VHDL期末大作业

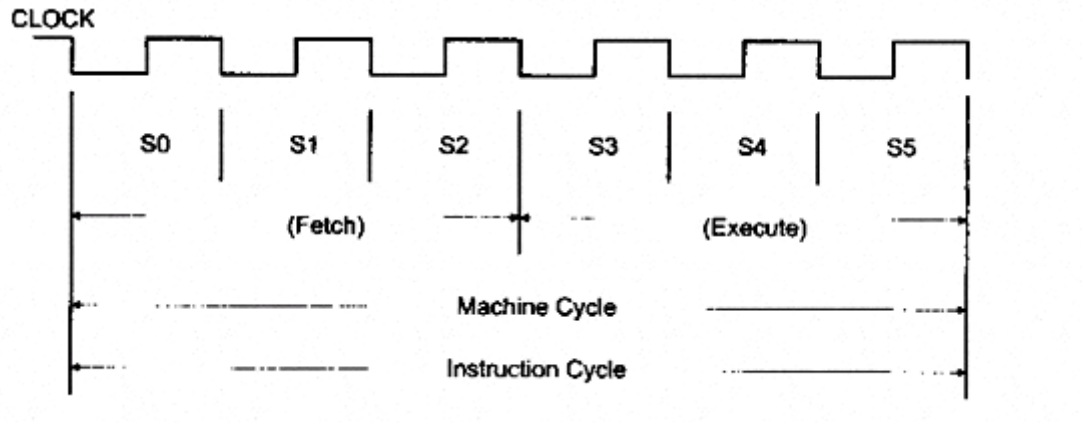
——VHDL实现简单的8位CPU分析与仿真

学号：22920192204097

姓名：吴雨娟

一、CPU工作分析

CPU工作时序图如下：



从图中可以看到，CPU工作时一共有6个状态：S0、S1、S2、S3、S4、S5。

在S0状态，程序计数器会把值赋给MAR。

在S1状态，迭代器PC的值迭代加一，由于一个状态会占用两个时钟周期，所以要引入标志寄存器FLAG，以保证S1状态中，PC只加一次。

在S2状态，从ROM中读取指令，并将指令赋给指令寄存器IR。

在S3状态，取出指令寄存器的高4位，高4位为操作指令，不同的值代表不同操作。如果高4位值为“0000”，则执行MOV操作，如果为“0001”则执行ADD操作，如果为“0010”则执行SUB操作。当高4 位为上述三个值时，将指令的低4位取出，赋给MAR，指令低4位是数据存储的地址。如果高4位的值为“1110”，则执行OUT操作，将寄存器ax中的值赋给output_data进行输出。如果高4位的值为“1111”，则执行HLT操作，run赋值为‘0’，程序停止运行。

在S4状态，读出数据地址中存储的数据，若为MOV操作，直接将数据的值赋给ax，若为ADD或SUB操作，则将数据的值赋给bx，等待进行计算操作。

在S5状态，进行计算操作。若为ADD操作，则将ax与bx相加的值赋给ax。若为SUB操作，则将ax减去bx的值赋给ax。

可以看到，经过一个完整的S0-S5状态，会得到一个结果存到ax中去。计算本例中的10h+20h+60h-10h=80h，最后的output_data的值为80h。在第四部分的仿真图中可以看到结果。

二、ROM16_8代码分析

代码块下方文字是对代码块的分析。

(1)

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

library ieee;表示调用IEEE库。

use ieee.std_logic_1164.all;表示引入ieee.std_logic_1164包中的所有项目。

use ieee.std_logic_unsigned.all;表示引入ieee.std_logic_unsigned包中的所有项目。

use ieee.std_logic_arith.all;表示引入ieee.std_logic_arith包中的所有项目。

(2)

```
entity rom16_8 is
  port( addr:in std_logic_vector(3 downto 0);
          ce:in std_logic;
     dataout:out std_logic_vector(7 downto 0)
       );
end entity rom16_8;
```

该段代码定义了一个实体rom16_8，有两个输入端口，其中，addr的数据类型是具有4个标准逻辑位的逻辑矢量，ce的数据类型是标准逻辑位。有一个输出端口dataout，其数据类型是具有8个标准逻辑位的逻辑矢量。

(3)

```
architecture behave of rom16_8 is
begin
            --10+20+60-10=80h
  dataout<="00001001" when addr="0000" and ce='0' else   --mov ax,9h
           "00011010" when addr="0001" and ce='0' else   --add ax,addr
           "00011011" when addr="0010" and ce='0' else   --add ax,bh
           "00101100" when addr="0011" and ce='0' else   --sub ax,addr

           "11100000" when addr="0100" and ce='0' else   --out
           "11110000" when addr="0101" and ce='0' else   --hlt
           "00000000" when addr="0110" and ce='0' else
           "00000000" when addr="0111" and ce='0' else

           "00000000" when addr="1000" and ce='0' else
           "00010000" when addr="1001" and ce='0' else   --10h
           "00100000" when addr="1010" and ce='0' else   --20h
           "01100000" when addr="1011" and ce='0' else   --60h

           "00010000" when addr="1100" and ce='0' else   --10h
           "00000000" when addr="1101" and ce='0' else
           "00000000" when addr="1110" and ce='0' else
           "00000000" when addr="1111" and ce='0' else
           "11111111";
end architecture behave;
```

该段代码定义了一个构造体，构造体名为behave，即采用行为描述的方法描述逻辑功能，实体名为rom16_8。本例设计的这个rom16_8可进行10+20+60-10=80h的操作，begin到end中间的语句为并行处理语句。dataout的赋值部分采用的是条件信号赋值语句。因为addr的数据类型是具有4个标准逻辑位的逻辑矢量，所以addr共有16种二进制情况，而且ce=0时才允许读，所以总的二进制情况数也是16种。列举出16种二进制情况，对dataout进行赋值。

接下来解释rom16_8如何模拟进行10+20+60-10=80h的操作。

当 addr="0000" 且 ce='0'时，高四位为指令，对应MOV的指令‘0000’，低四位为值所在地址，初始将10h赋值给ax，而地址addr=‘1001’处存储10h的值"00010000”，所以将‘00001001’赋给dataout。

当 addr="0001" 且 ce='0'时，高四位为指令，对应ADD的指令‘0001’，低四位为值所在地址，此次要相加的数是20h，而地址addr=‘1010’处存储20h的值"00100000"，所以将‘00011010’赋给dataout。

当 addr="0010" 且 ce='0'时，高四位为指令，对应ADD的指令‘0001’，低四位为值所在地址，此次要相加的数是60h，而地址addr=‘1011’处存储60h的值"01100000"，所以将‘00011011’赋给dataout。

当 addr="0011" 且 ce='0'时，高四位为指令，对应SUB的指令‘0010’，低四位为值所在地址，此次要减去的数是10h，而地址addr=‘1100’处存储10h的值"00010000”，所以将‘00101100’赋给dataout。

当 addr="0100" 且 ce='0'时，高四位为指令，对应OUT的指令‘1110’，将累加器中的结果值输出。而低四位无意义，将‘0000’放在低四位，所以将‘11100000’赋给dataout。

当 addr="0101" 且 ce='0'时，高四位为指令，对应HLT的指令‘1111’，程序停止运行。而低四位无意义，将‘0000’放在低四位，所以将‘11110000’赋给dataout。

当 addr="0110" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

当 addr="0111" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

当 addr="1000" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

当 addr="1001" 且 ce='0'时，将十六进制数10转化成二进制数赋值给dataout，所以将‘00010000’赋给dataout。

当 addr="1010" 且 ce='0'时，将十六进制数20转化成二进制数赋值给dataout，所以将‘00100000’赋给dataout。

当 addr="1011" 且 ce='0'时，将十六进制数60转化成二进制数赋值给dataout，所以将‘01100000’赋给dataout。

当 addr="1100" 且 ce='0'时，将十六进制数10转化成二进制数赋值给dataout，所以将‘00010000’赋给dataout。

当 addr="1101" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

当 addr="1110" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

当 addr="1111" 且 ce='0'时，无操作，所以将‘00000000’赋给dataout。

因为addr的数据类型是std_logic_vector，所以会有高阻态和不确定状态等，当addr不属于上述16种情况时，将‘11111111’赋给dataout。

三、CPU_CONTROL代码分析

代码块下方文字是对代码块的分析。

（1）

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

library ieee;表示调用IEEE库。

use ieee.std_logic_1164.all;表示引入ieee.std_logic_1164包中的所有项目。

use ieee.std_logic_unsigned.all;表示引入ieee.std_logic_unsigned包中的所有项目。

use ieee.std_logic_arith.all;表示引入ieee.std_logic_arith包中的所有项目。

(2)

```
entity cpu_control is
  port(clk:in std_logic;
       rst:in std_logic;
       output_data,databus_out:out std_logic_vector(7 downto 0)
       );
end entity cpu_control;
```

该段代码定义了一个实体cpu_control，有两个输入端口，其中，clk和rst的数据类型都是标准逻辑位。有两个输出端口output_data和databus_out，数据类型都是具有8个标准逻辑位的逻辑矢量。

(3)

```
architecture behave of cpu_control is
```

定义了一个构造体，构造体名为behave，即采用行为描述的方法描述逻辑功能，实体名为cpu_control。

(4)

```
component rom16_8 is
    port( addr:in std_logic_vector(3 downto 0);
          ce:in std_logic;
     dataout:out std_logic_vector(7 downto 0)
       );
end component rom16_8;
```

component语句说明构造体由rom16_8元件组成，rom16_8有两个输入端口，其中，addr的数据类型是具有4个标准逻辑位的逻辑矢量，ce的数据类型是标准逻辑位。有一个输出端口dataout，其数据类型是具有8个标准逻辑位的逻辑矢量。

(5)

```
type states is (s0,s1,s2,s3,s4,s5);
signal current_state,next_state:states;
signal flag:std_logic;
signal pc,addrbus:std_logic_vector(3 downto 0);
signal ax,bx:std_logic_vector(7 downto 0);--Ax,Bx
signal cs:std_logic;
signal databus:std_logic_vector(7 downto 0);
signal run:std_logic;
signal mar:std_logic_vector(3 downto 0);
signal ir:std_logic_vector(7 downto 0);

signal f1:std_logic;
```

定义了5个状态：s0,s1,s2,s3,s4,s5。

定义了states类型信号current_state和next_state。

定义了标准逻辑位类型信号flag。

定义了具有4个标准逻辑位的逻辑矢量类型信号pc和addrbus。

定义了具有8个标准逻辑位的逻辑矢量类型信号ax和bx。

定义了标准逻辑位类型信号cs。

定义了具有8个标准逻辑位的逻辑矢量类型信号databus。

定义了标准逻辑位类型信号run。

定义了具有4个标准逻辑位的逻辑矢量类型信号mar。

定义了具有8个标准逻辑位的逻辑矢量类型信号ir。

定义了标准逻辑位类型信号f1。

（6）

```
begin
  reg:process
    variable temp:std_logic_vector(3 downto 0);
```

begin表示开始下面的并行处理语句。

开始名为reg的进程。

定义了具有4个标准逻辑位的逻辑矢量类型变量temp。

（7）

```
begin
    if rst='1' then                    --系统复位
       pc<="0000";
       ax<="00000000";
       bx<="00000000";
       run<='1';
       flag<='1';                      --PC可以加1
       f1<='1';
       current_state<=s0;
       --databus<="00000000";
```

开始进程，处理顺序处理语句。

当复位键rst='1'时，进行复位操作。程序计数器pc清零，寄存器ax和bx也清零，当前处于运行状态，所以run赋值为'1'，当前pc可以加一，所以标志寄存器flag赋值为'1'，处于可计算状态，所以f1赋值为'1'，当前处于初始状态，所以current_state赋值为's0'。

(8)

```
    elsif rising_edge(clk) then
       if run='1' then
         case current_state is
           when s0=>next_state<=s1;  --Address State&Fetch
                    mar<=pc;         --将PC的值赋给MAR，在S2状态时从ROM中读出指令
（IR<=databus）
```

当复位键rst不等于1'时，不进行复位操作。如果当前处于时钟信号上升沿，且run='1'，即当前处于运行状态，则进行状态的变化。运用case语句，判断当前的状态，再决定下一步操作。

若当前状态处于s0，则下一个状态是s1。

并且将程序计数器pc的值赋给mar。

(9)

```
           when s1=>next_state<=s2;
                    if flag='1' then
                       pc<=pc+1;      --地址加1
                       flag<='0';
                    end if;
```

若当前状态处于s1，则下一个状态是s2。

若当前flag='1'，则pc加一，并且将flag赋值为'0'。因为引入了复位键rst，为了避免一个信号有多个源，将时序进程和组合进程合并，所以导致一个状态占用了两个时钟周期。判断标志寄存器flag可以确保在s1状态pc只加一次。

(10)

```
           when s2=>next_state<=s3;
                    flag<='1' ;
                    ir<=databus;      --将ROM中的指令赋给IR（指令寄存器）
```

若当前状态处于s2，则下一个状态是s3。

将flag赋值为'1'，从只读存储器rom中读取指令，并将值赋给指令寄存器ir。

(11)

```
            when s3=>next_state<=s4;
                    temp:=ir(7 downto 4);          --取指令寄存器的高4位（指令代码 MOV
0000，ADD 0001，SUB 0010）
                    -- Mov Add Sub
                    if (temp="0000") or (temp="0001")or (temp="0010") then
                        mar<=ir(3 downto 0);       --将数据存储的地址赋给MAR
                    elsif temp="1110" then
                        output_data<=ax;          -- out over
                    elsif temp="1111" then
                        run<='0';                   --Hlt   over
                    end if;
```

若当前状态处于s3，则下一个状态是s4。

取出指令寄存器的高四位，将其值赋给temp，如果指令是‘0000’，即MOV，或者‘0001’，即ADD，或者‘0010’，即SUB，则取出低四位作为数据存储的地址，将数据存储的地址赋给mar。当指令是‘1110’，即OUT时，将ax中的值取出，赋给output_data进行输出。当指令是‘1111’，即HLT时，run赋值为‘0’，程序停止运行。

（12）

```
            when s4=>next_state<=s5;          --在S4状态，将从数据总线上读取数据
                    if temp="0000" then      --MOV AX,data over
                       ax<=databus;
                    elsif temp="0001" then   --ADD ax,data
                       bx<=databus;
                       f1<='1';
                    elsif temp="0010" then   --SUB ax,bx
                       bx<=databus;
                       f1<='1';
                    end if;
```

若当前状态处于s4，则下一个状态是s5。

从数据总线上读取数据，当指令为‘0000’时，直接将databus的值赋给ax。当指令为‘0001’时，将databus的值暂时赋给bx，f1赋值为‘1’，准备进行加法计算。当指令为‘0010’时，将databus的值暂时赋给bx，f1赋值为‘1’，准备进行减法计算。

（13）

```
            when s5=>next_state<=s0;
                    if temp="0001" and f1='1'  then
                       ax<=ax+bx;
                       f1<='0';
                    elsif temp="0010" and f1='1'  then
                       ax<=ax-bx;
                       f1<='0';
                    end if;
            end case;
            current_state<=next_state;      --此设计使每个状态为两个时钟周期
(Attention!!!!)
          end if;
       end if;
    end process reg;
```

若当前状态处于s5，则下一个状态是s0。

如果指令为‘0001’，且f1='1'，即可进行加法计算，将ax+bx的值赋给ax，且f1赋值为‘0’，即取消计算准备。如果指令为‘0010’，且f1='1'，即可进行减法计算，将ax-bx的值赋给ax，且f1赋值为‘0’，即取消计算准备。结束case语句后，将next_state的值赋给current_state，因为状态赋值会有延迟，所以每个状态第一个时钟周期结束后，next_state还没有产生变化，只有经历两个时钟周期后，才完成对next_state的赋值。所以next_state的值在经历两个时钟周期后才发生改变，变成下一个状态的值，所以“current_state<=next_state;”语句可以使每个状态为两个时钟周期。然后结束进程。

（14）

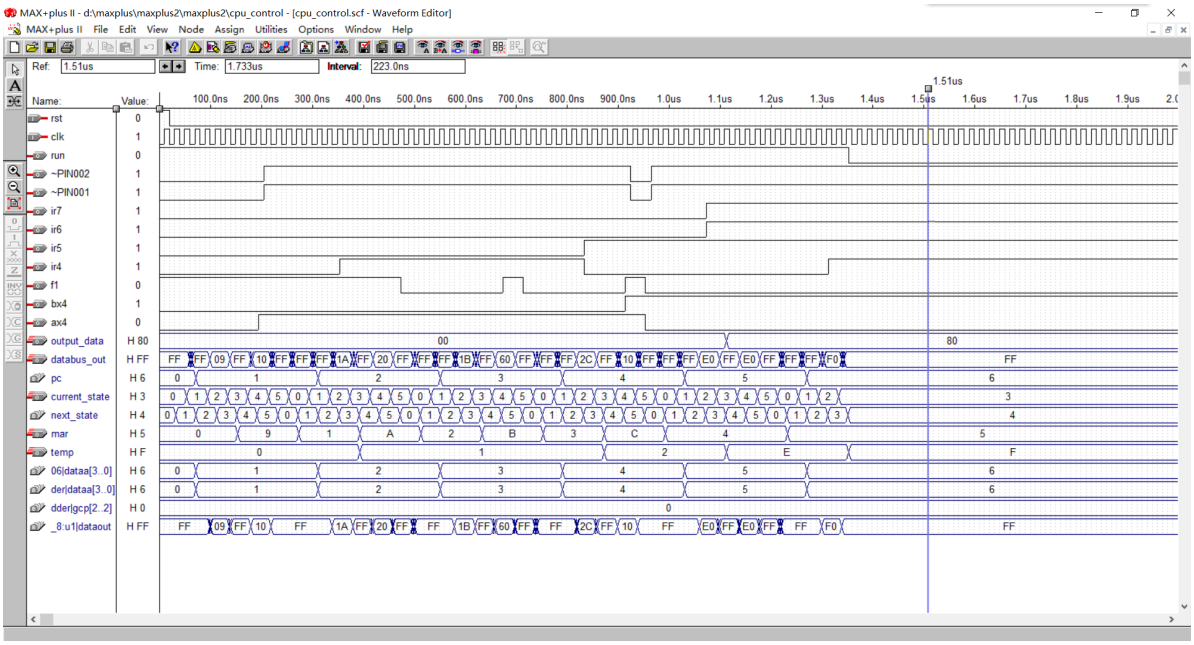
```
u1:rom16_8 port map(addrbus,cs,databus);
addrbus<=mar when (current_state=s2) or (current_state=s4) else
         "0000";
cs<='0' when (current_state=s2) or (current_state=s4) else
    '1';
databus_out<=databus;
end architecture behave;
```

port map语句说明了元件之间的连接关系，元件标号u1，进行位置映射，addrbus对应rom16_8中的addr端口，cs对应ce端口，databus对应dataout端口。

接下来进行连线。如果当前状态处于s2，即取指令状态，或者s4，即取值状态，将mar中地址的值赋给addrbus，否则将无效地址‘0000’赋给addrbus。如果当前状态处于s2，即取指令状态，或者s4，即取值状态，将‘0’赋给cs，就可以进行读取操作，否则将‘1’赋给cs，无法进行读取操作。将databus的值赋给databus_out，即把databus和databus_out连起来。然后结束构造体。

四、仿真结果

将文件中所给源代码输入MAXplus II软件，得到仿真结果图如下：



从仿真结果图可以看出，output_data最后输出的值为80h，结果正确。