**f4a 8位12指令硬布线CPU设计**

**实验目标**

设计一个硬布线控制器的8位模型计算机按照如下图要求，设计一台硬布线控制器的8位模型计算机，用VHDL 语言完成设计并调试成功，完成仿真波形图。

**模型机的设计要求**

|  |  |
| --- | --- |
| 总线 | 单总线结构 |
| 数据线、地址线 | 8位 |
| 指令系统 | 寻址方式  2种（立即数寻址、直接寻址）  类型    5种（算术运算、逻辑运算、数据传送、程序控制）  指令 12条(add、sub、mul、div、neg、and 、not、 or、load、store、branch、halt) |
| 运算器 | 单累加器结构, 8种运算 |
| 控制器 | 硬布线，微控制信号根据需要确定 |

**1.1文字描述**

此模型计算机以控制器为中心，指令数据存储在RAM中，指令数据通过数据总线传输到指令寄存器，控制器从指令寄存器中取指令，编译指令，输出控制信号，控制ALU运算，PC加一为取下一条指令做准备，并从RAM中取出数据进行运算，运算结果寄存到累加器中，后通过数据总线存到RAM中，指令寄存器读下一条指令，依次循环。

**模型机设计框图**



## 1.2数学模型 指令执行过程

**指令执行过程：**

CPU是一个复杂的有限状态机。指令执行包括三个阶段：

1、取指令阶段：从存储器取出一条指令；

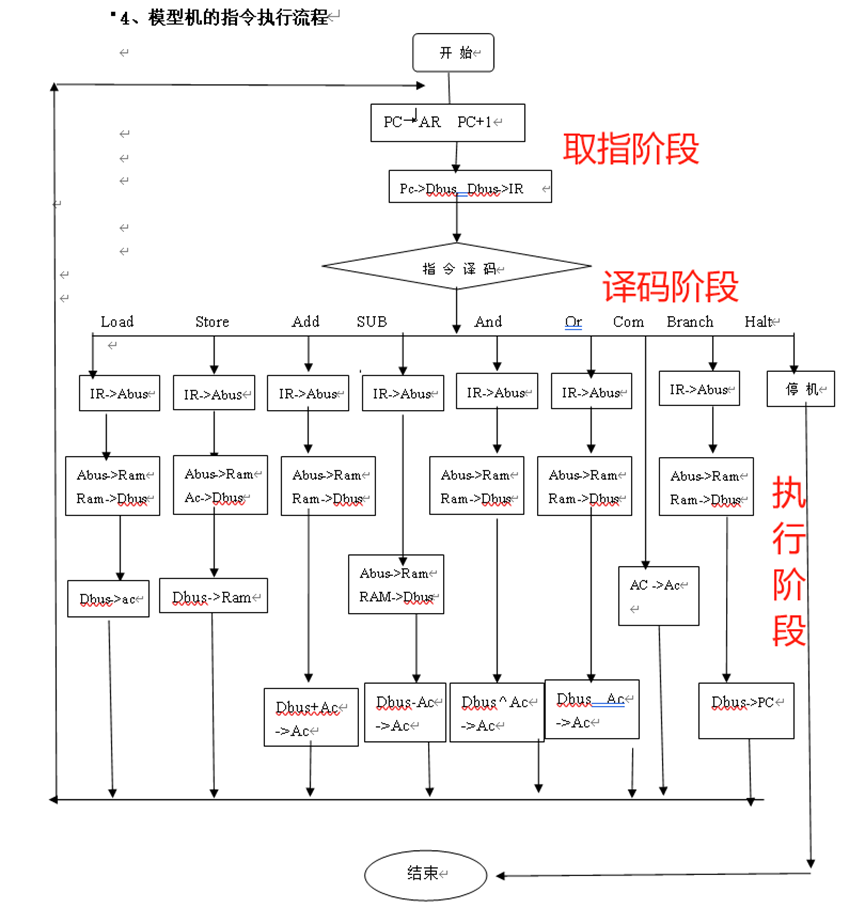
2、指令译码阶段：对取出的指令进行译码，即确定取到的指令是何种指令，然后转移到该种指令的执行阶段；

3、指令执阶段：执行指令。

指令执行完毕，又转移到下一条指令的取指令阶段，开始新一轮的循环。

**指令格式**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | load | Store | Add | sub | mul | div | neg | and | or | not | halt | branch |
| 操作码编码 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 |



## 模型机的微操作控制信号及其实现方法

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 肋记符 | 时序 | 状态 | 12位控制字 | | | | | | | | | | | | |
|  |  |  | Pc\_enA | MEM\_enD | Pc\_inc | Pc\_ld | Mem\_rw | Ir\_enA | Ir\_enD | Ir\_ld | Acc\_enD | Acc\_selalu | Acc\_ld | Alu\_op | |
| Load | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Store | T1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Add | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Sub | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| And | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Or | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Not | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Neg | T2 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |  |
|  | T1 |  | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Branch | T1 |  | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |
| Halt | T1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |
|  | T0 |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |  |

## 1.3指令格式和功能

**模型的寻址方式为直接寻址**  
直接寻址方其特点是：在指令格式的地址字段中直接指出操作数在内存中的地址D。指令字中的形式地址D就是操作数的有效地址EA



|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | load | Store | Add | sub | mul | div | neg | and | or | not | halt | branch |
| 操作码编码 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 |

**指令执行操作和功能**

(1) Load D    (将D存储单元的数据存入ACC)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=0,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | MEN->DBUS | Mem\_rw=1,mem\_enD=1 |
| DBUS->ACC | Acc\_ld=1 |

(2) STORE D   (将ACC中的数据存入内存单元D中)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | ACC->DBUS | Acc\_enD=1 |
| DBUS->(MEM) | (MEM)\_RW=0 |

(3)ADD  D  (将两数相加运算)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1,mem\_end=1 |
| A+(MEM)->A | Acc\_enD=1,(MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1  Mem\_end=1,alu\_op=1 |

(4)SUB D  (将ACC数据减去D单元数据存入acc)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1 |
| A-(MEM)->A | Acc\_enD=1,(MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1  Mem\_end=1,alu\_op=1 |

  （5）AND D   (将两数做与运算)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1 |
| A and (MEM)->A | Alu\_op=1,mem\_end=1  Acc\_enD=1,(MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1 |

  (6)OR D  (将两数做或运算)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1 |
| A or (MEM)->A | Alu\_op=1,mem\_end=1  Acc\_enD=1,(MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1 |

  (7)NEG D (将数求补)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1 |
| 0- M(D)->A | Alu\_op=1,mem\_end=1  (MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1 |

   (8)NOT D  （对数求反）

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | PC->ABUS | PC\_enA=1 |
| **T2** | (MEM)->DBUS | MEM\_RW=1 |
| Not M(D)->A | Alu\_op=1,mem\_end=1  (MEM)\_RW=1  Acc\_ld=1,acc\_selalu=1 |

(9) BRANCH  D  (无条件转移)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | Pc\_inc=1 |
| **T1** | (MEM)->PC | (MEM)\_RW=1,mem\_enD=1   PC\_ld=1 |

(10)HALT      (停机指令)

|  |  |  |
| --- | --- | --- |
| 周期 | 所执行的操作 | 对应的微程序操作 |
| **T0** | (PC)->ABUS | PC\_enA=1 |
| (MEN)->IR | (MEM)\_RW=1,IRENA=1 |
| (PC)+1->PC CLK PC+1 | PC\_inc=1 |
| **T1** | halt |  |

**二、代码**

**top\_level**

library IEEE;

use IEEE.std\_logic\_1164.all;

entity top\_level is                ----声明实体外部接口

    port (

        clk, reset:

in  STD\_LOGIC;

        abusX:     out STD\_LOGIC\_VECTOR(7 downto 0); --数据总线输出

        dbusX:     out STD\_LOGIC\_VECTOR(7 downto 0);

        mem\_enDX, mem\_rwX:

 out STD\_LOGIC;

        pc\_enAX, pc\_ldX, pc\_incX:

out STD\_LOGIC;

        ir\_enAX, ir\_enDX, ir\_ldX:

 out STD\_LOGIC;

        acc\_enDX, acc\_ldX, acc\_selAluX:

 out STD\_LOGIC;

        acc\_QX:    out STD\_LOGIC\_VECTOR(7 downto 0);

        alu\_accZX:   out STD\_LOGIC;

        alu\_opX:   out STD\_LOGIC\_VECTOR(2 downto 0)

    );

end top\_level;

architecture topArch of top\_level is

component program\_counter

    port (

        clk, en\_A, ld, inc, reset: in STD\_LOGIC;

        aBus: out STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: in STD\_LOGIC\_VECTOR(7 downto 0)

    );

end component;

component instruction\_register

    port (

        clk, en\_A, en\_D, ld, reset: in STD\_LOGIC;

        aBus: out STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0);

        load, store, add, sub,mul,div,neg,andd,orr,nott, halt, branch: out STD\_LOGIC

           );

end component;

component accumulator

    port (

        clk, en\_D, ld, selAlu, reset: in STD\_LOGIC;

        aluD: in STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0);

        q: out STD\_LOGIC\_VECTOR(7 downto 0)

    );

end component;

component alu

    port (

        op: in STD\_LOGIC\_VECTOR(2 downto 0);

        accD: in STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: in STD\_LOGIC\_VECTOR(7 downto 0);

        result: out STD\_LOGIC\_VECTOR(7 downto 0);

        accZ: out STD\_LOGIC

    );

end component;

component ram

    port (

        r\_w, en, reset: in STD\_LOGIC;

        aBus: in STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0)

    );

end component;

component controller

    port (

     clk, reset:   in  STD\_LOGIC;

     mem\_enD, mem\_rw:   out STD\_LOGIC;

     pc\_enA, pc\_ld, pc\_inc:   out STD\_LOGIC;

     ir\_enA, ir\_enD, ir\_ld:   out STD\_LOGIC;

     ir\_load, ir\_store, ir\_add: in  STD\_LOGIC;

     ir\_sub,  ir\_mul, ir\_div:  in STD\_LOGIC;

     ir\_and,  ir\_or, ir\_not: in  STD\_LOGIC;

     ir\_neg, ir\_halt, ir\_branch:  in  STD\_LOGIC;

     acc\_enD, acc\_ld, acc\_selAlu:  out STD\_LOGIC;

     alu\_op:    out STD\_LOGIC\_VECTOR(2 downto 0)

    );

end component;

signal abus: STD\_LOGIC\_VECTOR(7 downto 0);

signal dbus: STD\_LOGIC\_VECTOR(7 downto 0);

signal mem\_enD, mem\_rw:   STD\_LOGIC;

signal pc\_enA, pc\_ld, pc\_inc:  STD\_LOGIC;

signal ir\_enA, ir\_enD, ir\_ld:  STD\_LOGIC;

signal ir\_load, ir\_store, ir\_add: STD\_LOGIC;

signal ir\_sub,  ir\_mul, ir\_div: STD\_LOGIC;

signal ir\_and, ir\_or, ir\_not: STD\_LOGIC;

signal ir\_negate, ir\_halt, ir\_branch: STD\_LOGIC;

signal acc\_enD, acc\_ld, acc\_selAlu: STD\_LOGIC;

signal acc\_Q:    STD\_LOGIC\_VECTOR(7 downto 0);

signal alu\_op:    STD\_LOGIC\_VECTOR(2 downto 0);

signal alu\_accZ:   STD\_LOGIC;

signal alu\_result:   STD\_LOGIC\_VECTOR(7 downto 0);

begin

  pc: program\_counter port map(clk, pc\_enA, pc\_ld, pc\_inc, reset, abus, dbus);

  ir: instruction\_register port map(clk, ir\_enA, ir\_enD, ir\_ld, reset, abus,dbus,ir\_load,ir\_store,ir\_add,ir\_sub,ir\_mul,ir\_div,ir\_and,ir\_or,ir\_not,ir\_negate, ir\_halt, ir\_branch );

  acc: accumulator port map(clk, acc\_enD, acc\_ld, acc\_selAlu, reset, alu\_result, dbus, acc\_Q);

  aluu: alu port map(alu\_op, acc\_Q, dbus, alu\_result, alu\_accZ);

  mem: ram port map(mem\_rw, mem\_enD, reset, abus, dbus);

  ctl: controller port map (

clk, reset, mem\_enD, mem\_rw, pc\_enA, pc\_ld, pc\_inc,

      ir\_enA, ir\_enD, ir\_ld, ir\_load, ir\_store, ir\_add,ir\_sub,

ir\_mul,ir\_div,ir\_and,ir\_or,ir\_not,

      ir\_negate, ir\_halt, ir\_branch,acc\_enD,

 acc\_ld, acc\_selAlu, alu\_op

);

   abusX <= abus;

   dbusX <= dbus;

   mem\_enDX <= mem\_enD;

   mem\_rwX <= mem\_rw;

   pc\_enAX <= pc\_enA;

   pc\_ldX <= pc\_ld;

   pc\_incX <= pc\_inc;

   ir\_enAX <= ir\_enA;

   ir\_enDX <= ir\_enD;

   ir\_ldX <= ir\_ld;

   acc\_enDX <= acc\_enD;

   acc\_ldX <= acc\_ld;

   acc\_selAluX <= acc\_selAlu;

   acc\_QX <= acc\_Q;

   alu\_opX <= alu\_op;

   alu\_accZX <= alu\_accZ;

end topArch;

**ram**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_arith.all;

entity ram is

    port (

        r\_w, en, reset: in STD\_LOGIC;

        aBus: in STD\_LOGIC\_VECTOR(7 downto 0);  ----数据总线输入

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0)

    );

end ram;

architecture ramArch of ram is

type ram\_typ is array(0 to 63) of STD\_LOGIC\_VECTOR(7 downto 0);

signal ram: ram\_typ;

begin

  process(

en, reset, r\_w, aBus, dBus

) begin

   if reset = '1' then

    ram(0) <= x"03";

    ram(1) <= x"44";

    ram(2) <= x"55";

    ram(3) <= x"02";

    ram(4) <= x"06";

    ram(5) <= x"02";

--    ram(6) <= x"20";

--    ram(7) <= x"06";

--ram(8) <= x"01";

  elsif r\_w = '0' then

    ram(conv\_integer(unsigned(aBus))) <= dBus;

   end if;

  end process;

  dBus <= ram(conv\_integer(unsigned(aBus)))

    when reset = '0' and en = '1' and r\_w = '1' else

   "ZZZZZZZZ";

end ramArch;

**program\_counter**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

entity program\_counter is

    port (

        clk, en\_A, ld, inc, reset: in STD\_LOGIC;

        aBus: out STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: in STD\_LOGIC\_VECTOR(7 downto 0)

                         );

end program\_counter;

architecture pcArch of program\_counter is

signal pcReg: STD\_LOGIC\_VECTOR(7 downto 0);

begin

  process(clk) begin

   if clk'event and clk = '1' then

    if reset = '1' then

     pcReg <= "00000000";

    elsif ld = '1' then

     pcReg <= dBus;

    elsif inc = '1' then

     pcReg <= pcReg + "00000001";

    end if;

   end if;

  end process;

  aBus <= pcReg when en\_A = '1' else "ZZZZZZZZ";

end pcArch;

library IEEE;

use IEEE.std\_logic\_1164.all;

**instruction\_register**

entity instruction\_register is          ----声明实体外部接口

    port (

        clk, en\_A, en\_D, ld, reset: in STD\_LOGIC;

        aBus: out STD\_LOGIC\_VECTOR(7 downto 0); ----数据总线输出

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0);

        load, store, add, sub,mul,div,andd,orr,nott,neg, halt, branch: out STD\_LOGIC

        );

end instruction\_register;

architecture irArch of instruction\_register is

signal irReg: STD\_LOGIC\_VECTOR(7 downto 0);

begin

  process(clk) begin

   if clk'event and clk = '0' then -- load on falling edge

    if reset = '1' then

     irReg <= "00000000";

    elsif ld = '1' then

     irReg <= dBus;

    end if;

   end if;

  end process;

  aBus <= "0000" & irReg(3 downto 0) when en\_A = '1' else

     "ZZZZZZZZ";

  dBus <= "0000" & irReg(3 downto 0) when en\_D = '1' else

     "ZZZZZZZZ";

  load    <= '1' when irReg(7 downto 4) = "0000"  else '0';

  store   <= '1' when irReg(7 downto 4) = "0001"  else '0';

  add     <= '1' when irReg(7 downto 4) = "0010"  else '0';

  sub     <= '1' when irReg(7 downto 4) = "0011"  else '0';

  mul     <= '1' when irReg(7 downto 4) = "0100"  else '0';

  div     <= '1' when irReg(7 downto 4) = "0101"  else '0';

  neg     <= '1' when irReg = "0110" & "0000"  else '0';

  andd    <= '1' when irReg (7 downto 4) = "0111" else '0';

  orr     <= '1' when irReg (7 downto 4) = "1000" else '0';

  nott    <= '1' when irReg (7 downto 4) = "1001" else '0';

  halt    <= '1' when irReg = "1010" & "0001"  else '0';

  branch  <= '1' when irReg(7 downto 4) = "1011"  else '0';

end irArch;

**controller**

library IEEE;

use IEEE.std\_logic\_1164.all;

entity controller is                       ----声明实体外部接口

    port (

     clk, reset:

  in  STD\_LOGIC;

     mem\_enD, mem\_rw:

out STD\_LOGIC;

     pc\_enA, pc\_ld, pc\_inc:   out STD\_LOGIC;

     ir\_enA, ir\_enD, ir\_ld:   out STD\_LOGIC;

     ir\_load, ir\_store, ir\_add: in  STD\_LOGIC;

     ir\_sub,ir\_mul,ir\_div:  in STD\_LOGIC;

     ir\_and,ir\_or, ir\_not:  in STD\_LOGIC;

     ir\_neg, ir\_halt, ir\_branch:  in  STD\_LOGIC;

     acc\_enD, acc\_ld, acc\_selAlu:  out STD\_LOGIC;

     alu\_op:    out STD\_LOGIC\_VECTOR(2 downto 0)

    );

end controller;

architecture controllerArch of controller is

type state\_type is ( reset\_state,

   fetch0, fetch1,

   load0, load1,

   store0, store1,

   add0, add1,

   sub0, sub1,

   mul0, mul1,

   div0, div1,

   and0, and1,

   or0,  or1,

   not0, not1,

   negate0, negate1,

   halt,

   branch0, branch1

   );

signal state: state\_type;

begin

  process(clk) begin

   if clk'event and clk = '1' then

    if reset = '1' then state <= reset\_state;

      else

       case state is

       when reset\_state => state <= fetch0;

       when fetch0 => state <= fetch1;

     when fetch1 =>

      if ir\_load = '1' then state <= load0;

      elsif ir\_store   = '1' then state <= store0;

      elsif ir\_add     = '1' then state <= add0;

      elsif ir\_sub     = '1' then state <= sub0;

      elsif ir\_mul     = '1' then state <= mul0;

      elsif ir\_div     = '1' then state <= div0;

      elsif ir\_and     = '1' then state <= and0;

      elsif ir\_or      = '1' then state <= or0;

      elsif ir\_not     = '1' then state <= not0;

      elsif ir\_neg     = '1' then state <= negate0;

      elsif ir\_halt    = '1' then state <= halt;

      elsif ir\_branch  = '1' then state <= branch0;

      end if;

     when load0 =>  state <= load1;

     when load1 =>  state <= fetch0;

     when store0 =>  state <= store1;

     when store1 => state <= fetch0;

     when add0 =>  state <= add1;

     when add1 =>  state <= fetch0;

     when sub0 =>  state <= sub1;

     when sub1 =>  state <= fetch0;

     when mul0 =>  state <= mul1;

     when mul1 =>  state <=fetch0;

     when div0 =>  state <=div1;

     when div1 =>  state <=fetch0;

     when and0 =>  state <=and1;

     when and1 =>  state <=fetch0;

     when or0  =>  state <=or1;

     when or1  =>  state <=fetch0;

     when not0 =>  state <=not1;

     when not1 =>  state <=fetch0;

     when negate0 => state <= negate1;

     when negate1 => state <= fetch0;

     when halt =>  state <= halt;

     when branch0 => state <= branch1;

     when branch1 => state <= fetch0;

     when others =>  state <= halt;

     end case;

    end if;

   end if;

  end process;

  process(clk) begin -- special process for memory write timing

   if clk'event and clk = '0' then

    if state = store0 then

     mem\_rw <= '0';

    else

     mem\_rw <= '1';

    end if;

   end if;

  end process;

  mem\_enD <= '1'   when state =  fetch0 or state =  fetch1 or

     state =   load0 or state =   load1 or

     state =    add0 or state =    add1 or

     state =    sub0 or state =    sub1 or

     state =    mul0 or state =    mul1 or

     state =    div0 or state =    div1 or

     state =    and0 or state =    and1 or

     state =    or0  or state =    or1 else '0';

  pc\_enA <= '1'    when state =  fetch0 or

          state = fetch1   else '0';

  pc\_ld <= '1'    when state = branch0

       else '0';

  pc\_inc <= '1'    when state = fetch1

       else '0';

  ir\_enA <= '1'    when state = load0 or

     state = load1 or state = store0 or

     state =  store1 or state =  add0 or

     state =  add1 or   state =    sub0 or

     state =    sub1 or state =    mul0 or

     state =    mul1 or state =    div0 or

     state =    div1 or state =    and0 or

     state =    and1 or state =    or0  or

     state =    or1        else '0';

  ir\_enD <= '1'    when state = branch0

       else '0';

  ir\_ld <= '1'    when state = fetch1

       else '0';

  acc\_enD <= '1'   when state =  store0 or state =  store1  else '0';

  acc\_ld <= '1'    when state =  load1 or state =   add1 or state = negate1

                   or state =  sub1 or state = mul1 or state = div1

                   or state = not1 or state = or1 or state = not1   else '0';

  acc\_selAlu <='1' when state = add1 or state = negate1 or state =  sub1 or

                    state = mul1 or state = div1  or state = not1 or

                    state = or1 or state = not1  else '0';

  alu\_op <= "000" when state = add0 or state = add1

       else "001" when state = sub0 or state = sub1

       else "010" when state = mul0 or state = mul1

       else "011" when state = div0 or state = div1

       else "100" when state = negate0 or state = negate1

       else "101" when state = and0 or state = and1

       else "110" when state = or0 or state = or1

       else "111" when state = not0 or state = not1;

  --alu\_op <= "01"   when state =  add0 or state =  add1  else "00";

end controllerArch;

**alu**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

use ieee.NUMERIC\_STD.all;

entity alu is                           -----实体声明外部接口

        generic (

     constant N: natural := 1  -- number of shited or rotated bits

    );

    port (

        op: in STD\_LOGIC\_VECTOR(2 downto 0);   --选择控制运算类型

        accD: in STD\_LOGIC\_VECTOR(7 downto 0); -- 累加器的8位数据

        dBus: in STD\_LOGIC\_VECTOR(7 downto 0); -- 数据总线用于运算

        result: out STD\_LOGIC\_VECTOR(7 downto 0);   --结果的输出

        accZ: out STD\_LOGIC

                         );

end alu;

architecture Behavioral of ALU is

signal ALU\_Result : std\_logic\_vector (7 downto 0);

signal tmp: std\_logic\_vector (8 downto 0);

begin

   process(accD,dBus,op)

 begin

  case(op) is

  when "000" => -- Addition

   ALU\_Result <= accD + dBus ;

  when "001" => -- Subtraction

   ALU\_Result <= accD - dBus ;

  when "010" => -- Multiplication

   ALU\_Result <= std\_logic\_vector(to\_unsigned((to\_integer(unsigned(accD)) \* to\_integer(unsigned(dBus))),8)) ;

  when "011" => -- Division

   ALU\_Result <= std\_logic\_vector(to\_unsigned(to\_integer(unsigned(accD)) / to\_integer(unsigned(dBus)),8)) ;

  when "100" => -- Logical shift left

   ALU\_Result <= std\_logic\_vector(unsigned(accD) sll N);

  when "101" => -- Logical shift right

   ALU\_Result <= std\_logic\_vector(unsigned(accD) srl N);

  when "110" => --  Rotate left

   ALU\_Result <= std\_logic\_vector(unsigned(accD) rol N);

  when "111" => -- Rotate right

   ALU\_Result <= std\_logic\_vector(unsigned(accD) ror N);

  when others => ALU\_Result <= accD + dBus ;

  end case;

 end process;

 result <= ALU\_Result; -- ALU out

 tmp <= ('0' & accD) + ('0' & dBus);

 accZ <= tmp(8); -- Carryout flag

end Behavioral;

**accumulator**

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

entity accumulator is                   ----声明外部实体接口

    port (

        clk, en\_D, ld, selAlu, reset: in STD\_LOGIC; --时钟信号

        aluD: in STD\_LOGIC\_VECTOR(7 downto 0);

        dBus: inout STD\_LOGIC\_VECTOR(7 downto 0);

        q: out STD\_LOGIC\_VECTOR(7 downto 0)

    );

end accumulator;

architecture accArch of accumulator is

signal accReg: STD\_LOGIC\_VECTOR(7 downto 0);

begin

  process(clk) begin

   if clk'event and clk = '1' then

    if reset = '1' then

     accReg <= "00000000";

    elsif ld = '1' and selAlu = '1' then

     accReg <= aluD;

    elsif ld = '1' and selAlu = '0' then

     accReg <= dBus;

    end if;

   end if;

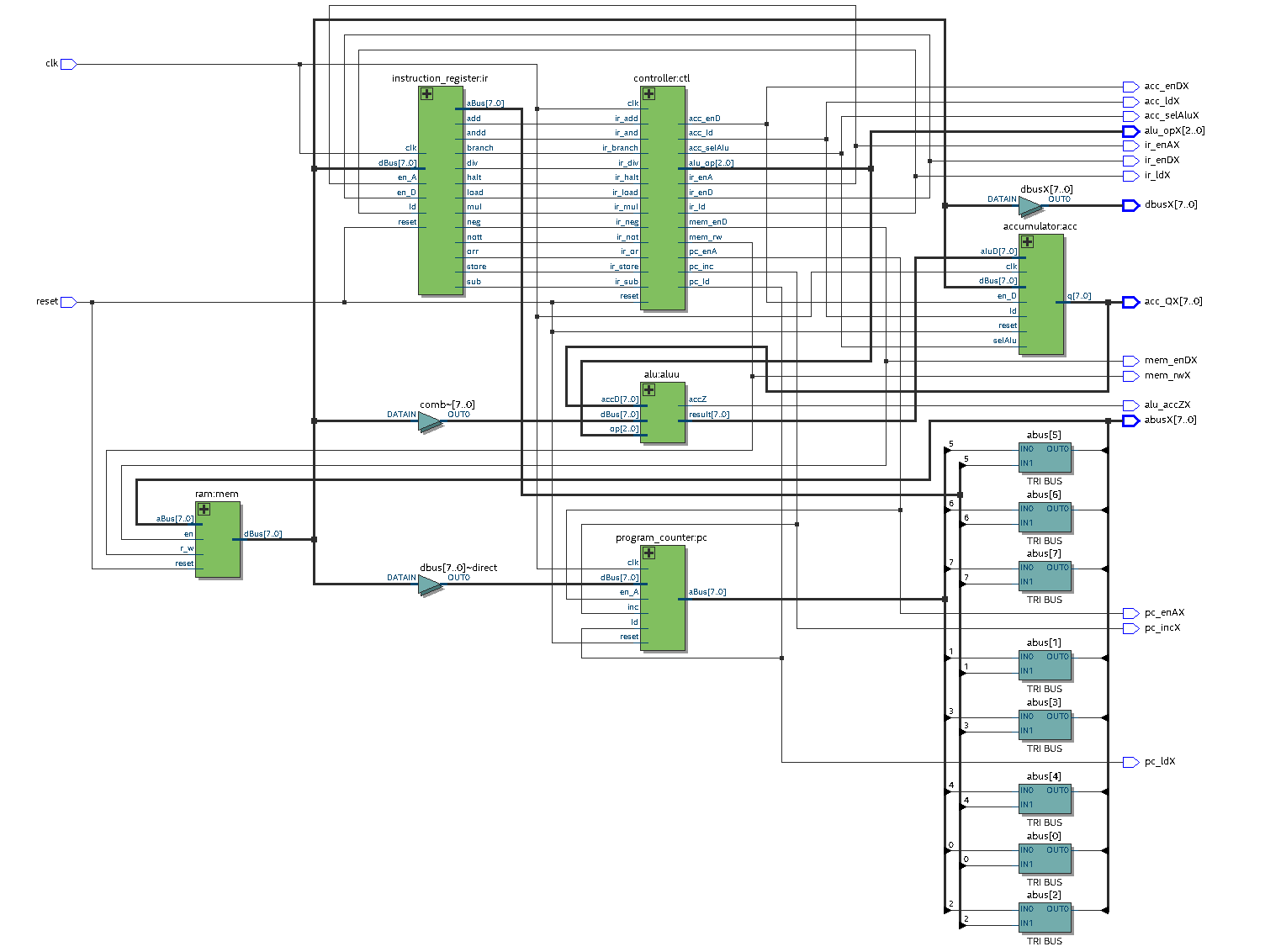
  end process;

  dBus <= accReg when en\_D = '1' else

     "ZZZZZZZZ";

  q <= accReg;

end accArch;

**三、电路图**

# 四、波形图结果验证

##### ****1、ram中测试程序说明：****

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 指令字 | 汇编指令 | 说明 |
| 0 | ram(0) <= x"14"; | STORE 4 | 把累加器acc的内容存入地址4的存储单元 |
| 1 | ram(1) <= x"30"; | SUB 0 | 累加器ACC中数据减去0号单元数据后结果存储到ACC中 |
| 2 | ram(2) <= x"25"; | ADD 5 | 累加器ACC中数据加上5号单元数据后结果存储到ACC中 |
| 3 | ram(3) <= x"15"; | STORE 5 | ACC中的数据存储到地址5存储单元 |
| 4 | ram(4) <= x"46"; | MUL 6 | 累加器ACC中数据乘以6号单元数据后结果存储到ACC中 |
| 5 | ram(5) <= x"31"; | STORE D | 把累加器acc的内容存入地址D的存储单元 |
| 6 | ram(6) <= x"55"; | DIV 5 | 将ACC中的值除以地址5的存储单元数据后存储到ACC中 |
| 7 | ram(7) <= x"06"; | LOAD 6 | 将地址6存储单元的值存入ACC中 |
| 8 | ram(8) <= x"01"; | LOAD 1 | 将地址1存储单元的值存入ACC中 |

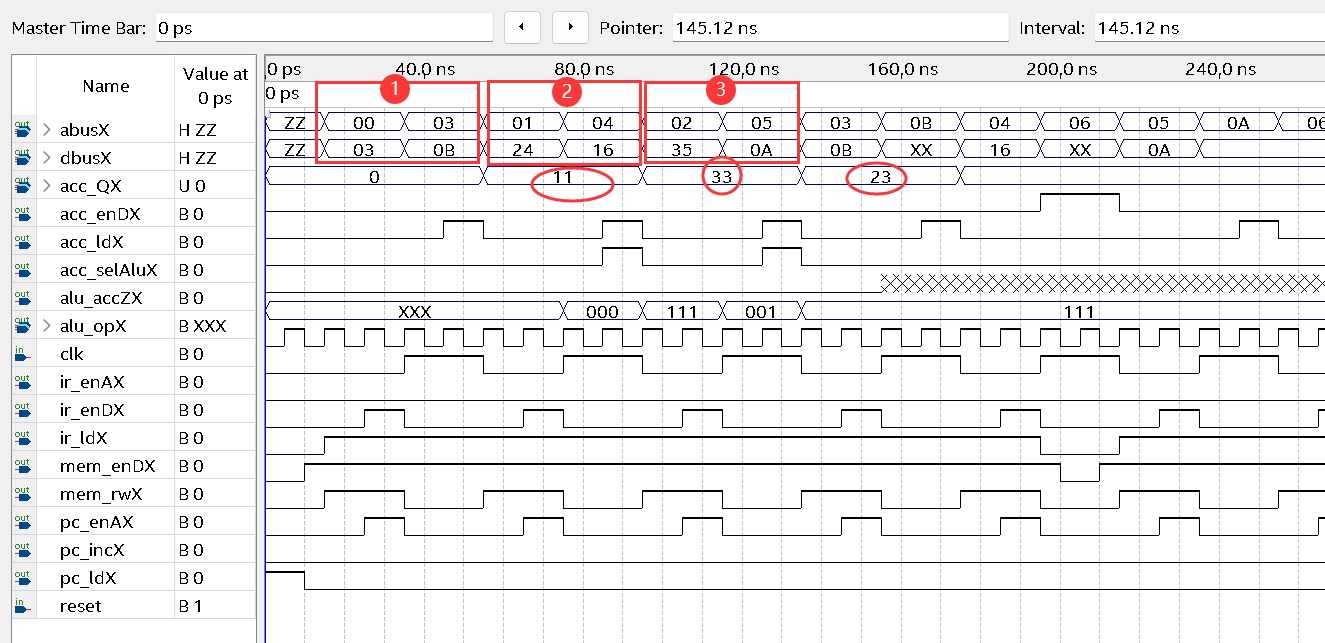
**波形图结果：**

**2、测试程序，波形图及指令的对应关系分析，数据运算过程分析**

（1）测试程序: 11+22-10

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 指令字 | 汇编指令 | 说明 |
| 0 | ram(0) <= x"03"; | Load 3 | 将地址3存储单元的值存入ACC中 |
| 1 | ram(1) <= x"24"; | Add 4 | 累加器ACC中数据加上4号单元数据后结果存储到ACC中 |
| 2 | ram(2) <= x"35"; | Sub 5 | 累加器ACC中数据减去5号单元数据后结果存储到ACC中 |

（2）程序数据运算过程分析，波形图及逐条指令、逐个部件逐个数据变化的对应关系分析。

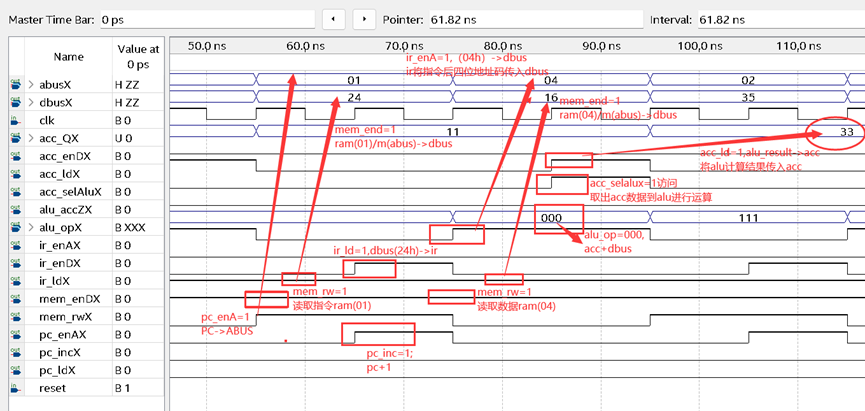
**波形图：**  
**程序运算过程分析：**

通过波形图可知指令1进行LOAD操作将存储在ram(3)的数据11d存入累加器acc中；

再通过指令2进行ADD操作，将存储在ram(4)的数据22d与累加器acc中的数据11d相加的结果即33d存回到累加器acc中；

最后再执行指令3进行SUB操作，将累加器acc中的数据32减去ram(5)中的数据10d得到结果23d存回到累加器acc中，至此我们实现了(*11+22-10)D*操作。

（3）以第二条add指令为例详细分析



*r\_w是存储器的读写控制线，当它为0是写操作，1时是读操作；en总线请求信号，当它为1是表示允许，为0时表示忙；ld是总线数据信号，当有总线上有数据时为1；selAlu是一个运算器访问的标志，如果运算器有访问为1；inc是pc寄存器加1的标志，当它为1时需加1；OP是一个选择控制信号，控制运算类型；aBus 表示地址总线；dBus 表示数据总线。*

 **取指周期的数据通路和微指令序列如下表**  
---------------------------------  
  数据通路          微指令序列  
----------------------------------  
fetch0  
1 (PC)→ABUS        Pc\_enA  
2  1->read         Mem\_rw  
3 M(ABUS)→DBUS     Mem\_enD  
fetch1   
4 (DBUS)→IR       Ir\_ld  
5 (PC)+1→PC       Pc\_inc

##### ****Fetch0阶段：****

② 控制器在时序信号的控制下转换状态到fetch0，

②此时Mem\_rw读有效；  Pc\_enA有效，完成PC将指令地址传给地址总线ABUS的操作，记作(PC)→ABUS；

③Mem\_enD有效，RAM通过数据总线DBUS将ABUS里数值所指单元的内容（指令）读取，记作M(ABUS)→DBUS；

此时地址总线abus为0000,0001(01H）-----存放指令的地址

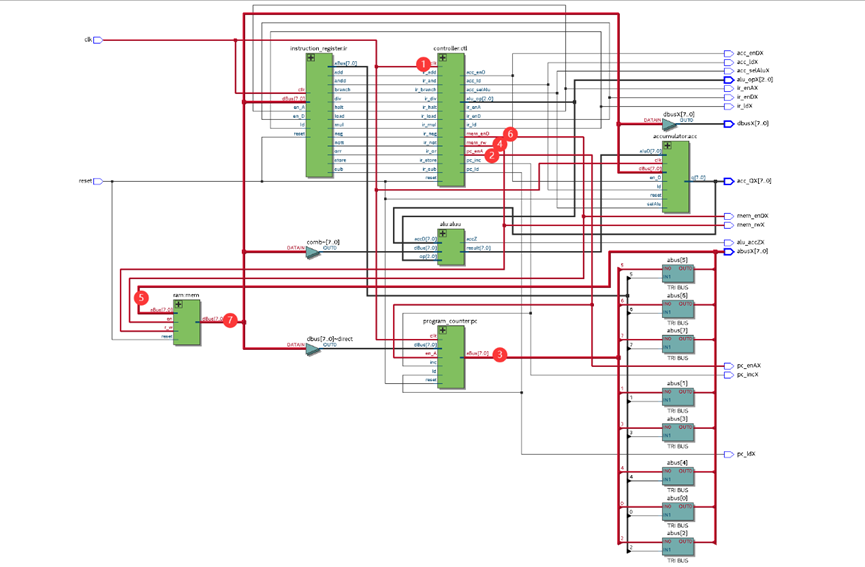
此时数据总线dbus为0010,0100( 24H）------指令

**该阶段数据流动过程**  
pcReg=”0000,0001b”->abusX=“00000001b(01h)”->ram(1)->dbusX=”0010,0100b(24h)”

**该阶段对应部件的实现操作代码**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段  （周期） | 部件 | 对应的控制信 号,数据，地址变化 | 对应的数据通路 | 对应的代码 |
| fetch0 | CU控制器 | clk = '1'  reset = '1' | fetch0; | process(clk) begin     if clk'event and clk = '1' then      if reset = '1' then state <= reset\_state;        else         case state is         when reset\_state => state <= fetch0; |
| PC程序计数器 | clk=0->1,  pc\_enAX=1,  pc\_ldX=0,  pc\_incX=0,  abusX=”01h” | (PC)→ABUS | aBus <= pcReg when en\_A = '1' else "ZZZZZZZZ"; |
| RAM静态存储器 | (mem)\_rwX=1,  mem\_enDX=1,  reset=0,  dbusx=”24h” | RAM(ABUS)→DBUS | dBus<=ram(conv\_integer(unsigned(aBus)))  when reset = '0' and en = '1' and r\_w = '1' else "ZZZZZZZZ"; |

**（fetch0）在电路图上的数据通路**



##### ****Fetch1阶段：****

1、cu控制器在时序信号的控制下转换状态到fetch1，记作fetch0->fetch1

2、Ir\_ld有效，将数据总线DBUS数值送至指令寄存器IR，记作(DBUS)→IR。

3、Pc\_inc有效，使PC内容加1，为下一条指令做准备，记作(PC)+1→PC；此时PC=00000010b(02h)

**该阶段数据流动过程**  
abusX=x”01h”；

dbusX=x”0010,0100b(24h)”->irReg=”0010,0100b(24h)”;

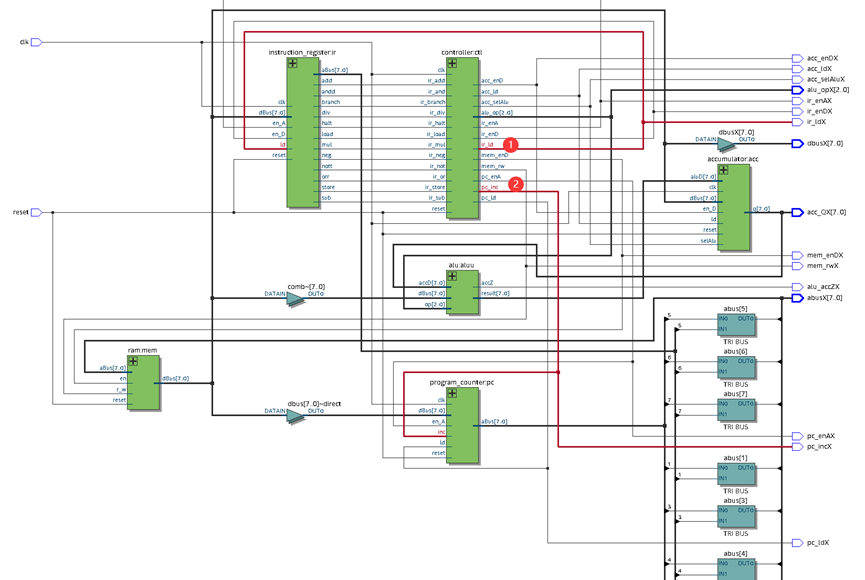
pcReg="0000,0001b”->“0000,0010b”

**该阶段对应部件的实现操作的代码**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段  （周期） | 部件 | 对应的控制信号,数据，地址变化 | 对应的数据通路 | 对应的代码 |
| fetch1 | CU控制器 | clk = '1'  reset = '0' | fetch0->fetch1 | when fetch0 => state <= fetch1; |
| PC程序计数器 | clk=0->1,  pc\_enAX=1,  pc\_ldX=0,  pc\_incX=1 | (PC)+1→PC | elsif inc = '1' then pcReg<=pcReg+"00000001"; |
| IR指令寄存器 | clk=0->1,  ir\_enDX=0,  ir\_ldX=1,  ir\_enAX=0,  irReg=” 00100100b;  ir\_load=1 | (DBUS)→IR | elsif ld = '1' then irReg <= dBus; |

|  |
| --- |
|  |
|  |  |

**（fetch1）对应在电路图上的数据通路**



取指执行周期的数据通路和微指令序列如下

---------------------------------  
  数据通路          微指令序列  
---------------------------------  
add0(分析取数)  
1 AD(IR)→ABUS       Ir\_enA  
2   read               Mem\_rw  
3 M(ABUS)→DBUS      Mem\_enD  
Add1（执行指令）  
1 (DBUS)+ACC->ACC      ir\_add,;acc\_selalu;Acc\_ld

##### ****ADD0阶段：****

（1）IR根据操作码部分数值0010对应add操作，将add=1信号给控制器；cu控制器转换状态到add0,

记作op(IR)->CU; fetch1->add0

（2）Mem\_rw读有效； Ir\_enA有效，将IR分析出的指令后四位地址码传入地址总线ABUS，

记作AD(IR)→ABUS；

（3）Mem\_enD有效，存储器RAM将ABUS的数值所指单元的内容（数据）送至DBUS，记M(ABUS)→DBUS；

指令0010,0100中操作码为0010-------ADD操作;  地址码为0100----4号存储单元

此时地址总线abus为0000,0100b(04h) ------存放运算数据的地址

数据总线dbus为0001,0110b(16h)(22D)-----运算的数据

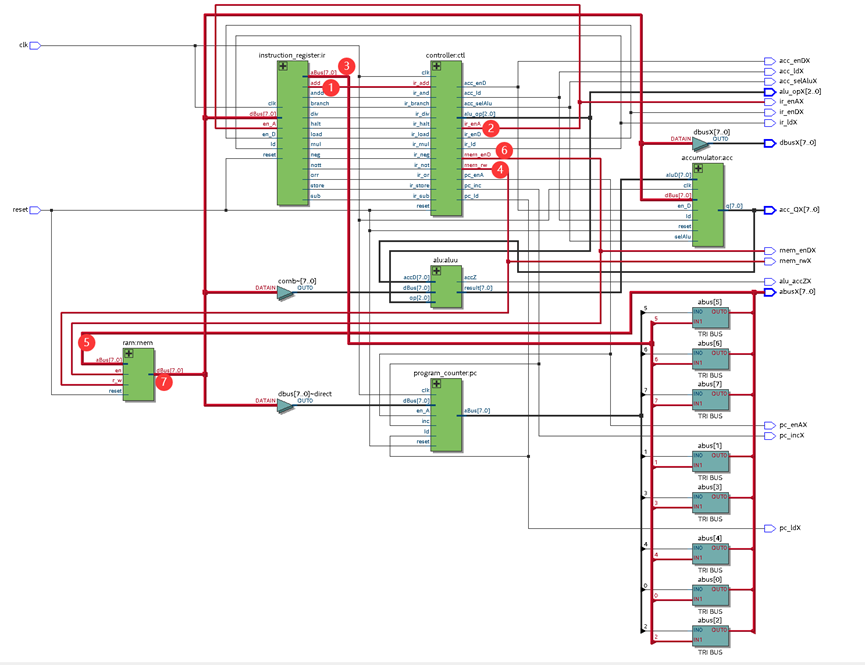
**该阶段的数据流动过程**

irReg="0010,0100b(24h)”->“0100b”&irReg[3:0]->abusX="0000,0100b(04h)”->ram(4)->dbusX="(16h)”

**该阶段对应部件的实现操作的代码**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段  （周期） | 部件 | 对应的控制信号,数据，地址变化 | 对应的数据通路 | 对应的代码 |
| Add0 | CU控制器 | clk = '1'  reset = '0'  ir\_add=1 | Fetch->add0 | when fetch1 =>  elsif ir\_add     = '1' then state <= add0; |
| IR指令寄存器 | clk=0->1,  ir\_enDX=0,  ir\_ldX=0,  ir\_enAX=1,  abusX=x”04” | (IR)→ABUS | aBus <= "0010" & irReg(3 downto 0) when en\_A = '1' else "ZZZZZZZZ"; |
| RAM静态存储器 | (mem)\_rwX=1,  mem\_enDX=1,  reset=0,  dbusX=x”16” | M(ABUS)→DBUS | dBus<=ram(conv\_integer(unsigned(aBus)))  when reset = '0' and en = '1' and r\_w = '1' else "ZZZZZZZZ"; |

**（add0）在电路图上的数据通路**



##### ****ADD1阶段：****

1、cu控制器在时序信号控制下转换状态到add1，记作add0 ->add1

2、acc\_selAlu=1，将acc数据11d输入到alu中

      acc\_op = 000，使运算器alu将数据总线dbus和acc中内容相加。

3、Acc\_ld有效，将运算结果传入acc, 记作(DBUS)+ACC->ACC

此时ACC\_QX为33(D)

至此执行周期执行完毕，该条指令的指令周期结束**。**

**该阶段的数据流动过程**

dbusX="16h(22d)”+acc\_qx="11d”-> acc\_qx="33d”

**该阶段对应部件的实现操作的代码**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 阶段（周期） | 部件 | 对应的控制信号,数据，地址变化 | 对应的数据通路 | 对应的代码 |
| Add1 | CU控制器 | clk = '1'  reset = '0' | add0->add1 | when add0 =>  state <= add1; |
| Alu运算器 | ir\_add=1  alu\_op=000 | (DBUS)+ACC | case(op) is  when "000" => -- Addition     ALU\_Result <= accD + dBus ; |
| ACC累    加器 | clk=0->1,  acc\_selAluX=1,  acc\_enDX=1,  acc\_ldX=1,  acc\_QX=”00100001” | (DBUS)+ACC→ACC | elsif ld = '1' and selAlu = '1' then       accReg <= aluD; |

**（add1）在电路图上的数据通路**

