目录

一.课程设计的目的及要求

1.1 目的

1.掌握 CPU 与内存数据交换的方法。
2.学会指令格式的设计与用汇编语言编写简易程序。
3.能够使用 VHDL 硬件描述语言在 Quartus II 软件环境下完成 CPU 模型机的设计。

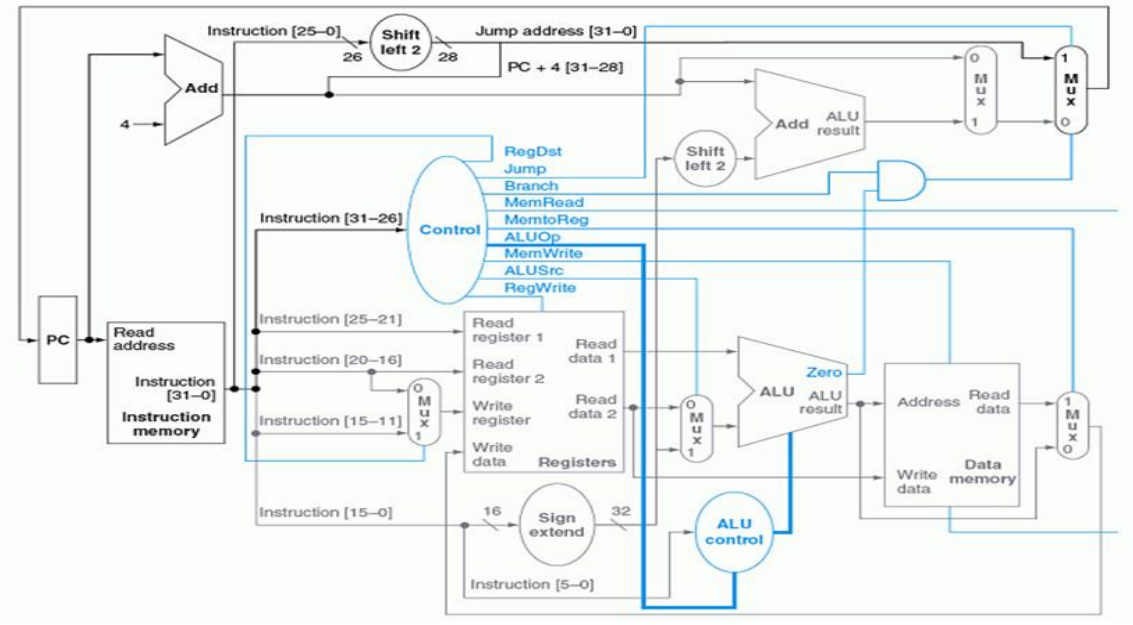
1.2 要求

1.处理器应支持 MIPS-Lite1 指令集。
2.MIPS-Lite1＝{addu，subu，ori，lw，sw，beq，jal}。
3.所有运算类指令均可以不支持溢出。
4.处理器为多周期设计。
5.多周期处理器由 datapath(数据通路)和 controller(控制器)组成。
6.数据通路应至少包括如下 module：PC(程序计数器)、、RF (通用寄存器组，也称为寄存器文件、寄存器堆)、ALU(算术逻辑单元)、EXT(扩展单元)、IM(指令存储器)、DM(数据存储器)等。

二.处理器的设计思想和设计内容

2.1 数据通路

CPU 结构数据通路如图 2-1

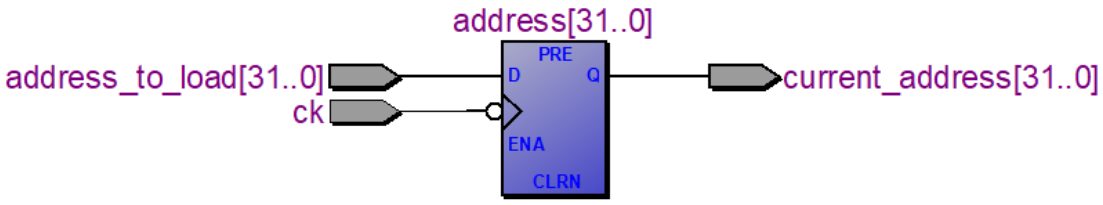


（图 2-1）

2.2 核心功能部件及其说明

2.2.1 指令计数器 PC

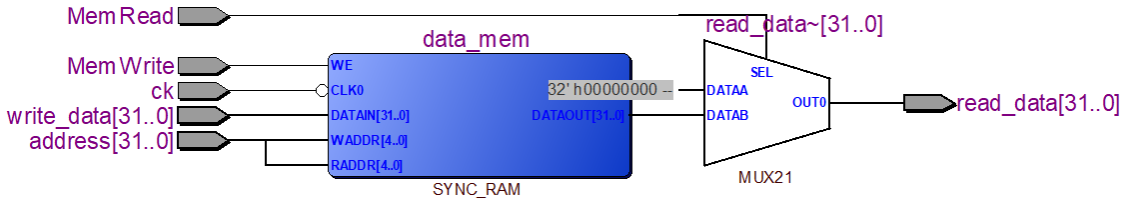
功能：指向下一条指令的程序计数器
RTL：图 2-2-2



（图 2-2-1）

2.2.2 内存 memory

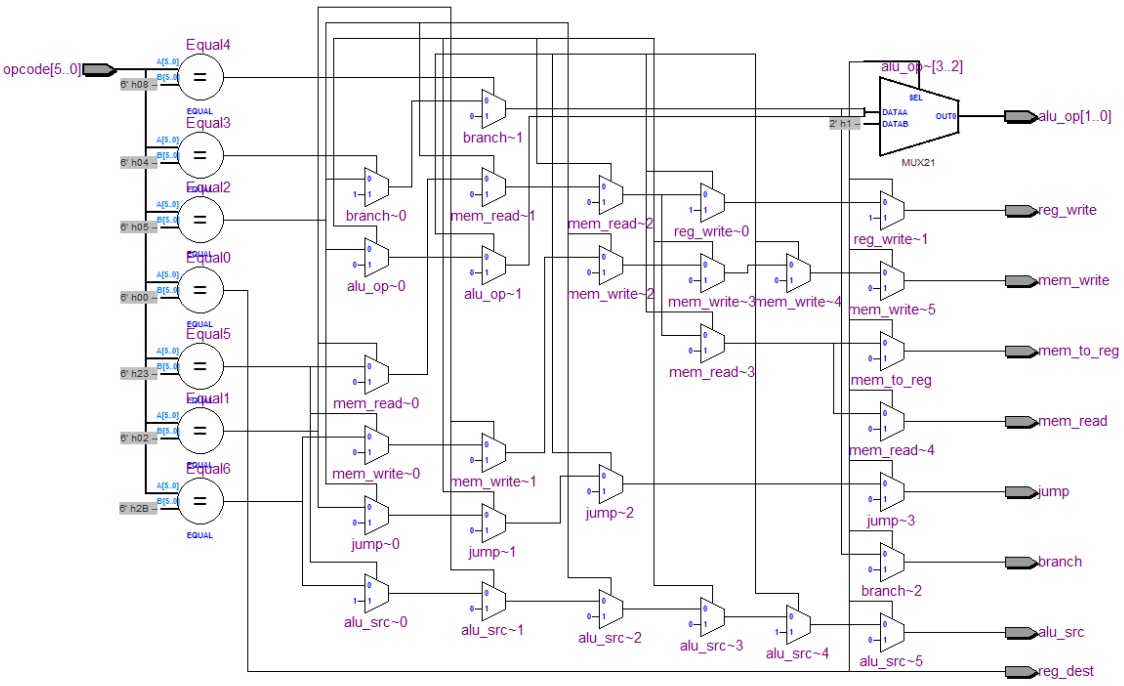
功能：内存模块，用于模拟计算机的 RAM，容量量为 128byte。
RTL：图 2-2-2



（图 2-2-2）

2.2.3 控制器 control

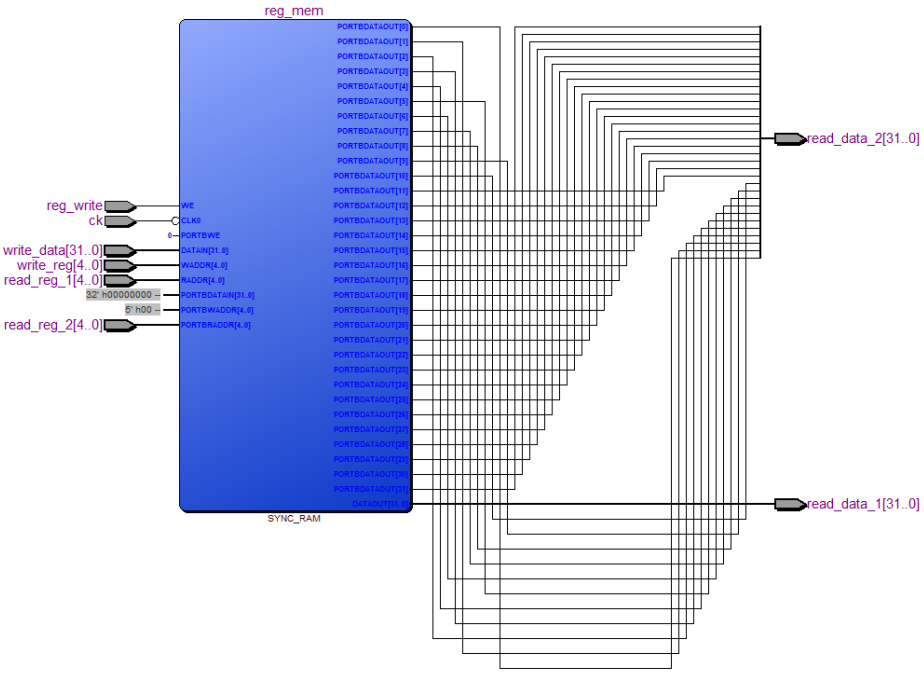
功能：主控制模块，用于解析操作码，发送控制信号
RTL：图 2-2-3



（图 2-2-3）

2.2.4 通用寄存器组 registers

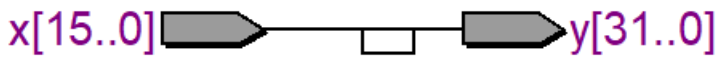
功能：32bit*32 个通用寄存器组
RTL：图 2-2-4



（图 2-2-4）

2.2.5 指令扩展器件 sign_extend

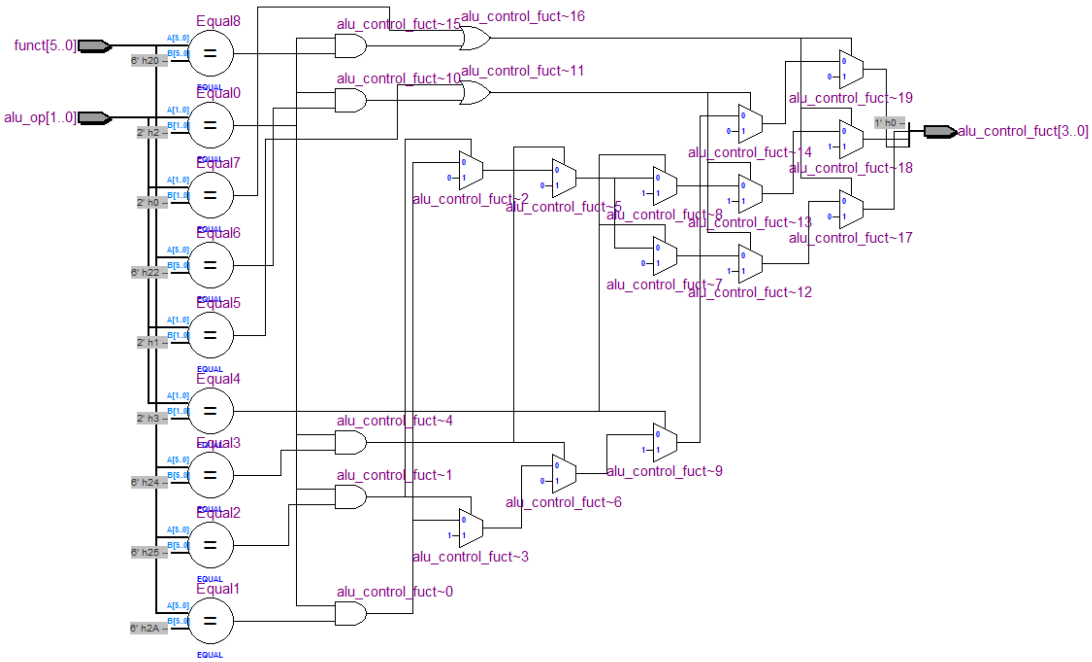
功能：扩展模块，16 位变 32 位
RTL：图 2-2-5



（图 2-2-5）

2.2.6 ALU 控制器 alu_control

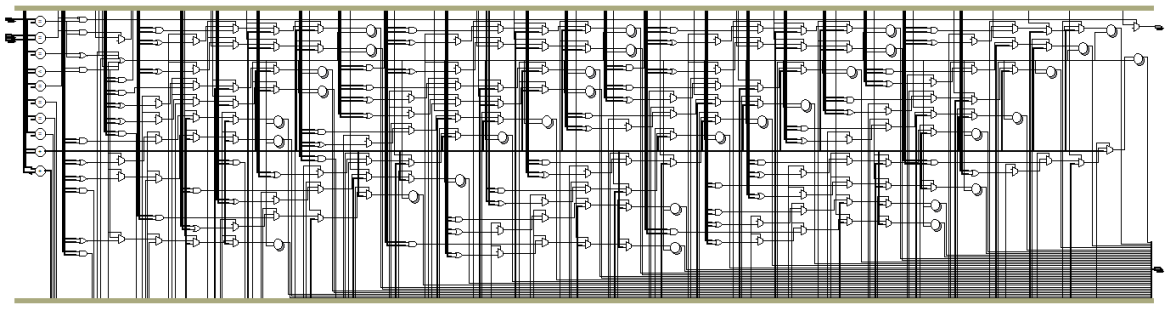
功能：ALU 专用的小型控制模块，用于从总控制模块中接收分离出来的操作指令，将其发送给 ALU 执行相应的运算操作。
RTL：图 2-2-6



（图 2-2-6）

2.2.7 ALU

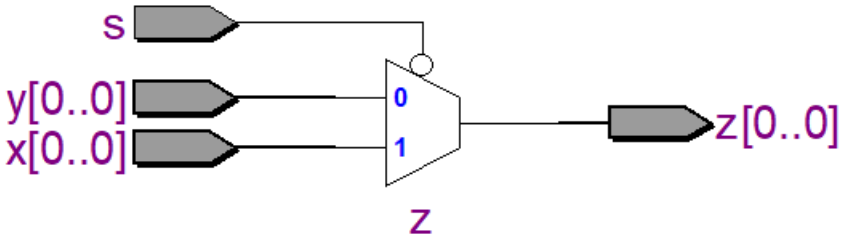
功能：ALU 模块，支持加、减、OR、AND、NOT 等常规操作
RTL：图 2-2-7



（图 2-2-7）

2.2.8 复用器 mux

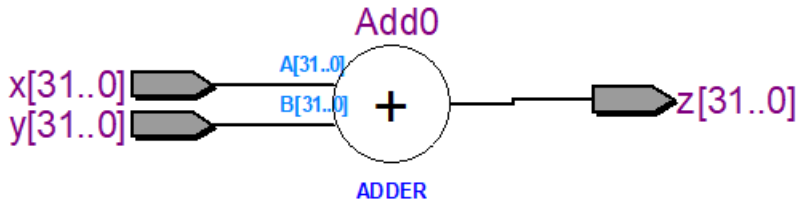
功能：简单的复用器，2 选 1
RTL：图 2-2-8



（图 2-2-8）

2.2.9 加法器 adder

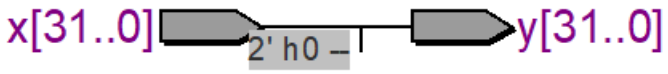
功能：一个独立的加法器，因为计算机在实际执行的过程中更多的执行的是加法指令，将加法部件从 ALU 独立出来，主要是为了提高数据处理速度。
RTL：图 2-2-9



（图 2-2-9）

2.2.10 移位器 shifter

功能：移位器。
RTL：图 2-2-10



（图 2-2-10）

2.2.11 指令存储器 instruction_memory

功能：这是一个内存装载模块，用于测试的时候将测试文件 instructions.txt 装载到模拟内存 instruction_memory.vhd

2.3 CPU 总架构

CPU 总体结构如图 2-3

（图 2-3）

三．设计处理器的结构和实现方法

3.1 总体结构

CPU 总线宽度位 32 位。5 周期非流水结构。包含 32 个通用寄存器组，通用寄存器组总容量 128byte。运行标准 MIPS 编码格式指令。支持 10 种指令（addi,beq,bne,jal,lw,sw,addu,and,and,or,slt）。支持直接寻址，立即寻址，变址寻址，寄存器寻址四种寻址方式。

3.2 主要实现思想

（1）我们这次分成很多的小模块，这样实现起来 debug 很快，因为代码分离了，所以更改起来也会更加方面，可以看到模块化的思想很重要；
（2）我们需要一个主模块将所有的子模块通过连线囊括起来，这个主模块有两个输入<1.CLK 时钟，2.Reset 重置信号>，在整个系统中，这个主模块起到一个提供数据通路（也就是图中所看到的一些线路）的角色。

3.3 元件接口定义

3.3.1 PC

表 3-3-1

信号名称	宽度	方向（I/O）	含义
ck	-	I	时钟信号
address_to_load	32	I	读指令地址
current_address	32	O	输出指令地址

3.3.2 control

表 3-3-2

信号名称	宽度	方向（I/O）	含义
opcode	6	I	操作码输入
reg_dest	1	O	寄存器片选
jump	1	O	跳转信号
branch	1	O	分支信号
mem_read	1	O	主存读使能
mem_to_reg	1	O	主存-寄存器使能
mem_write	1	O	主存写使能
alu_src	1	O	ALU 使能
reg_write	1	O	寄存器写使能

alu_op	2	O	ALU 操作码

3.3.3 memory

表 3-3-3

信号名称	宽度	方向（I/O）	含义
address	32	I	地址
write_data	32	I	写入数据
MemWrite	1	I	写使能
MemRead	1	I	读使能
ck	-	I	时钟信号
read_data	32	O	读出数据

3.3.4 registers

表 3-3-4

信号名称	宽度	方向（I/O）	含义
ck	-	I	时钟信号
reg_write	1	I	寄存器写使能
read_reg_1	5	I	寄存单元选择
read_reg_2	5	I	寄存单元选择
write_data	32	I	写入数据
read_data_1	32	O	读出数据 1
read_data_2	32	O	读出数据 2

3.3.5 sign_extend

表 3-3-5

信号名称	宽度	方向（I/O）	含义
x	16	I	16 位输入
y	32	O	32 位输出

3.3.6 alu_control

表 3-3-6

信号名称	宽度	方向（I/O）	含义
funct	6	I	来自操作码的 6 位操作信号

alu_op	2	I	ALU 操作码
alu_control_fuct	4	O	对应 ALU 操作的信号

3.3.7 ALU

表 3-3-7

信号名称	宽度	方向（I/O）	含义
in_1	32	I	操作数 1
in_2	32	I	操作数 2
alu_control_fuct	4	I	ALU 功能选择
Zero	1	O	判 0 输出
alu_result	32	O	运算结果

3.3.8 mux

表 3-3-8

信号名称	宽度	方向（I/O）	含义
x	n	I	多路数据选择器 n 选 1
y	n	I	
s	1	I	
z	n	O	

3.3.9 adder

表 3-3-9

信号名称	宽度	方向（I/O）	含义
x	32	I	操作数 1
y	32	I	操作数 2
z	32	O	结果输出

3.3.10 shifter

表 3-3-10

信号名称	宽度	方向（I/O）	含义
x	n1	I	逻辑左移
y	n2	O	

3.3.11 instruction_memory

表 3-3-11

信号名称	宽度	方向（I/O）	含义
read_address	32	I	读入一条指令
instruction	32	O	输出当前指令
last_instr_address	32	O	下一条指令地址

四．模型机的指令系统

4.1MIPS 指令格式说明

在本次课程设计中，在数据类型上只支持整数类型，在指令格式上直接 R、I 和 J 型指令。以下是对三型指令的简介：

（1） R（register）类型的指令从寄存器组中读取两个源操作数，计算结果写回寄存 器组。

（2） I（immediate）类型的指令使用一个 16 位的立即数作为一个源操作数。

（3） J（jump）类型的指令使用一个 26 位立即数作为跳转的目标地址。

〈1〉 op 表示指令操作码。

〈2〉 rs 为源操作数的寄存器号。

〈3〉 rd 为目的寄存器号，RT 既可为源寄存器号，也可为目的寄存器号。

（4） funct 可认为是扩展的操作码。

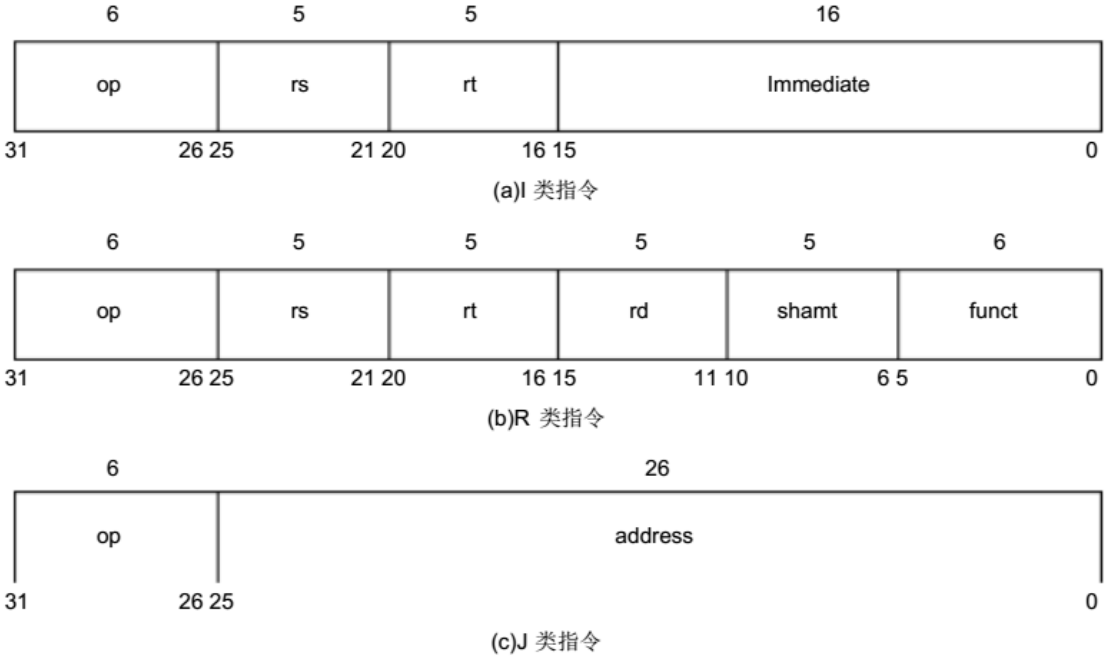
（5） shamte 由移位指令使用，定义移位位数。

（6） Immediate 是 16 位立即数，根据指令需求进行无符号或有符号扩展。

（7） Address 是 26 位立即数，由 J 型指令使用，用于产生跳转的目的地址。

4.2 指令划分

MIPS 指令集中 I、R、J 三类指令的划分如图 4-2



（图 4-2）

4.3 支持指令的详细说明

总共支持 10 条标准 MIPS 编码的指令。详细见下：

4.3.1 addi

指令格式：addi $1,$2,100->$1=$2+100
指令说明： rt <- rs + (sign-extend)immediate ；其中 rt=$1,rs=$2
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP						rs					rt					immediate																

4.3.2 beq

指令格式：beq $1,$2,10-> if($1==$2) goto PC+4+40
指令说明：if (rs == rt) PC <- PC+4 + (sign-extend)immediate<<2
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

OP	rs	rt	immediate

4.3.3 bne

指令格式：bne $1,$2,10-> if($1!=$2) goto PC+4+40
指令说明：if (rs != rt) PC <- PC+4 + (sign-extend)immediate<<2
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP						rs					rt					immediate																

4.3.4 jal

指令格式：jal 10000->$31<-PC+4 goto 10000
指令说明：$31<-PC+4; PC <- (PC+4)[31..28],address,0,0; address=10000/4
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op						address																									

4.3.5 lw

指令格式： lw $1,10($2)-> $1=memory[$2+10]
指令说明：rt <- memory[rs + (sign-extend)immediate] ; rt=$1,rs=$2
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP						rs					rt					immediate																

4.3.6 sw

指令格式：sw $1,10($2)-> memory[$2+10]=$1
指令说明：memory[rs + (sign-extend)immediate] <- rt ; rt=$1,rs=$2
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OP						rs					rt					immediate																

4.3.7 addu

指令格式： addu $1,$2,$3->$1=$2+$3
指令说明：rd <- rs + rt ；其中 rs＝$2，rt=$3, rd=$1,无符号数
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op						rs					rt					rd					shamt						func					

4.3.8 and

指令格式： and $1,$2,$3->$1=$2 & $3
指令说明： rd <- rs & rt ；其中 rs＝$2，rt=$3, rd=$1
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op						rs					rt					rd					shamt						func					

4.3.9 or

指令格式： or $1,$2,$3->$1=$2 | $3
指令说明：rd <- rs | rt ；其中 rs＝$2，rt=$3, rd=$1

编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op						rs					rt					rd					shamt					func					

4.3.10 slt

指令格式：slt $1,$2,$3-> if($2<$3) $1=1 else $1=0
指令说明：if (rs < rt) rd=1 else rd=0 ；其中 rs＝$2，rt=$3, rd=$1
编码方式：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
op						rs					rt					rd					shamt					func					

五．处理器的状态跳转操作过程

5.1 周期划分

5.1.1 取指令周期（IF）

以程序计数器 PC 中的内容作为地址，从存储器中取出指令并放 入指令寄存器 IR；同时 PC 值加 4（假设每条指令占 4 个字节），指向顺序的下一条指令。

5.1.2 指令译码/读寄存器周期（ID）

根据标准 MIPS 指令编码格式对指令进行译码，并用 IR 中的寄存器地址去访问通用寄存器组，读出所需的操作数。

5.1.3 执行/有效地址计算周期（EX）

在这个周期，ALU 对在上一个周期准备好的操作 数进行运算或处理。不同指令所进行的操作不同。

5.1.4 存储器访问/分支完成周期（MEM）

load 指令用上一个周期计算出的有效地址从存储器中读出相应的数据；store 指令把指定的数据写入这个有效地址所指出的存储器单元；分支指令若分支成功就把钱一个周期中计算好的转移目标地址送入 PC，否则不进行任何操作；其他类型的指令在该周期不做任何操作。

5.1.5 写回周期（WB）

把结果写入通用寄存器组。

5.2 微操作

5.2.1 取指令周期（IF）

IR←Mem[PC]
NPC←PC+1
以 PC 中的值从指令 cache 中取出一条指令，放入指令寄存器 IR；同时 PC 值加 1，然后放入 NPC，这时 NPC 中的值为顺序的下调指令的地址。

5.2.2 指令译码/读寄存器周期（ID）

A←Regs[rs]
B←Regs[rt]
imm←((IR16)16##IR16..31)
对指令进行译码，并以指令中的 rs 和 rt 字段作为地址访问通用寄存器组，将读出的数据让如 A 和 B 寄存器中。同时 IR 的低 16 位进行有符号或者无符号扩展，然后存入 Imm 寄存器

5.2.3 执行/有效地址计算周期（EX）

① LW 和 SW 指令
ALUo←A+Imm
ALU 将操作数相加形成有效地址，并存入临时寄存器 ALUo
② R-TYPE
ALUo←Afunct B
ALU 根据 funct 字段指出的操作类型对 A 和 B 中的数据进行运算，并将结果存入 ALUo
③I-TYPE
ALUo←A op Imm
ALU 根据操作码 op 指出的操作类型对 A 和 Imm 中的数据进行运算，并将结果存入 ALUo
④分支指令　ALUo←NPC+Imm　ALU 将临时寄存器 NPC 和 Imm 中的值相加得到转移目标的地址，存入 ALUo

5.2.4 存储器访问/分支完成周期（MEM）

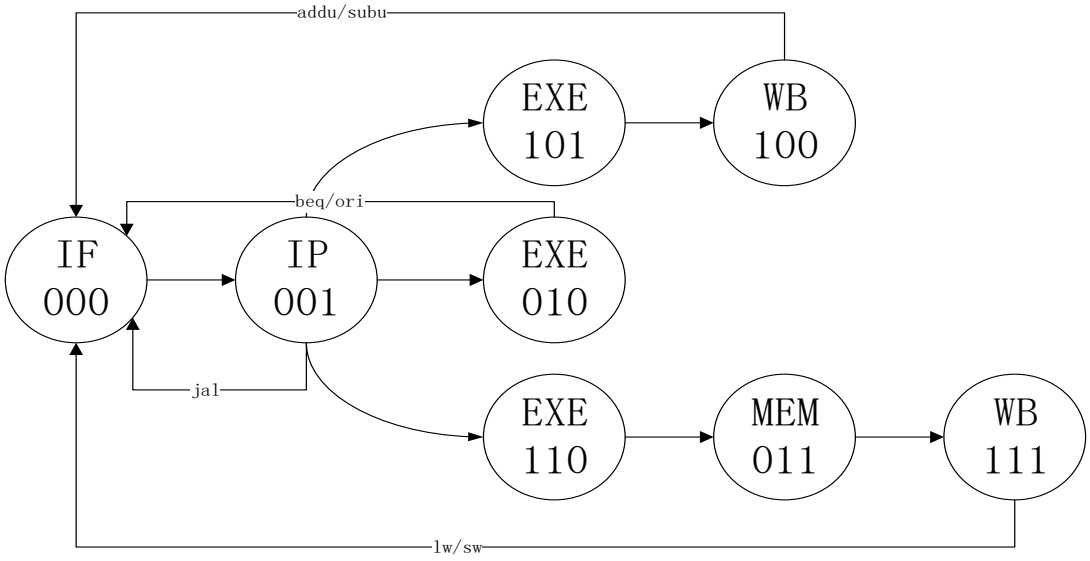
①LW 和 SW 指令
LW：LMD←Mem[ALUo]
即从存储器中读出相应数据，放入临时寄存器 LMD 中
SW：Mem[ALUo]←B
即把 B 中数据写入存储器
②分支指令
If (cond) PC←ALUo else PC←NPC
若 cond 中的内容为真，则将 ALUo 中的转移目标地址放入 PC，否则 PC+1。

5.2.5 写回周期（WB）

① R-TYPE
Regs[rd]←ALUo
② I-TYPE
Regs[rt]←ALUo
③LW 指令
Regs[rt]←LMD

5.3 状态转移图

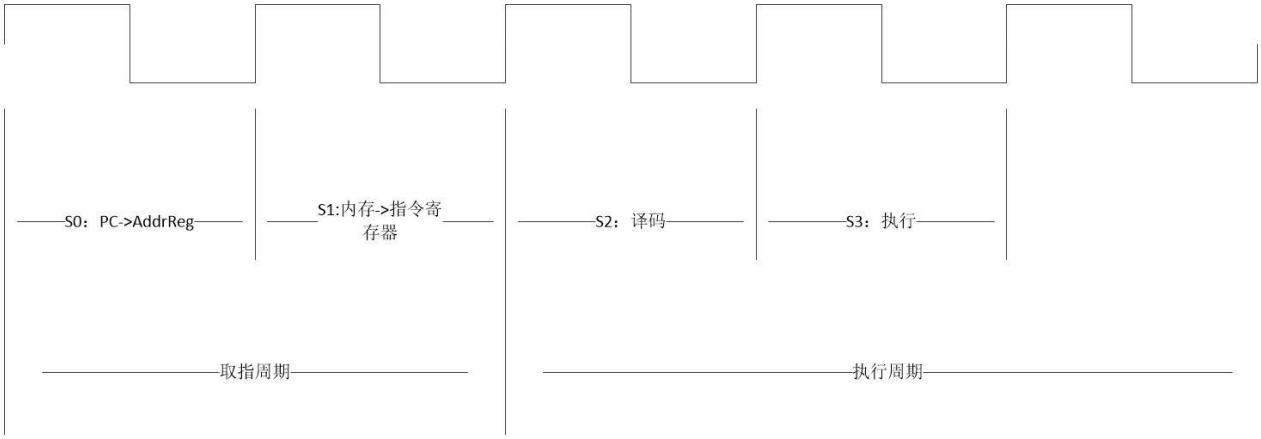
具体的状态转移图如图 5-3



（图 5-3）

5.4 节拍的划分

由于指令是变长的，所以并没有固定的指令周期，但前三个周期是一样的。如图 5-4



（图 5-4）

5.5 状态转移对应的操作

处理器的状态跳转操作过程

表 5-5

周期名称	R 类指令 OP=“R 类”	存储器访问指令 OP=“lw”或 OP=“sw”	分支指令 OP=“beqz”
IF	操作: IR←IM[PC] PC+4 控制型号:IRWrite=1		
ID	操作: A<-Regs[rs] B<-Regs[rt] Imm<-（IR 按符号位扩展为 32 位}） 控制信号:不需要		
EX	操作: ALUo<-A funct B 控制信号: ALUSrcA=1 ALUSrcB=00 ALUOp=10	操作: ALUoo<-A + imm 控制信号: ALUSrcA=1 ALUSrcB=01 ALUOp=00	操作: ALUo<-PC + (imm<<2) Cond<-(A==0) 控制信号: ALUSrcA=0 ALUSrcB=10 ALUOp=00
MEM	操作： Regs[rd]←ALUo 控制信号： DMtoReg=0 RegDst=1 RegWrite=1	load 指令： 操作； LMD<-DM[ALUo] 控制信号： DMRead=1 PCWrite=1	操作： If（cond&Branch） PC<-ALUo Else PC<-PC+4 控制信号：

	PCWrite=1	Store 指令： 操作： DM[ALUo]<-B 控制信号： DMWrite=1 PCWrite=1	Branch=1 PCWrite=1
WD		操作： load 指令: Regs[rt]←LMD 控制信号： DMtoReg=1 RegDst=0 RegWrite=1	

六．主要部件的 VHDL 代码实现

6.1 adder.vhd

```
--加法器，32bit
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adder is
	port (
		x,y: in std_logic_vector(31 downto 0);
		z: out std_logic_vector(31 downto 0)
	);
end entity;

architecture beh of adder is
	begin
	z <= x+y;
end beh;
```

6.2 alu.vhd

```
--ALU

library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu is
	port (
		in_1, in_2: std_logic_vector(31 downto 0);
		alu_control_fuct: in std_logic_vector(3 downto 0);--功能选择
		zero: out std_logic;--判零输出
		alu_result: out std_logic_vector(31 downto 0)
	);
end alu;

architecture beh of alu is
	signal and_op: std_logic_vector(3 downto 0):= "0000";--and
	signal or_op: std_logic_vector(3 downto 0):= "0001";--or
	signal add: std_logic_vector(3 downto 0):= "0010";--+
	signal subtract_not_equal: std_logic_vector(3 downto 0):= "0011";--≠
	signal subtract: std_logic_vector(3 downto 0):= "0110";---
	signal set_on_less_than: std_logic_vector(3 downto 0):= "0111";

	begin

	alu_result <=  in_1 + in_2 when(alu_control_fuct=add) else
				in_1 - in_2 when(alu_control_fuct=subtract or alu_control_fuct=subtract_not_equal) else
				in_1 and in_2 when(alu_control_fuct=and_op) else
				in_1 or in_2 when(alu_control_fuct=or_op) else
				"00000000000000000000000000000001"  when(alu_control_fuct=set_on_less_than  and
in_1 < in_2) else
				"00000000000000000000000000000000" when(alu_control_fuct=set_on_less_than);

	zero <=  '1' when(in_1/=in_2 and alu_control_fuct=subtract_not_equal) else
		'0' when(in_1=in_2 and alu_control_fuct=subtract_not_equal) else
		'1' when(in_1=in_2) else
		'0';

end beh;
```

6.3 alu_control.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity alu_control is
```

```
    port (
        funct: in std_logic_vector(5 downto 0);
        alu_op: in std_logic_vector(1 downto 0);
        alu_control_fuct: out std_logic_vector(3 downto 0)
    );
end alu_control;

architecture beh of alu_control is
    signal and_op: std_logic_vector(3 downto 0):= "0000";
    signal or_op: std_logic_vector(3 downto 0):= "0001";
    signal add: std_logic_vector(3 downto 0):= "0010";
    signal subtract_not_equal: std_logic_vector(3 downto 0):= "0011";
    signal subtract: std_logic_vector(3 downto 0):= "0110";
    signal set_on_less_than: std_logic_vector(3 downto 0):= "0111";

    begin

    alu_control_fuct <= add when(alu_op="00" or (alu_op="10" and funct="100000")) else
                        subtract when(alu_op="01" or (alu_op="10" and funct="100010")) else
                        subtract_not_equal when(alu_op="11") else
                        and_op when(alu_op="10" and funct="100100") else
                        or_op when(alu_op="10" and funct="100101") else
                        set_on_less_than when(alu_op="10" and funct="101010") else
                        "0000";


end beh;
```

6.4 control.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity control is
    port (
        opcode: in std_logic_vector(5 downto 0);
        reg_dest,jump, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write:
out std_logic;
        alu_op: out std_logic_vector(1 downto 0)
    );
end control;

architecture beh of control is
```

```
    begin

    -- The consequences of vhdl syntax
    --                          R-types                          addi
beq                                    bne                                    jump
lw                                  sw
    reg_dest <=   '1' when opcode="000000"  else '0' when opcode="001000"  else '0'
when opcode="000100"  else '0' when opcode="000101"  else '0' when opcode="000010"
else '0' when opcode="100011"  else '0' when opcode="101011"  else '0';
    jump <=          '0' when opcode="000000"  else '0' when opcode="001000"  else
'0'  when  opcode="000100"   else  '0'  when  opcode="000101"   else  '1'  when
opcode="000010"  else '0' when opcode="100011"  else '0' when opcode="101011"  else
'0';
    branch <=     '0' when opcode="000000"  else '0' when opcode="001000"  else '1'
when opcode="000100"  else '1' when opcode="000101"  else '0' when opcode="000010"
else '0' when opcode="100011"  else '0' when opcode="101011"  else '0';
    mem_read <=      '0' when opcode="000000"  else '0' when opcode="001000"  else
'0'  when  opcode="000100"   else  '0'  when  opcode="000101"   else  '0'  when
opcode="000010"  else '1' when opcode="100011"  else '0' when opcode="101011"  else
'0';
    mem_to_reg <=    '0' when opcode="000000"  else '0' when opcode="001000"  else
'0'  when  opcode="000100"   else  '0'  when  opcode="000101"   else  '0'  when
opcode="000010"  else '1' when opcode="100011"  else '0' when opcode="101011"  else
'0';
    mem_write <=  '0' when opcode="000000"  else '0' when opcode="001000"  else '0'
when opcode="000100"  else '0' when opcode="000101"  else '0' when opcode="000010"
else '0' when opcode="100011"  else '1' when opcode="101011"  else '0';
    alu_src <=       '0' when opcode="000000"  else '1' when opcode="001000"  else
'0'  when  opcode="000100"   else  '0'  when  opcode="000101"   else  '0'  when
opcode="000010"  else '1' when opcode="100011"  else '1' when opcode="101011"  else
'0';
    reg_write <=  '1' when opcode="000000"  else '1' when opcode="001000"  else '0'
when opcode="000100"  else '0' when opcode="000101"  else '0' when opcode="000010"
else '1' when opcode="100011"  else '0' when opcode="101011"  else '0';
    alu_op <=     "10" when opcode="000000" else "00" when opcode="001000" else "01"
when opcode="000100" else "11" when opcode="000101" else "00" when opcode="000010"
else "00" when opcode="100011"  else "00" when opcode="101011"  else "00";

end beh;
```

6.5 instruction_memory.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.numeric_std.all;
use STD.textio.all; -- Required for freading a file

entity instruction_memory is
    port (
        read_address: in STD_LOGIC_VECTOR (31 downto 0);
        instruction, last_instr_address: out STD_LOGIC_VECTOR (31 downto 0)--输出当前指令和下一条指令的地址
    );
end instruction_memory;


architecture behavioral of instruction_memory is

    -- 指令寄存器大小 128 byte：32bit(4byte)*32 行
    type mem_array is array(0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
    signal data_mem: mem_array := (
        "00000000000000000000000000000000", -- initialize data memory
        "00000000000000000000000000000000", -- mem 1
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- mem 10
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- mem 20
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
```

```
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- mem 30
        "00000000000000000000000000000000"
);

begin

-- 从指令文件中读取指令进内存
process
        file file_pointer : text;
        variable line_content : string(1 to 32);
        variable line_num : line;
        variable i: integer := 0;
        variable j : integer := 0;
        variable char : character:='0';

        begin
        -- 从 instructions.txt 文件中读指令
        file_open(file_pointer, "instructions.txt", READ_MODE);
        -- 一直读到文件尾
        while not endfile(file_pointer) loop
                readline(file_pointer,line_num); -- 从文件中读取一行（一条指令）
                READ(line_num,line_content);
                --以 bit 为划分存入内存
                for j in 1 to 32 loop
                        char := line_content(j);
                        if(char = '0') then
                                data_mem(i)(32-j) <= '0';
                        else
                                data_mem(i)(32-j) <= '1';
                        end if;
                end loop;
                i := i + 1;
        end loop;
        if i > 0 then
                last_instr_address <= std_logic_vector(to_unsigned((i-1)*4, last_instr_address'length));
        else
                last_instr_address <= "00000000000000000000000000000000";
        end if;

        file_close(file_pointer); -- 关闭指令文件
        wait;
```

```
    end process;
    instruction <= data_mem(to_integer(unsigned(read_address(31 downto 2))));

end behavioral;
```

6.6 main.vhd

```
--主接线模块
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity main is
    port(
        ck: in std_logic
    );
end main;

architecture beh of main is
    signal instr_address: std_logic_vector(31 downto 0); -- Address of the instruction to run
    signal next_address: std_logic_vector(31 downto 0); -- Next address to be loaded into PC
    signal instruction: std_logic_vector(31 downto 0); -- The actual instruction to run
    signal read_data_1, read_data_2, write_data, extended_immediate, shifted_immediate, alu_in_2, alu_result, last_instr_address, incremented_address, add2_result, mux4_result, concatenated_pc_and_jump_address, mem_read_data: std_logic_vector(31 downto 0):= "00000000000000000000000000000000"; -- vhdl does not allow me to port map " y => incremented_address(31 downto 28) & shifted_jump_address "
    signal shifted_jump_address: std_logic_vector(27 downto 0);
    signal jump_address: std_logic_vector(25 downto 0);
    signal immediate: std_logic_vector(15 downto 0);
    signal opcode, funct: std_logic_vector(5 downto 0);
    signal rs, rt, rd, shampt, write_reg: std_logic_vector(4 downto 0);
    signal alu_control_fuct: std_logic_vector(3 downto 0);
    signal reg_dest, jump, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write, alu_zero, branch_and_alu_zero: std_logic:= '0'; -- vhdl does not allow me to port map " s => (branch and alu_zero) "
    signal alu_op: std_logic_vector(1 downto 0);

     -- Enum for checking if the instructions have loaded
    type state is (loading, running, done);
    signal s: state:= loading;

    -- The clock for the other components; starts when the state is ready
    signal en: std_logic:= '0';
```

```
-- Load the other components
component pc
    port (
        ck: in std_logic;
        address_to_load: in std_logic_vector(31 downto 0);
        current_address: out std_logic_vector(31 downto 0)
    );
end component;


component instruction_memory
    port (
        read_address: in STD_LOGIC_VECTOR (31 downto 0);
        instruction, last_instr_address: out STD_LOGIC_VECTOR (31 downto 0)
    );
end component;


component registers
    port (
        ck: in std_logic;
        reg_write: in std_logic;
        read_reg_1, read_reg_2, write_reg: in std_logic_vector(4 downto 0);
        write_data: in std_logic_vector(31 downto 0);
        read_data_1, read_data_2: out std_logic_vector(31 downto 0)
    );
end component;


component control
    port (
        opcode: in std_logic_vector(5 downto 0);
        reg_dest,jump, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write: out std_logic;
        alu_op: out std_logic_vector(1 downto 0)
    );
end component;


component mux
    generic (n: natural:= 1);
    port (
        x,y: in std_logic_vector(n-1 downto 0);
        s: in std_logic;
```

```
            z: out std_logic_vector(n-1 downto 0)
        );
end component;


component alu_control
        port (
            funct: in std_logic_vector(5 downto 0);
            alu_op: in std_logic_vector(1 downto 0);
            alu_control_fuct: out std_logic_vector(3 downto 0)
        );
end component;


component sign_extend
        port (
            x: in std_logic_vector(15 downto 0);
            y: out std_logic_vector(31 downto 0)
        );
end component;


component alu
        port (
            in_1, in_2: std_logic_vector(31 downto 0);
            alu_control_fuct: in std_logic_vector(3 downto 0);
            zero: out std_logic;
            alu_result: out std_logic_vector(31 downto 0)
        );
end component;


component shifter
        generic (n1: natural:= 32; n2: natural:= 32; k: natural:= 2);
        port (
            x: in std_logic_vector(n1-1 downto 0);
            y: out std_logic_vector(n2-1 downto 0)
        );
end component;


component adder
        port (
            x,y: in std_logic_vector(31 downto 0);
```

```
        z: out std_logic_vector(31 downto 0)
    );
end component;


component memory is
port (
    address, write_data: in STD_LOGIC_VECTOR (31 downto 0);
    MemWrite, MemRead,ck: in STD_LOGIC;
    read_data: out STD_LOGIC_VECTOR (31 downto 0)
);
end component;


begin

process(ck)
    begin
    case s is
        when running =>
            en <= ck;
        when others =>
            en <= '0';
    end case;

    if ck='1' and ck'event then
        case s is
            when loading =>
                s <= running; -- give 1 cycle to load the instructions into memory
            when running =>
                if instr_address > last_instr_address then
                    s <= done; -- stop moving the pc after it has passed the last instruction
                    en <= '0';
                end if;
            when others =>
                null;
        end case;
    end if;
end process;

-- Wire some stuff
opcode <= instruction(31 downto 26);
rs <= instruction(25 downto 21);
rt <= instruction(20 downto 16);
```

```
rd <= instruction(15 downto 11);
shampt <= instruction(10 downto 6);
funct <= instruction(5 downto 0);
immediate <= instruction(15 downto 0);
jump_address <= instruction(25 downto 0);

Prog_Count: pc port map (en, next_address, instr_address);

IM: instruction_memory port map (instr_address, instruction, last_instr_address);

CONTROL1: control port map (
        opcode => opcode,
        reg_dest => reg_dest,
        jump => jump,
        branch => branch,
        mem_read => mem_read,
        mem_to_reg => mem_to_reg,
        mem_write => mem_write,
        alu_src => alu_src,
        reg_write => reg_write,
        alu_op => alu_op
);

-- This mux is going into Register's Write Register port; chooses between rt and rd
MUX1: mux generic map(5) port map (
        x => rt,
        y => rd,
        s => reg_dest,
        z => write_reg
);

REG: registers port map (
        ck => en,
        reg_write => reg_write,
        read_reg_1 => rs,
        read_reg_2 => rt,
        write_reg => write_reg,
        write_data => write_data,
        read_data_1 => read_data_1,
        read_data_2 => read_data_2
);

ALU_CONTRL: alu_control port map (funct, alu_op, alu_control_fuct);
```

```
---- This mux is going into the ALU's second input; chooses between read_data_2 and the immediate
SGN_EXT: sign_extend port map (immediate, extended_immediate);

MUX2: mux generic map(32) port map (
        x => read_data_2,
        y => extended_immediate,
        s => alu_src,
        z => alu_in_2
);

ALU1: alu port map (read_data_1, alu_in_2, alu_control_fuct, alu_zero, alu_result);

-- This mux is going into the Register's Write Data; chooses between the alu_result and read_data from data memory
MUX3: mux generic map (32) port map (
        x => alu_result,
        y => mem_read_data,
        s => mem_to_reg,
        z => write_data
);

-- The Shift Left 2 for the immediate
SHIFT1: shifter port map (
        x => extended_immediate,
        y => shifted_immediate
);

-- The +4 adder for the pc
ADD1: adder port map (
        x => instr_address,
        y => "00000000000000000000000000000100",
        z => incremented_address
);

-- The mux between the +4 adder and the following adder
branch_and_alu_zero <= branch and alu_zero;
MUX4: mux generic map (32) port map (
        x => incremented_address,
        y => add2_result,
        s => branch_and_alu_zero,
        z => mux4_result
);

-- The adder between the PC and the sign-extended immediate
```

```
    ADD2: adder port map (
        x => incremented_address,
        y => shifted_immediate,
        z => add2_result
    );

    -- The Shift Left 2 for the jump instruction
    SHIFT2: shifter generic map (n1 =>26, n2 => 28) port map (
        x => jump_address,
        y => shifted_jump_address
    );

    -- This mux chooses between the result of mux4 and the jump address
    concatenated_pc_and_jump_address <= incremented_address(31 downto 28) & shifted_jump_address; -- I'm ashamed of myself
    MUX5: mux generic map (32) port map (
        x => mux4_result,
        y => concatenated_pc_and_jump_address,
        s => jump,
        z => next_address
    );

    MEM: memory port map (
        address => alu_result,
        write_data => read_data_2,
        MemWrite => mem_write,
        MemRead => mem_read,
        ck => en,
        read_data => mem_read_data
    );

end beh;
```

6.7 memory.vhd

```
--内存模块（RAM）
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity memory is
    port (
        address, write_data: in STD_LOGIC_VECTOR (31 downto 0);
        MemWrite, MemRead,ck: in STD_LOGIC;
```

```
        read_data: out STD_LOGIC_VECTOR (31 downto 0)
    );
end memory;


architecture behavioral of memory is

type mem_array is array(0 to 31) of STD_LOGIC_VECTOR (31 downto 0);

signal data_mem: mem_array := (
    X"00000000", -- initialize data memory
    X"00000000", -- mem 1
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000", -- mem 10
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000", -- mem 20
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000",
    X"00000000", -- mem 30
    X"00000000");

begin
```

```
read_data <= data_mem(conv_integer(address(6 downto 2))) when MemRead = '1' else X"00000000";

mem_process: process(address, write_data,ck)
begin
	if ck = '0' and ck'event then
		if (MemWrite = '1') then
			data_mem(conv_integer(address(6 downto 2))) <= write_data;
		end if;
	end if;
end process mem_process;

end behavioral;
```

6.8 mux.vhd

```
--多路数据选择器，2 选 1
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity mux is
	generic (n: natural:= 1);
	port (
		x,y: in std_logic_vector(n-1 downto 0);
		s: in std_logic;
		z: out std_logic_vector(n-1 downto 0)
	);
end mux;

architecture beh of mux is
	begin
	z <= x when (s='0') else y;
end beh;
```

6.9 pc.vhd

```
--程序计数器 PC，输出当前指令地址
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity pc is
```

```
    port(
        ck: in std_logic;
        address_to_load: in std_logic_vector(31 downto 0);
        current_address: out std_logic_vector(31 downto 0)
    );
end pc;

architecture beh of pc is

    signal address: std_logic_vector(31 downto 0):= "00000000000000000000000000000000";
    begin
    process(ck)
        begin
        current_address <= address;
        if ck='0' and ck'event then
            address <= address_to_load;
        end if;
    end process;

end beh;
```

6.10 registers.vhd

```
--通用寄存器
-- 128byte，32bit*32 行
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity registers is
    port (
        ck: in std_logic;
        reg_write: in std_logic;
        read_reg_1, read_reg_2, write_reg: in std_logic_vector(4 downto 0);
        write_data: in std_logic_vector(31 downto 0);
        read_data_1, read_data_2: out std_logic_vector(31 downto 0)
    );
end registers;

architecture beh of registers is


    type mem_array is array(0 to 31) of STD_LOGIC_VECTOR (31 downto 0);
    signal reg_mem: mem_array := (
```

```
        "00000000000000000000000000000000", -- $zero
        "00000000000000000000000000000000", -- mem 1
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- test add
        "00000000000000000000000000000000", -- test add
        "00000000000000000000000000000000", -- mem 10
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- mem 20
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000",
        "00000000000000000000000000000000", -- mem 30
        "00000000000000000000000000000000"
);

begin

read_data_1 <= reg_mem(to_integer(unsigned(read_reg_1)));
read_data_2 <= reg_mem(to_integer(unsigned(read_reg_2)));

process(ck)
    begin
    if ck='0' and ck'event and reg_write='1' then
        reg_mem(to_integer(unsigned(write_reg))) <= write_data;
    end if;
```

```
    end process;

end beh;
```

6.11 shifter.vhd

```
--移位器，逻辑左移模块
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity shifter is
    generic (n1: natural:= 32; n2: natural:= 32; k: natural:= 2);
    port (
        x: in std_logic_vector(n1-1 downto 0);
        y: out std_logic_vector(n2-1 downto 0)
    );
end entity;

architecture beh of shifter is
    signal temp: std_logic_vector(n2-1 downto 0);

    begin
    temp <= std_logic_vector(resize(unsigned(x), n2));
    y <= std_logic_vector(shift_left(signed(temp), k));
end beh;
```

6.12 sign_extend.vhd

```
--16->32 扩展组件
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity sign_extend is
    port (
        x: in std_logic_vector(15 downto 0);
        y: out std_logic_vector(31 downto 0)
    );
end sign_extend;

architecture beh of sign_extend is
    begin
```

```
    y <= std_logic_vector(resize(signed(x), y'length));
end beh;
```

七．模型机在 Quartus II 环境下的应用

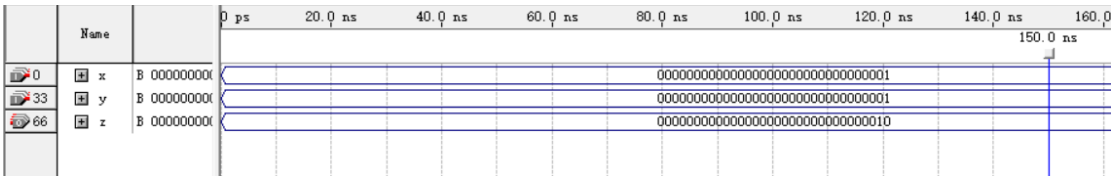
（1）建立工程：工程名 CPU
（2）独立编写各个元器件的 VHDL 代码
（3）各元件独立仿真验证
（4）各元件 RTL 预览
（5）利用元件例化的方式根据数据通路组装成 CPU
（6）组装集成仿真验证
（7）分析实验结果

八．仿真波形

8.1 核心部件仿真

8.1.1 adder 模块

测试用例：
输入：00000000000000000000000000000001
　　00000000000000000000000000000001
理论输出：00000000000000000000000000000010
波形验证：图 8-1-1



（图 8-1-1）

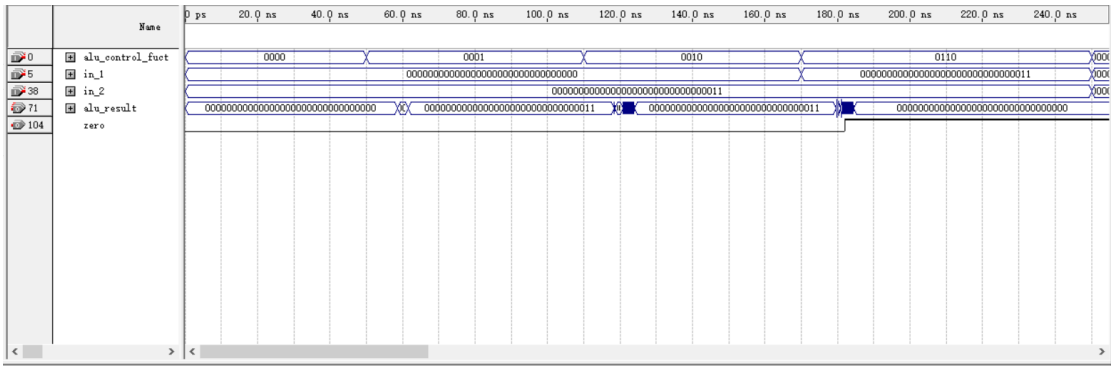
8.1.2 alu 模块:

测试用例
表 8-1-2

测试项	输入 1	输入 2	理论输出
0000(AND)	00000000000000000000000000000000	00000000000000000000000000000011	00000000000000000000000000000000
0001(OR)	00000000000000000000000000000000	00000000000000000000000000000011	00000000000000000000000000000011
0010(+)	00000000000000000000000000000000	00000000000000000000000000000011	00000000000000000000000000000011

0110(-)	00000000000000000000000000000011	00000000000000000000000000000011	00000000000000000000000000000000 ZERO=1

波形验证：图 8-1-2



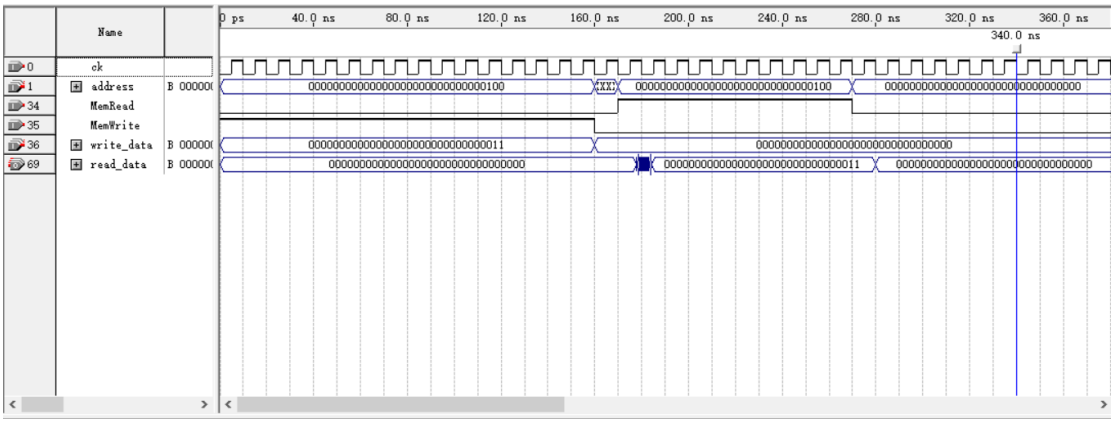
（图 8-1-2）

8.1.3 Memory 模块

在地址 00000000000000000000000000000100 写入数据：00000000000000000000000000000011

读数据使能允许，理论读出数据：00000000000000000000000000000011

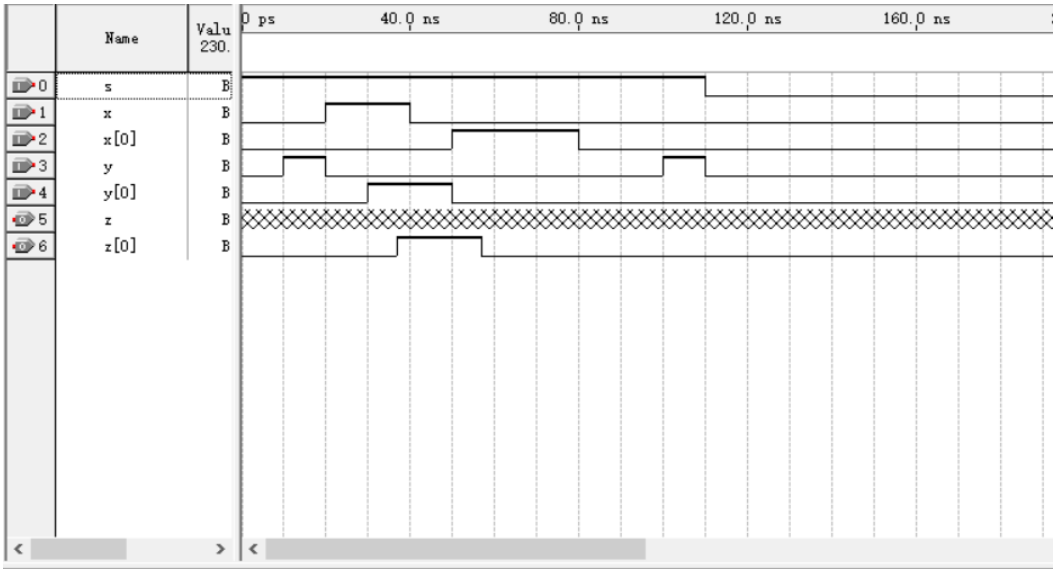
波形验证：图 8-1-3



（图 8-1-3）

8.1.4 Mux 模块

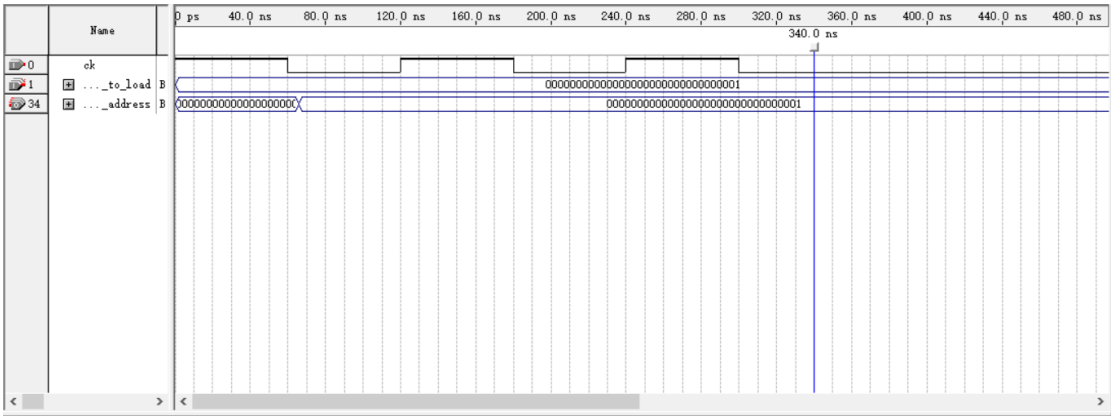
波形验证：图 8-1-4



（图 8-1-4）

8.1.5 PC 模块

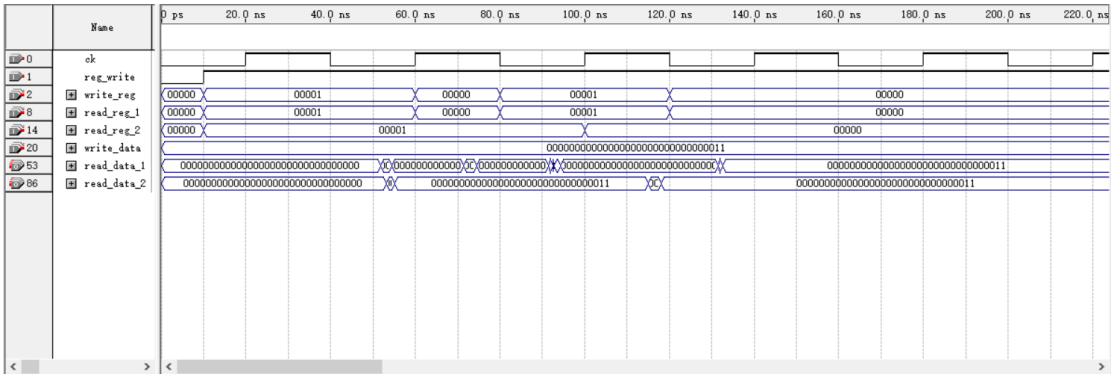
输入：00000000000000000000000000000001
理论输出：00000000000000000000000000000001
波形验证：图 8-1-5



（图 8-1-5）

8.1.6 Registers 模块

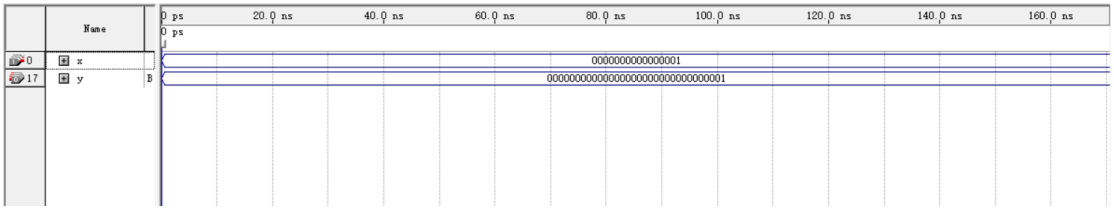
波形验证：图 8-1-6



（图 8-1-6）

8.1.7 sign_extend 模块

输入：0000000000000001
理论输出：00000000000000000000000000000001
波形验证：图 8-1-7



（图 8-1-7）

8.2 总仿真

8.2.1 测试用例

（1）测试汇编代码：

mips1.asm

```
addi $t0 $zero 2#8号寄存器置2
addi $t1 $zero 3#9号寄存器置3
addu $t2 $t1 $t0#8号9号寄存器数据相加，结果（值5）存到10号寄存器
or   $t3 $t0 $t1#8号9号寄存器数据OR操作，结果（值3）保存在11号寄存器
and  $t4 $t0 $t1#8号9号寄存器数据AND操作，结果（值2）保存在12号寄存器
sw   $t2 100($zero)     #把10号寄存器内容（5）装载到内存64H位置
lw   $t5 100($zero)     #把内存64H位置内容装载到13号寄存器
beq  $t0 $t4 LABEL#如果8号寄存器和12号寄存器值相等，跳转到LABEL位置继续执行
LABEL:addi $t1 $zero 0#9号寄存器置0
addi $t0 $zero 9#8号寄存器置9
jal NEXT#无条件跳转到NEXT执行
NEXT:addi $t0 $zero 10#8号寄存器置10
```

（2）指令地址及其寄存器理论值

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x20080002	addi $8, $0, 0x00000002	1: addi $t0 $zero 2#8号寄存器置2
	0x00400004	0x20090003	addi $9, $0, 0x00000003	2: addi $t1 $zero 3#9号寄存器置3
	0x00400008	0x01285021	addu $10, $9, $8	3: addu $t2 $t1 $t0#8号9号寄存器数据相加，结果（值5）存到10号寄存器
	0x0040000c	0x01095825	or $11, $8, $9	4: or $t3 $t0 $t1#8号9号寄存器数据OR操作，结果（值3）保存在11号寄存器
	0x00400010	0x01096024	and $12, $8, $9	5: and $t4 $t0 $t1#8号9号寄存器数据AND操作，结果（值2）保存在12号寄存器
	0x00400014	0xac0a0064	sw $10, 0x00000064($0)	6: sw $t2 100($zero) #把10号寄存器内容（5）装载到内存64H位置
	0x00400018	0x8c0d0064	lw $13, 0x00000064($0)	7: lw $t5 100($zero) #把内存64H位置内容装载到13号寄存器
	0x0040001c	0x110c0000	beq $8, $12, 0x00000000	8: beq $t0 $t4 LABEL#如果8号寄存器和12号寄存器值相等，跳转到LABEL位置继续执行
	0x00400020	0x20090000	addi $9, $0, 0x00000000	9: LABEL:addi $t1 $zero 0#9号寄存器置0
	0x00400024	0x20080009	addi $8, $0, 0x00000009	10: addi $t0 $zero 9#8号寄存器置9
	0x00400028	0x0c10000b	jal 0x0040002c	11: jal NEXT#无条件跳转到NEXT执行
	0x0040002c	0x2008000a	addi $8, $0, 0x0000000a	12: NEXT:addi $t0 $zero 10#8号寄存器置10

（3）二进制指令

00100000000001000000000000000010
00100000000001001000000000000011
00000001001010000101000000100001
00000001000010010101100000100101
00000001000010010110000000100100
10101100000010100000000001100100
10001100000011010000000001100100
00010001000011000000000000000000
00100000000001001000000000000000
00100000000010000000000000001001
00001100000010000000000000001011
00100000000010000000000000001010

8.2.2 仿真波形

图 8-2-2-（1）~（4）

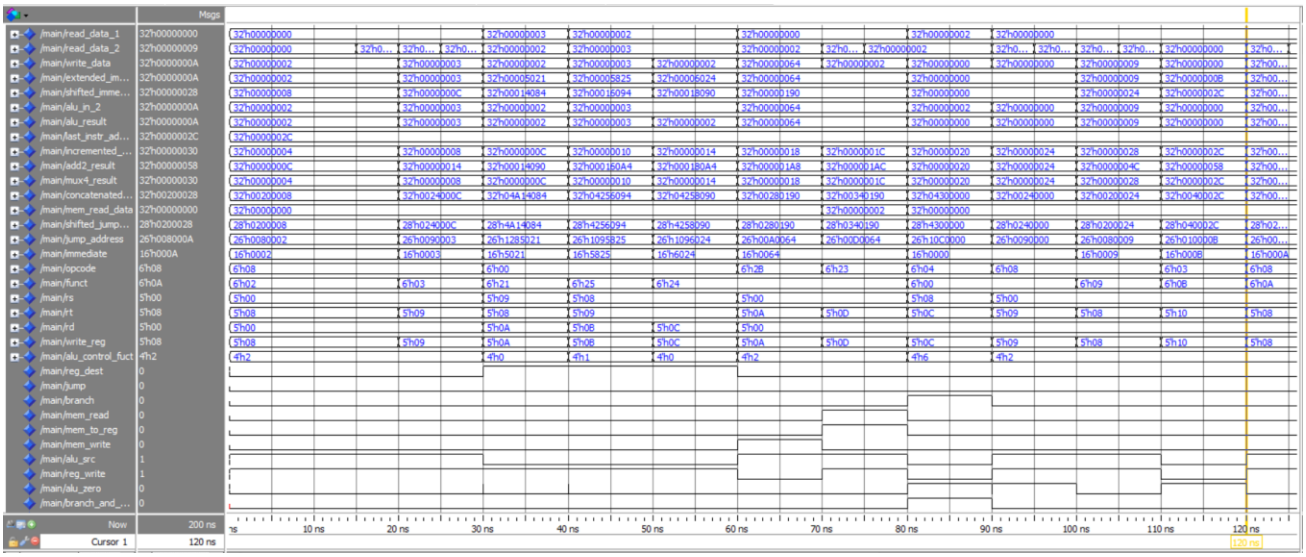


图 8-2-2-（1）

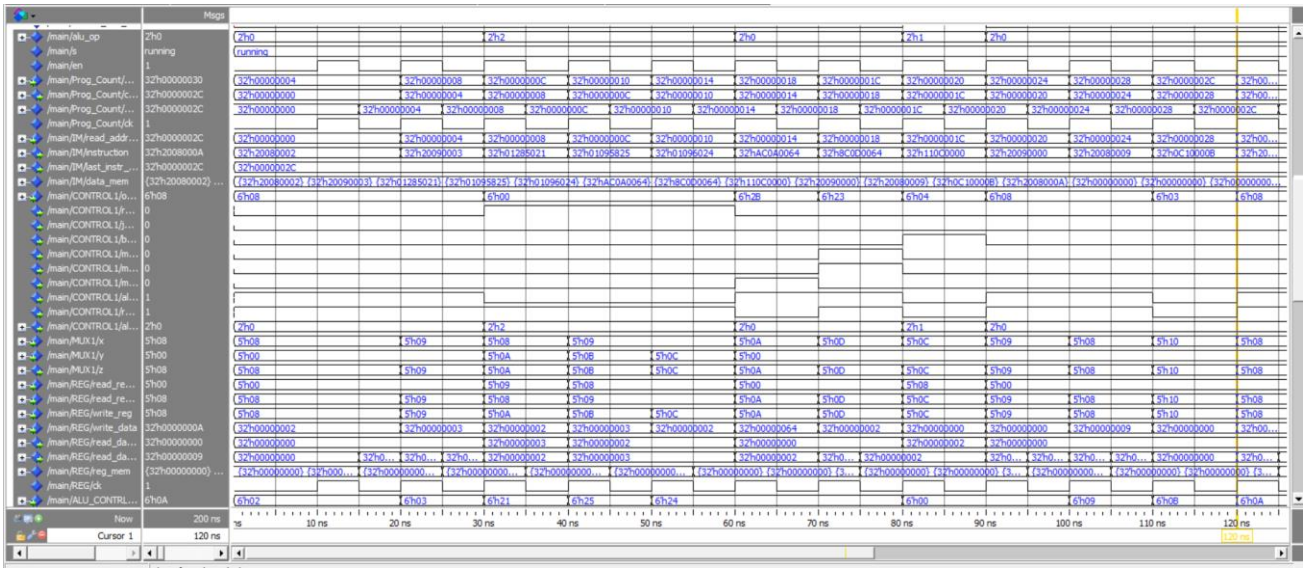


图 8-2-2-（2）

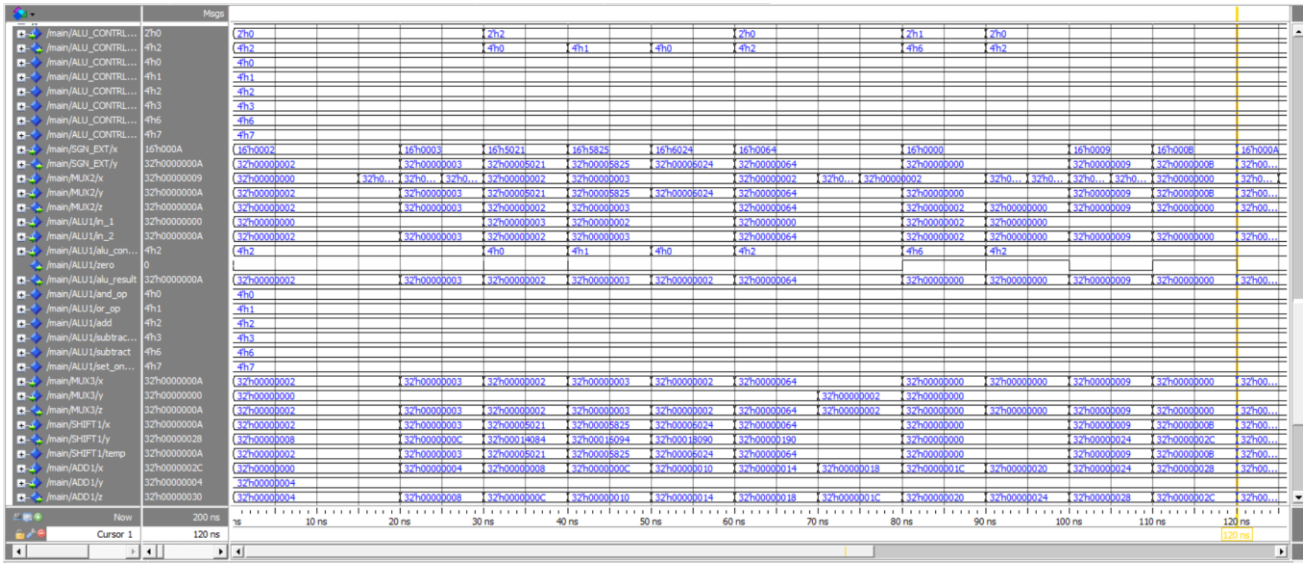


图 8-2-2-（3）

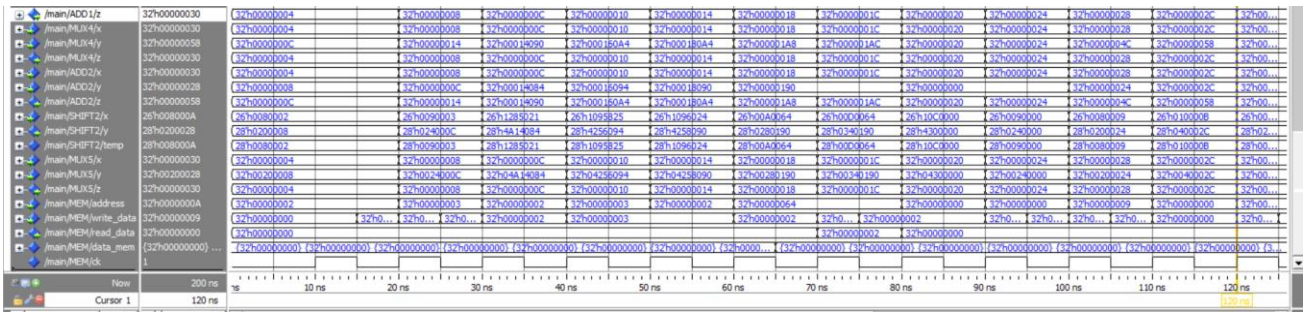


图 8-2-2-（4）

九．总结反思

略

参考文献

[1] 潘松,黄继业. EDA 技术与 VHDL(第 4 版)[M]. 北京: 清华大学出版社, 2013.

[2] 赵宇乾. 基于 FPGA 的 SOC 设计与验证[D]. 河北大学, 2016.

[3] 阿布杜. 带中断系统的五级流水线 CPU 设计[D]. 广东工业大学, 2015.

[4] 袁婷,刘怡俊. 自主设计精简指令集的流水线 CPU[J]. 微电子学与计算机, 2015, (2): 124-128.