Z : out std_logic;
        ToMove : out std_logic_vector(7 downto 0)
    );
end ALU;
```

S 为模式控制信号，直接输入为 4 位命令码；AO、BO 为来自寄存器组的数据输入；C、Z 为 ADD 与 SUB 命令的状态位，有进位 C 为 1，相减为 0，Z 为 1；ToMove 为 ALU 的输出信号，将运算完成的数据进行传输，或者进行直传。

```
architecture behanve of ALU is
signal dataA, dataB, dataBus : std_logic_vector(8 downto 0);
signal tempc, tempz : std_logic;
begin
    dataA <= '0' & AO;
    dataB <= '0' & BO;
    process(S, AO, BO)
    begin
        if S = "1111" then -- MOVE
            dataBus <= dataA;
        elsif S = "1001" then -- ADD
            dataBus <= dataA + dataB;
            tempc <= dataBus(8);
        elsif S = "0110" then -- SUB
            dataBus <= dataB - dataA;
            if dataBus = "000000000" then
                tempz <= '1';
            else
                tempz <= '0';
            end if;
        elsif S = "1011" then -- OR
            dataBus <= dataA or dataB;
            if dataBus = "000000000" then
                tempz <= '1';
            else
                tempz <= '0';
            end if;
        elsif S = "0101" then -- NOT
            dataBus <= not dataB;
            if dataBus = "000000000" then
                tempz <= '1';
            else
                tempz <= '0';
            end if;
        else
            dataBus <= dataB;
        end if;
    end process;
    C <= tempc;
    Z <= tempz;
    ToMove <= dataBus(7 downto 0);
end;
```

为 MOV 指令时，将 AO 口的值直传；为 ADD、SUB、OR、NOT 指令时进行运算后传输；其他指令时，将 BO 口的值进行直传。



MOV 指令时将 AO 进行直传；ADD、SUB、OR、NOT 时进行运算，其他情况将 BO 进行直传。

3） 移位部分

最后进行移位逻辑部分的设计，其接口设计、功能实现和仿真如下：

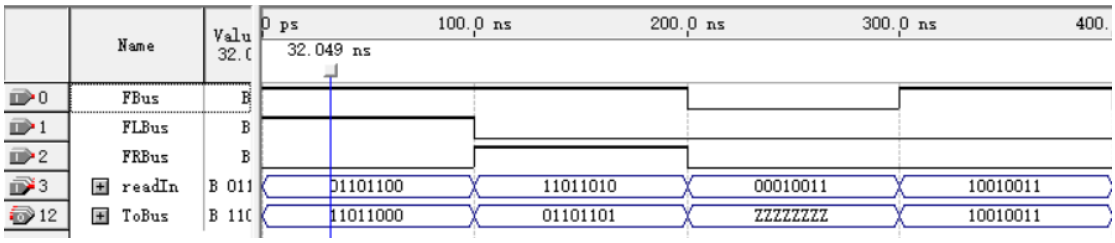
```
entity Move is
    port(
        FBus, FLBus, FRBus : in std_logic;
        C : out std_logic;
        readIn : in std_logic_vector(7 downto 0);
        ToBus : out std_logic_vector(7 downto 0)
    );
end Move;
```

FBus 控制是否可以进行传输，为 0 是，ToBus 为高阻态；FLBus 为向左循环移位的控制信号；FRBus 为向右循环移位的控制信号；readIn 为输入的数据；C 为输出移出位的端口；ToBus 为输出到总线的数据。

```
architecture behave of Move is
signal tempc : std_logic;
begin
    process(Fbus, FLBus, FRBus, readIn)
    begin
        if FBus = '0' then
            ToBus <= "ZZZZZZZZ";
        else
            if FLBus = '1' then
                ToBus <= readIn(6 downto 0) & readIn(7);
                tempc <= readIn(7);
            elsif FRBus = '1' then
                ToBus <= readIn(0) & readIn(7 downto 1);
                tempc <= readIn(0);
            else
                ToBus <= readIn;
            end if;
        end if;
        C <= tempc;
    end process;
end;
```

这是实现的代码，与接口设计相似不再进行解释。特别注意的是，在 VHDL 语

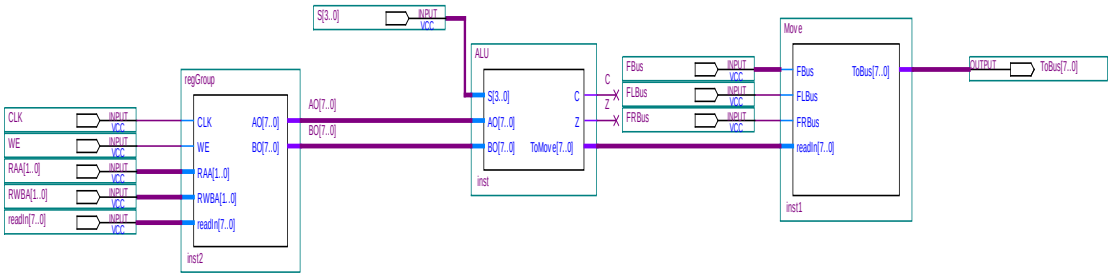
言中 Z 为高阻态。



第一个网格中，进行左移位的逻辑，成功实现；第二个进行右移位逻辑，成功实现；第三个网格 FBus 为 0 输出高阻态；第四个网格直接进行输出。

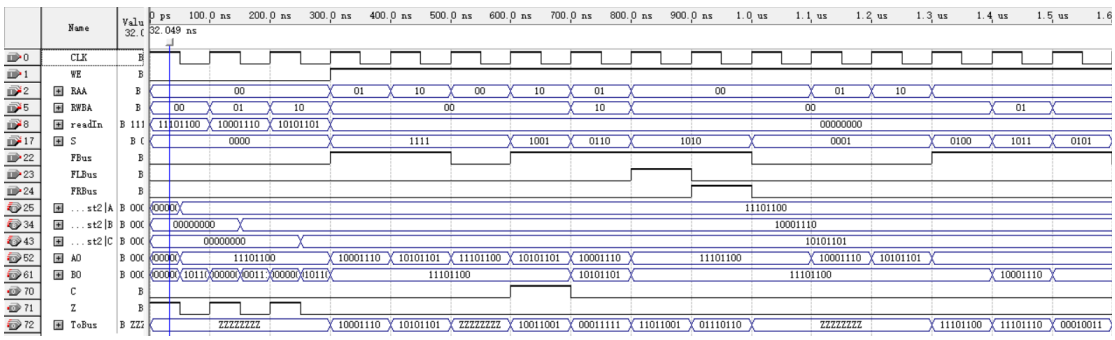
4） 联调

接下来进行上述三个部件的联调，其接口设计、功能实现和仿真如下：



这是 FuncPart 部分的实现，先行将寄存器组、ALU、移位逻辑器进行组合，简化顶层图的设计难度。

需要加载的外部输入有 CLK（时钟）、WE（写入控制信号）、RAA、RWBA（寄存器选择信号）、readIn（数据输入）、S（模式信号）、FBus、FLBus 和 FRBus（移位逻辑控制信号），给外部的输出有 AO、BO（给三路选择器）、C、Z（用 D 触发器存储）、ToBus（给总线的输出）。



依次进行三次置数；接着对 MOVA、MOVB、MOVC 等命令进行验证，均得到了预期的结果。

至此，FuncPart 部分完成。各种功能和控制信号通过验证。

2. PC+多路选择器+RAM

1） PC 计数器

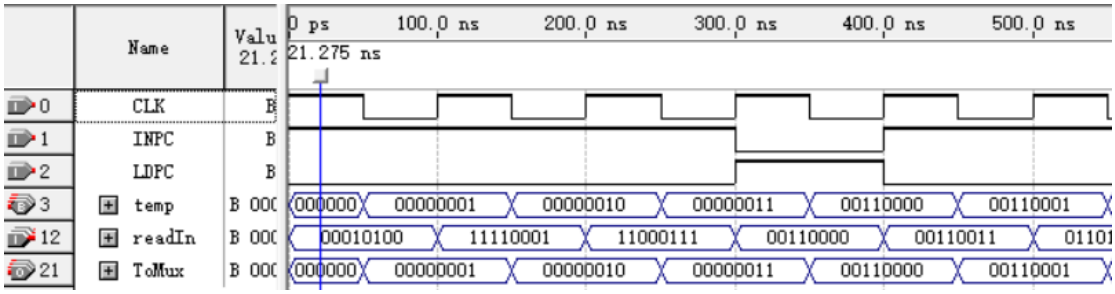
我们首先对 PC 计数器进行设计，其接口设计、功能实现和仿真如下：

```
entity PC is
    port(
        CLK, INPC, LDPC : in std_logic;
        readIn : in std_logic_vector(7 downto 0);
        ToMux : out std_logic_vector(7 downto 0)
    );
end PC;
```

CLK 为时钟输入信号；INPC、LDPC 为模式控制信号；readIn 为外部输入信号；ToMux 输出给多路选择器。

```
architecture behave of PC is
signal temp : std_logic_vector(7 downto 0) := "00000000";
begin
    process(CLK, INPC, LDPC, readIn)
    begin
        if CLK'event and CLK = '0' then
            if INPC = '1' and LDPC = '0' then
                temp <= temp + 1;
            elsif INPC = '0' and LDPC = '1' then
                temp <= readIn;
            end if;
        end if;
    end process;
    ToMux <= temp;
end;
```

当时钟下降沿时执行指令，INPC 为 1，LDPC 为 0 时，执行加一操作；INPC 为 0，LDPC 为 1 时，执行外部数据输入操作；



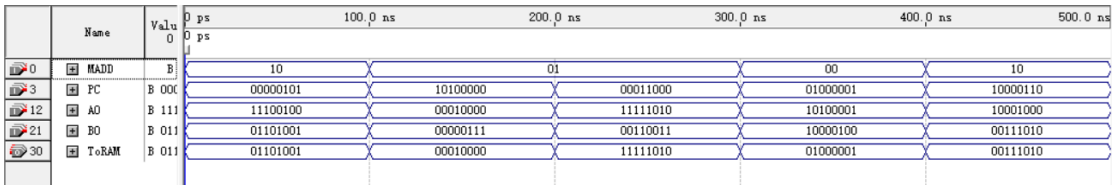
1-3 时钟周期执行加一操作，可以看到得到了预期的结果；第四个时钟周期，执行置数操作可以看到，得到预期结果；第五个时钟周期仍然执行加一的操作。

2） 多路选择器

接下来进行多路选择器的设计，其接口设计、功能实现和仿真如下：

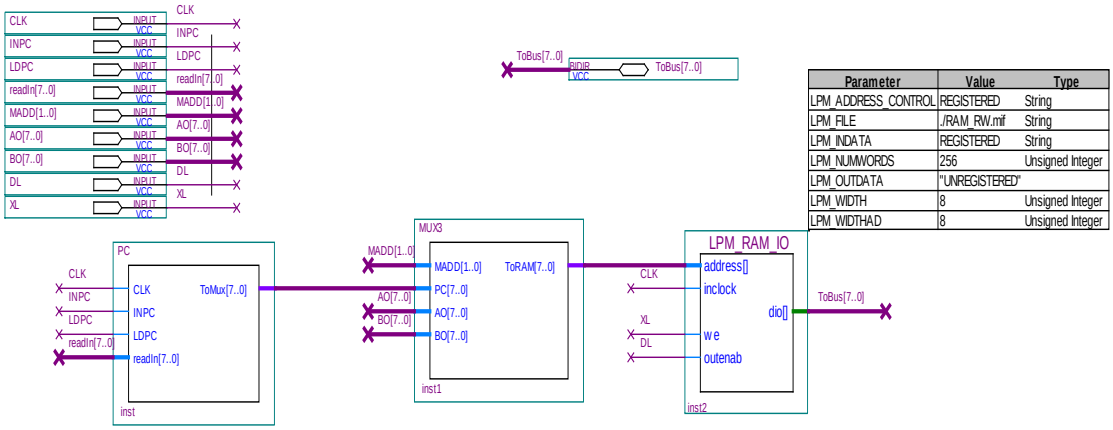
```
-- MUX
library ieee;
use ieee.std_logic_1164.all;

entity MUX3 is
    port(
        MADD : in std_logic_vector(1 downto 0);
        PC, AO, BO : in std_logic_vector(7 downto 0);
        ToRAM : out std_logic_vector(7 downto 0)
    );
end MUX3;

architecture behave of MUX3 is
begin
    ToRAM <= PC when MADD = "00" else
             AO when MADD = "01" else
             BO;
end;
```

简单的多路选择器，MADD 为 00 时，选择 PC；MADD 为 01 时，选择 AO；MADD 为 10 时，选择 BO。

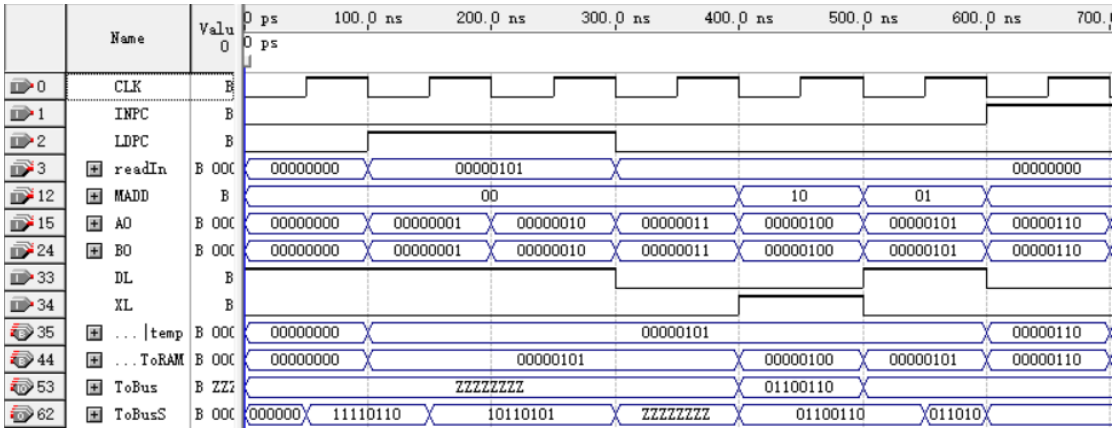


从图中可以看出我们完整的实现了功能，比较简单，不再进行说明。

3） RAM 以及联调



完成 RAM 部分的整体连接工作，接受的外部输入有 CLK 时钟；INPC、LDPC 计数器的控制信号；readIn 计数器的写入值信号；AO、BO 多路选择器的另外两个输入，DL、XL 为 RAM 的控制信号，均为 0 输出高阻态，DL 为 0、XL 为 1 写数据，DL 为 1、XL 为 0 读数据；还有一个双向端口 ToBus 进行 RAM 的输出写入操作。



完成 RAM 部分的所有功能。可以看到 RAM 的读、写、输出高阻态均正常；多路选择器的选择信号无误，PC 计数器完成置数和加一的操作。35 号为 PC 中的寄存器的监听，44 号为 MUX 传向 RAM 的值，53 为 ToBus 的输入，62 为 ToBus 的最终结果，可以看到都是正常工作的。

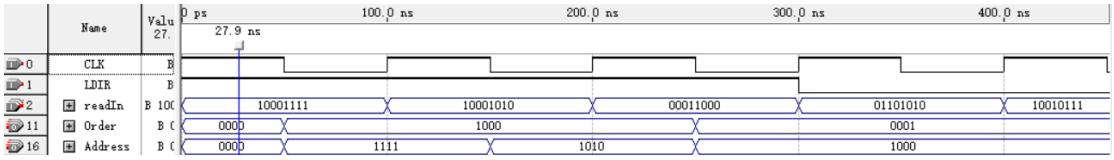
3. 指令译码器与指令寄存器

1） 指令寄存器

首先进行指令寄存器的设计，其接口、实现、功能仿真如下：

```
-- orderReg
library ieee;
use ieee.std_logic_1164.all;

entity orderReg is
    port(
        CLK, LDIR : in std_logic;
        readIn : in std_logic_vector(7 downto 0);
        Order, Address : out std_logic_vector(3 downto 0)
    );
end orderReg;

architecture behave of orderReg is
begin
    process(CLK, LDIR, readIn)
    begin
        if (CLK'event and CLK = '0') and LDIR = '1' then
            Order <= readIn(7 downto 4);
            Address <= readIn(3 downto 0);
        end if;
    end process;
end;
```

指令寄存器有三个输入，为 CLK 时钟，LDIR 写入控制信号，readIn 为来自总线的输入信号，order 和 Address 分别输出指令和地址。功能并不复杂，这里不再进行赘述。



可以看到，均是在时钟的下降沿完成操作，当 LDIR 为 0 时不进行写入，当 LDIR 为 1 的时候执行写入操作，并将其值进行输出。成功完成指令寄存器的设计工作。

2） 指令译码器

接下来进行指令译码器的设计。先进行重编码的设计，其接口设计、功能实现和仿真如下：

```
entity redecoder is
    port(
        SM : in std_logic;
        Order, Address : in std_logic_vector(3 downto 0);
        RESET, MOVA, MOVB, MOVC, ADD, SUB, OR1, NOT1, RSR, RSL, JMP, JZ, JC, IN1, OUT1, NOP, HALT : out std_logic;
        ToALU : out std_logic_vector(3 downto 0);
        RAA, RWBA : out std_logic_vector(1 downto 0)
    );
end redecoder;
```

SM 为复位控制信号，Order、Address 来自指令寄存器的输入数据，RESET 等为重编码之后的控制信号，ToALU 为传输给 ALU 的信号，RAA、RWBA 为传输给寄存器组的信号。

```
process(SM, Order, Address)
variable temp1, temp2 : std_logic_vector(1 downto 0);
begin
    temp1 := Address(1 downto 0);
    temp2 := Address(3 downto 2);
    RESET <= '0';
    MOVA <= '0';
    MOVB <= '0';
    MOVC <= '0';
    ADD <= '0';
    SUB <= '0';
    OR1 <= '0';
    NOT1 <= '0';
    RSR <= '0';
    RSL <= '0';
    JMP <= '0';
    JZ <= '0';
    JC <= '0';
    IN1 <= '0';
    OUT1 <= '0';
    NOP <= '0';
    HALT <= '0';
    if SM = '0' then
    elsif SM = '1' then
    ToALU <= Order;
    RAA <= temp1;
    RWBA <= temp2;
end process;
```

```
if Order = "1111" then -- MOV
    if temp2 = "11" then
        temp2 := "10";
        MOVB <= '1';
    elsif temp1 = "11" then
        temp1 := "10";
        MOVC <= '1';
    else
        MOVA <= '1';
    end if;
elsif Order = "1001" then -- ADD
    ADD <= '1';
elsif Order = "0110" then -- SUB
    SUB <= '1';
elsif Order = "1011" then -- OR
    OR1 <= '1';
elsif Order = "0101" then -- NOT
    NOT1 <= '1';
elsif Order = "1010" then -- RSR RSL
    if temp1 = "00" then
        RSR <= '1';
    elsif temp1 = "11" then
        RSL <= '1';
    end if;
elsif Order = "0001" then -- JMP JC JZ
    if temp1 = "00" then
        JMP <= '1';
    elsif temp1 = "01" then
        JZ <= '1';
    elsif temp1 = "10" then
        JC <= '1';
    end if;
elsif Order = "0010" then -- IN
    IN1 <= '1';
elsif Order = "0100" then -- OUT
    OUT1 <= '1';
elsif Order = "0111" then -- NOP
    NOP <= '1';
elsif Order = "1000" then -- HALT
    HALT <= '1';
end if;
```

重编码时，要进行初始化的置一操作，SM 为 0 时，输出 RESET 为 0，其他的根据指令对应表进行设计。这里不再赘述。特别注意的时 signal 不能在 process 中进行改变，我们应该使用 vairable 进行操作。

接下来进行控制信号译码的操作，其接口设计、功能实现和仿真如下：

```
entity Console is
    port(
        C, Z : in std_logic;
        RESET, MOVA, MOVB, MOVC, ADD, SUB, OR1, NOT1, RSR, RSL, JMP, JZ, JC, IN1, OUT1, NOP : in std_logic;
        WE, FBus, FLBus, FRBus, INPC, LDPC, DL, XL, LDIR, init, outit : out std_logic;
        MADD : out std_logic_vector(1 downto 0)
    );
end Console;
```

C、Z 为上一步操作时，保留下来的结果；RESET 等为重编码器的输出；WE 等为 CPU 的控制信号，MADD 为多路选择器的控制信号。

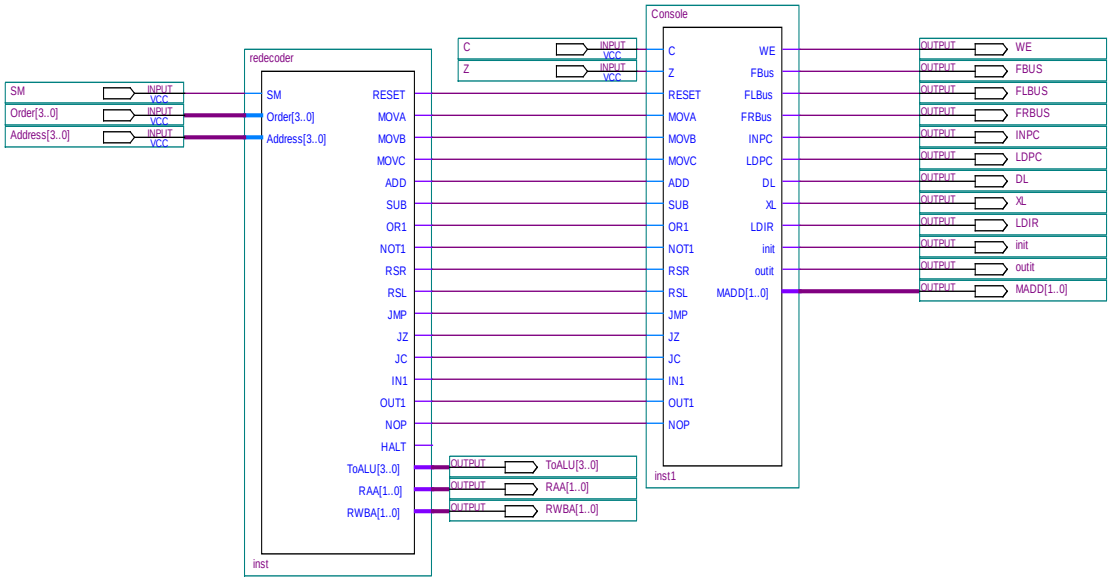
```
if RESET = '1' then
    WE <= '1';
    FBus <= '0';
    FLBus <= '0';
    FRBus <= '0';
    INPC <= '1';
    LDPC <= '0';
    MADD <= "00";
    DL <= '1';
    XL <= '0';
    LDIR <= '1';
    init <= '0';
    outit <= '0';
```

进行初始化的操作，以便可以从 RAM 中读取指令；

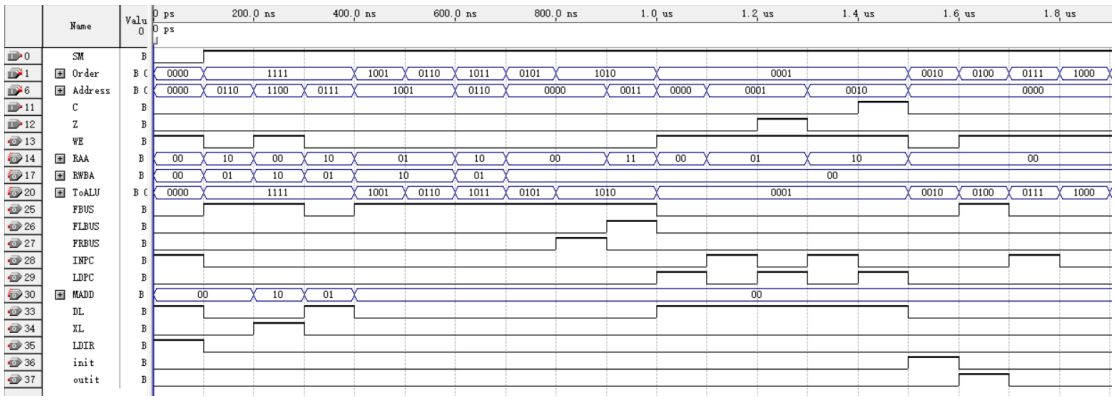
```
WE <= not(MOVA or MOVC or ADD or SUB or OR1 or NOT1 or RSR or RSL or IN1);
FBus <= MOVA or MOVB or ADD or SUB or OR1 or NOT1 or RSR or RSL or OUT1;
FLBus <= RSL;
FRBus <= RSR;
INPC <= (JZ and not(Z)) or (JC and not(C)) or NOP;
LDPC <= JMP or (JZ and Z) or (JC and C);
if MOVB = '1' then
    MADD <= "10";
elsif MOVC ='1' then
    MADD <= "01";
else
    MADD <= "00";
end if;
DL <= MOVC or JMP or JZ or JC;
XL <= MOVB;
init <= IN1;
outit <= OUT1;
LDIR <='0';
```

对指令进行编码操作，输出给 CPU 的对应控制器；

接下来将两个部件进行组合，完成整体的译码器的设计：



对应的仿真如下：



每条命令均得到对应的控制信号，比如 MOVA 指令，WE 为 0；FBus 为 1；FLBus、FRBus 为 0 等。顺利完成指令译码器的设计工作。

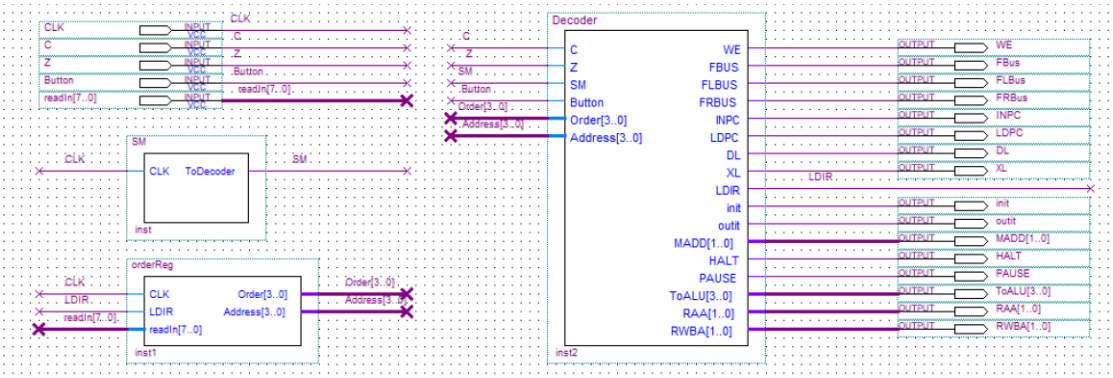
3） 联调

将 SM 计数信号、指令寄存器、指令译码器进行整合成 DecoderPart，其设计、实现和仿真如下：

```
library ieee;
use ieee.std_logic_1164.all;

entity SM is
    port(
        CLK : in std_logic;
        ToDecoder : out std_logic
    );
end SM;

architecture behave of SM is
signal temp: std_logic := '0';
begin
    process(CLK)
    begin
        if CLK'event and CLK ='0' then
            temp <= not(temp);
        end if;
    end process;
    ToDecoder <= temp;
end;
```

这是 SM 的实现，对时钟进行计数每次经过下降沿就进行一次取反。

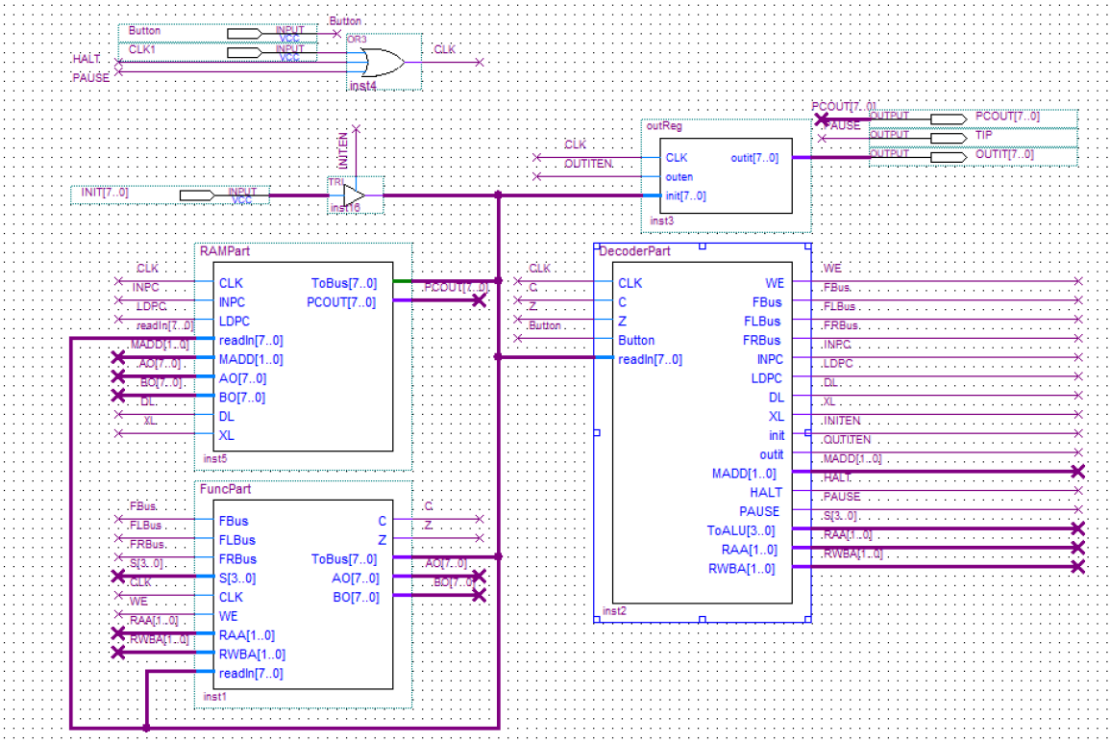
接下来是整个 Part 的实现：



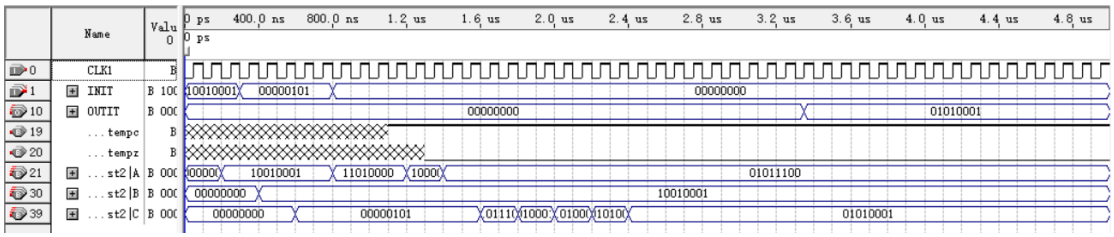
通过上面的逻辑图完成 DecoderPart 的设计，CLK 为时钟、readIn 为总线输入，C、Z 为上一次操作存留下来的值；其余的均为输出，是 CPU 的各种控制信号。

4. CPU 顶层文件设计

顶层文件的设计如下图所示：



需要注意的是向总线上传值时需要设计三态门进行缓冲，不可直接连接到总线，否则会造成问题。接下来给出 CPU 的功能仿真验证，第四部分为在学习板的移植部分。

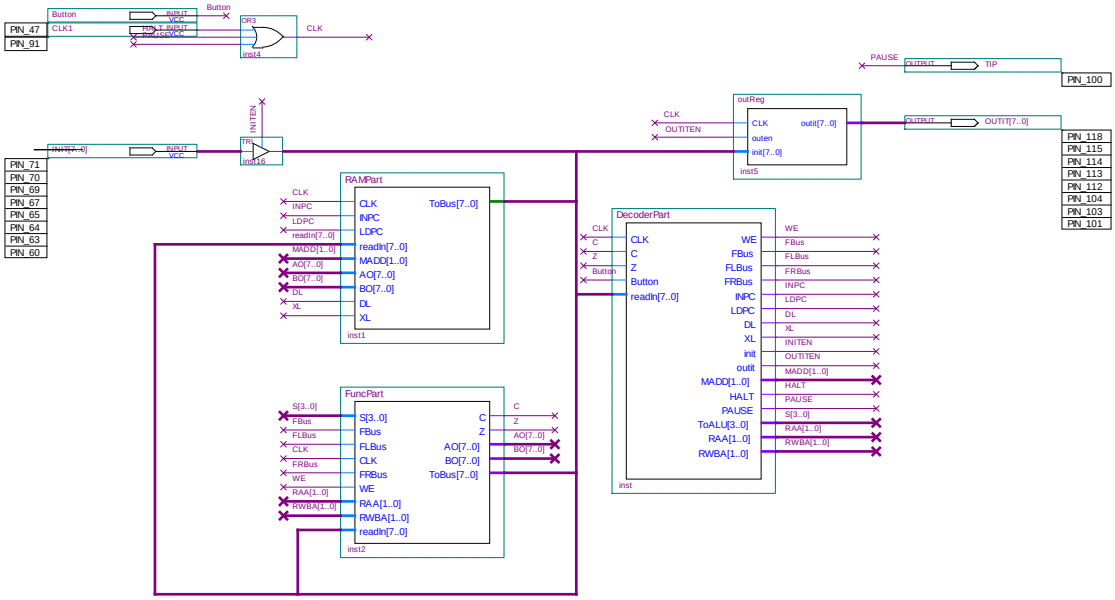


经过一系列的命令最后输出正确结果“01010001”。具体的指令可以参见第四部分。

四、 系统测试

(一) 测试环境

测试基于 DDA-IIIA 型学习板，学习板使用的芯片为 Cyclone II 系列的 EP2C5T144C8N 芯片。



可以看出顶层图与第三部分的初始版本有所区别，具体为：

输出前的三态门换成了寄存器，这样可以保存输出的值使得其可以持续输出值，并且通过他替代三态门的作用。需要说明的是这是一个时序部件，需要在输出完成确定的时候进行写入操作，避免延时的产生。

将控制时钟的三态门换成了三输入的或门，同时接入 HALT 与 PAUSE 命令，实现对电路的停机和暂停。

同时加入 Button 的输入这个按钮告诉程序，输入已经结束可以继续执行。同时加入提示需要输入的 LED 灯。需要说明的是在学习板上按钮按下为“0”。

输入的拨码开关的引脚以此为：

71 70 69 67 65 64 63 60

输出的 LED 灯的引脚分配为：

118 115 114 113 112 104 103 101

按钮的引脚分配为：

47

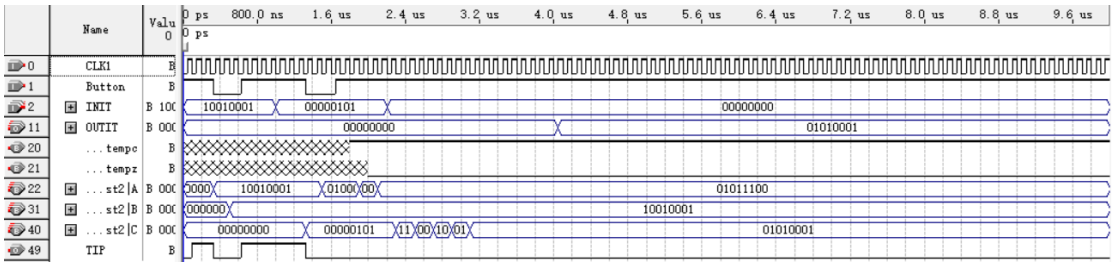
提示输入的引脚为：

100

时钟的引脚为：

91

下面给出进行了功能添加后的仿真图：



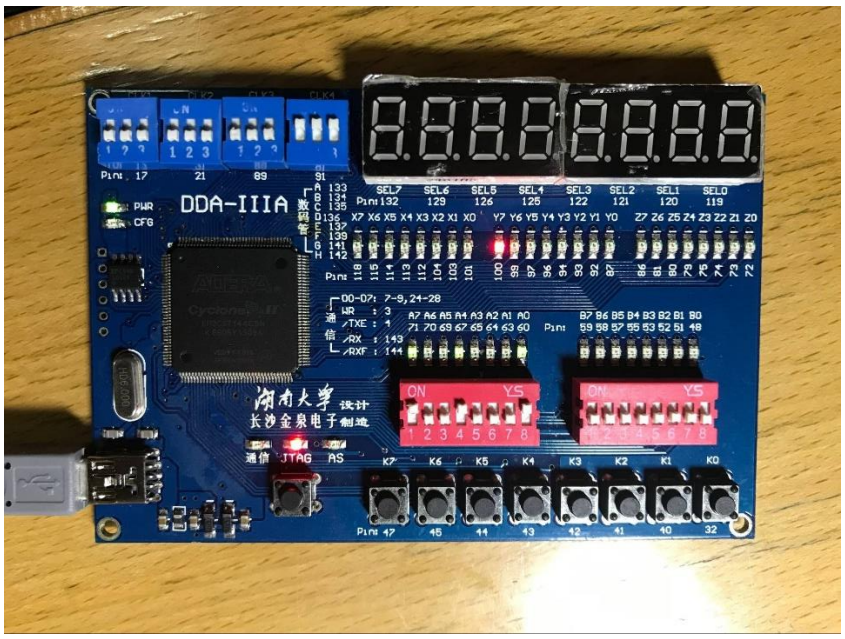
首先可以看出我们的程序依然得到了正确的结果“01010001”，其中的输入需要 Button 确认后才会进行写入的操作。

(二) 测试代码

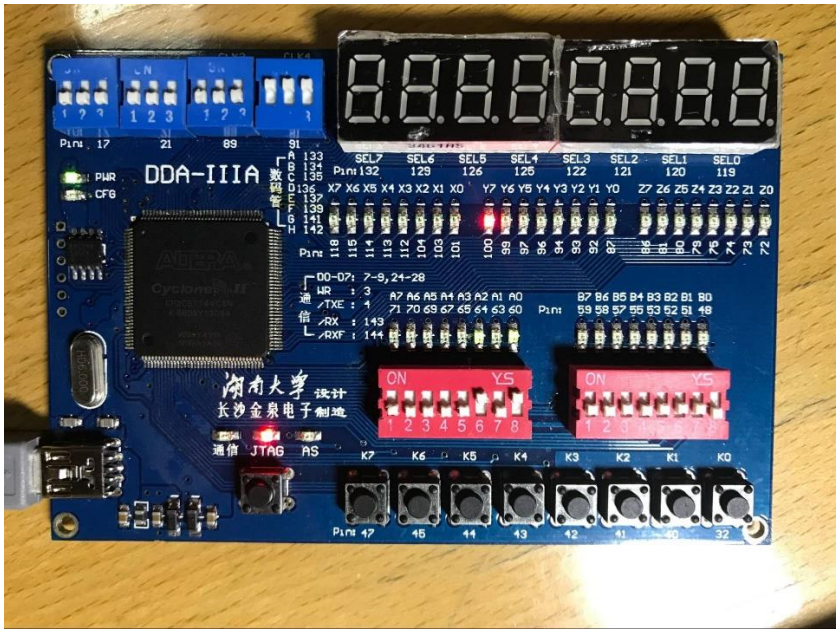
位置		指令	A	B	C	IN	C	RAM(5)	输出
0	INA	00100000	10010001			10010001		11010000	
1	MOVBA	11110100	10010001	10010001				11010000	
2	INC	00101000	10010001	10010001	00000101	00000101		11010000	
3	MOVA11	11110011	11010000	10010001	00000101			11010000	
4	MOV11B	11111101	11010000	10010001	00000101			10010001	
5	ADDAB	10010001	01100001	10010001	00000101		1		
6	SUBAC	01100010	01011100	10010001	00000101				
7	ORCA	10111000	01011100	10010001	01011101				
8	NOTC	01011000	01011100	10010001	10100010				
9	RSRC	10101000			01010001				
10	RSRC	10101000			10101000				
11	RSLC	10101011			01010001				
12	JMP	00010000							
13		00001111							
14		00000000							
15	JZ	00010001							
16		10011111							
17	JC	00010010							
18		00010100							
19		00000000							
20	NOP	01110000							
21		00000000							
22	OUTC	01001000							01010001
23	HALT	10000000							

总共使用了 RAM 中的 23 为来进行验证第一列为 RAM 中的位置；第二、三列为依次执行的指令及其二进制编码；第四、五、六列为通用寄存器组中 A、B、C 三个寄存器的值；第七列为输入的数据；第八列为 RAM(5)的内容的变化，主要是对 MOV 指令的说明；最后一列为期望输出。

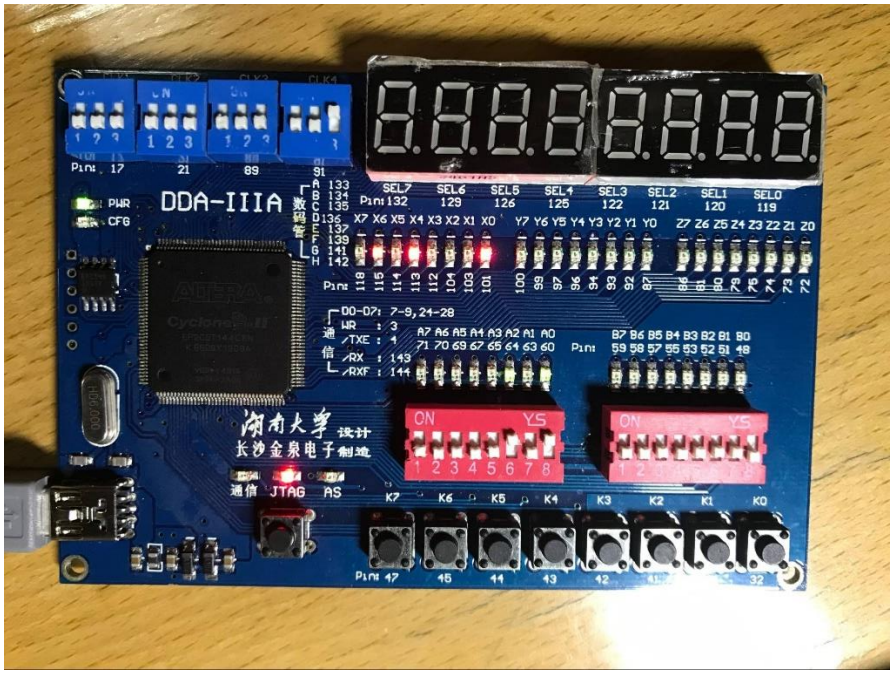
（三） 测试结果



第一次输入通过拨码开关输入“1001001”；



第二次输入通过拨码开关输入“00000101”；



输出最后结果“01010001”。

五、 总结

在本次实验中，遇到了不少问题。如：指令译码器的设计，各种控制信号的交互，关机指令如何实现等等。

但是，通过自己的一系列尝试和努力，还是成功的完成了本次实验。我认为，

数电可以极好地锻炼我的思维方式和严密逻辑，坚定了我这学期学好这门课的决心，我相信我一定能取得优良的成绩。