第八章 可综合的VerilogHDL设计实例
---简化的RISC CPU设计简介---

前言：

在前面七章里我们已经学习了VerilogHDL的基本语法、简单组合逻辑和简单时序逻辑模块的编写、Top-Down设计方法、还学习了可综合风格的有限状态机的设计，其中EEPROM读写器的设计实质上是一个较复杂的嵌套的有限状态机的设计，它是根据我们完成的实际工程项目设计为教学目的改写而来的,可以说已是真实的设计。

在这一章里，我们将通过一个经过简化的用于教学目的的 RISC_CPU 的设计过程,来说明这种新设计方法的潜力。这个模型实质上是第四章的RISC_CPU模型的改进。第四章中的RISC_CPU模型是一个仿真模型，它关心的只是总体设计的合理性，它的模块中有许多是不可综合的，只可以进行仿真。而本章中构成RISC_CPU的每一个模块不仅是可仿真的也都是可综合的，因为他们符合可综合风格的要求。为了能在这个虚拟的CPU上运行较为复杂的程序并进行仿真，因而把寻址空间扩大到8K（即15位地址线）。下面让我们一步一步地来设计这样一个CPU，并进行仿真和综合，从中我们可以体会到这种设计方法的魅力。本章中的VerilogHDL程序都是我们自己为教学目的而编写的，全部程序在CADENCE公司的LWB (Logic Work Bench)环境下和 Mentor 公司的ModelSim 环境下用Verilog语言进行了仿真,通过了运行测试，并分别用Synergy和Synplify综合器针对不同的FPGA进行了综合。分别用Xilinx和Altera公司的的布局布线工具在Xilinx3098上和Altera Flex10K10实现了布线。 顺利通过综合前仿真、门级结构仿真以及布线后的门级仿真。这个 CPU 模型只是一个教学模型，设计也不一定合理，只是从原理上说明了一个简单的RISC _CPU的构成。我们在这里介绍它的目的是想说明：Verilog HDL 仿真和综合工具的潜力和本文介绍的设计方法对软硬件联合设计是有重要意义的。我们也希望这一章能引起对 CPU 原理和复杂数字逻辑系统设计有兴趣的同学的注意,加入我们的设计队伍。由于我们的经验与学识有限，不足之处敬请读者指正。

8.1.什么是CPU？

CPU 即中央处理单元的英文缩写，它是计算机的核心部件。计算机进行信息处理可分为两个步骤：

1) 将数据和程序（即指令序列）输入到计算机的存储器中。
2) 从第一条指令的地址起开始执行该程序，得到所需结果，结束运行。CPU的作用是协调并控制计算机的各个部件执行程序的指令序列，使其有条不紊地进行。因此它必须具有以下基本功能：
   a)取指令：当程序已在存储器中时，首先根据程序入口地址取出一条程序，为此要发出指令地址及控制信号。
   b)分析指令：即指令译码。是对当前取得的指令进行分析，指出它要求什么操作，并产生相应的操作控制命令。
   c)执行指令：根据分析指令时产生的“操作命令”形成相应的操作控制信号序列，通过运算器，存储器及输入/输出设备的执行，实现每条指令的功能，其中包括对运算结果的处理以及下条指令地址的形成。

将其功能进一步细化，可概括如下：

1) 能对指令进行译码并执行规定的动作；
2) 可以进行算术和逻辑运算；
3) 能与存储器，外设交换数据；
4) 提供整个系统所需要的控制；

尽管各种CPU的性能指标和结构细节各不相同，但它们所能完成的基本功能相同。由功能分析，可知任何一种CPU内部结构至少应包含下面这些部件：

1)算术逻辑运算部件（ALU），
2)累加器，
3)程序计数器，
4)指令寄存器，译码器，
5)时序和控制部件。

RISC 即精简指令集计算机（Reduced Instruction Set Computer）的缩写。它是一种八十年代才出现的CPU，与一般的CPU 相比不仅只是简化了指令系统，而且是通过简化指令系统使计算机的结构更加简单合理，从而提高了运算速度。从实现的途径看，RISC_CPU与一般的CPU的不同处在于：**它的时序控制信号形成部件是用硬布线逻辑实现的而不是采用微程序控制的方式。**所谓硬布线逻辑也就是用触发器和逻辑门直接连线所构成的状态机和组合逻辑，故产生控制序列的速度比用微程序控制方式快得多，因为这样做省去了读取微指令的时间。RISC_CPU也包括上述这些部件，下面就详细介绍一个简化的用于教学目的的RISC_CPU的可综合VerilogHDL模型的设计和仿真过程。

8.2. RISC CPU结构

RISC_CPU是一个复杂的数字逻辑电路，但是它的基本部件的逻辑并不复杂。从第四章我们知道可把它分成八个基本部件：

1)时钟发生器
2)指令寄存器
3)累加器
4)RISC CPU算术逻辑运算单元
5)数据控制器
6)状态控制器
7)程序计数器
8)地址多路器

各部件的相互连接关系见图8.2。其中时钟发生器利用外来时钟信号进行分频生成一系列时钟信号，送往其他部件用作时钟信号。各部件之间的相互操作关系则由状态控制器来控制。各部件的具体结构和逻辑关系在下面的小节里逐一进行介绍。

8.2.1时钟发生器

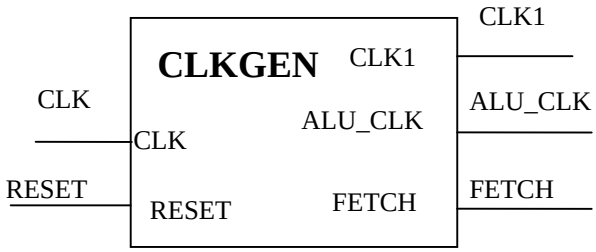


图1. 时钟发生器

时钟发生器 clkgen 利用外来时钟信号clk 来生成一系列时钟信号clk1、fetch、alu_clk 送往CPU的其他部件。其中fetch是外来时钟 clk 的八分频信号。利用fetch的上升沿来触发CPU控制器开始执行一条指令，同时fetch信号还将控制地址多路器输出指令地址和数据地址。clk1信号用作指令寄存器、累加器、状态控制器的时钟信号。alu_clk 则用于触发算术逻辑运算单元。
时钟发生器clkgen的波形见下图8.2.2所示：

其VerilogHDL 程序见下面的模块：

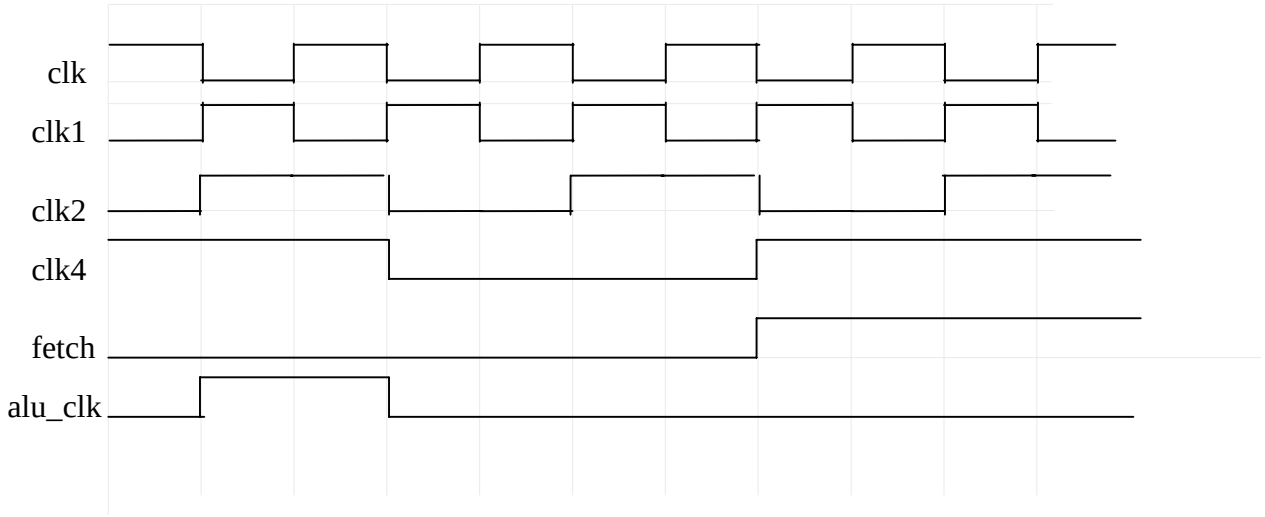


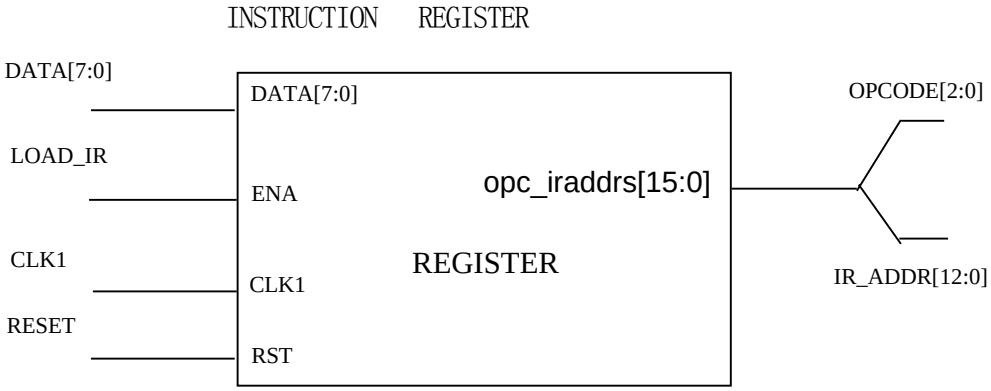
图8.2.2　时钟发生器clkgen的波形

```
module clk_gen (clk,reset,clk1,clk2,clk4,fetch,alu_clk);
input clk,reset;
output clk1,clk2,clk4,fetch,alu_clk;
wire clk,reset;
reg clk2,clk4,fetch,alu_clk;
reg[7:0] state;
parameter    S1 = 8'b00000001,
             S2 = 8'b00000010,
             S3 = 8'b00000100,
             S4 = 8'b00001000,
             S5 = 8'b00010000,
             S6 = 8'b00100000,
             S7 = 8'b01000000,
             S8 = 8'b10000000,
           idle = 8'b00000000;

assign clk1 = ~clk;

always @(negedge clk)
   if(reset)
           begin
                clk2 <= 0;
                clk4 <= 1;
                fetch  <= 0;
                alu_clk <= 0;
                state <= idle;
           end
   else
           begin
              case(state)
                 S1:
                      begin
                            clk2 <= ~clk2;
```

```
                    alu_clk <= ~alu_clk;
                      state <= S2;
                end
            S2:
                begin
                    clk2 <= ~clk2;
                    clk4 <= ~clk4;
                alu_clk <= ~alu_clk;
                  state <= S3;
                end
            S3:
                begin
                    clk2 <= ~clk2;
                   state <= S4;
                end
            S4:
                begin
                    clk2  <= ~clk2;
                    clk4  <= ~clk4;
                    fetch <= ~fetch;
                    state <= S5;
                end
            S5:
                 begin
                    clk2  <= ~clk2;
                    state <= S6;
                 end
            S6:
                 begin
                     clk2  <= ~clk2;
                      clk4 <= ~clk4;
                     state <= S7;
                  end
            S7:
                  begin
                       clk2  <= ~clk2;
                       state <=  S8;
                  end
            S8:
                  begin
                       clk2 <= ~clk2;
                      clk4  <= ~clk4;
                      fetch <= ~fetch;
                      state <= S1;
                  end
           idle:       state <= S1;
           default:    state <= idle;
        endcase
    end
endmodule
//-------------------------------------------------------------------------------
```

由于在时钟发生器的设计中采用了同步状态机的设计方法，不但使clk_gen模块的源程序可以被各种综合器综合，也使得由其生成的clk1、clk2、clk4、fetch、alu_clk 在跳变时间同步性能上有明显的提高，为整个系统的性能提高打下了良好的基础。

8.2.2 指令寄存器



顾名思义，指令寄存器用于寄存指令。

指令寄存器的触发时钟是clk1，在clk1的正沿触发下，寄存器将数据总线送来的指令存入高8位或低8位寄存器中。但并不是每个clk1的上升沿都寄存数据总线的数据，因为数据总线上有时传输指令，有时传输数据。什么时候寄存，什么时候不寄存由CPU状态控制器的load_ir信号控制。load_ir信号通过ena 口输入到指令寄存器。复位后，指令寄存器被清为零。

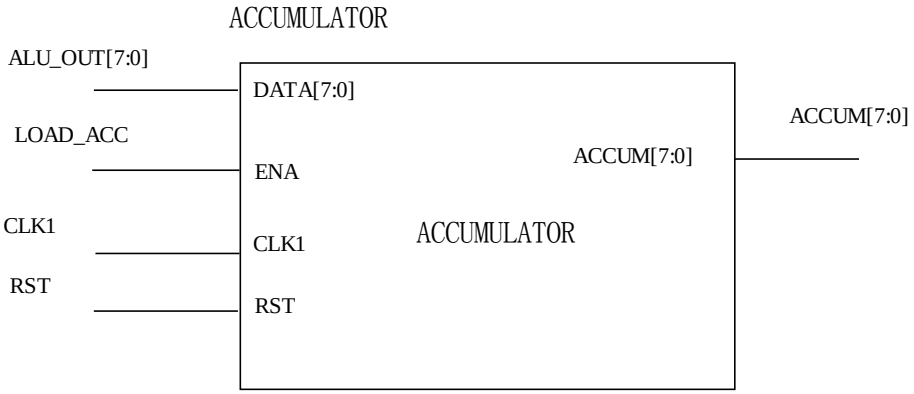
每条指令为2个字节，即16位。高3位是操作码，低13位是地址。（CPU的地址总线为13位，寻址空间为8K字节。）本设计的数据总线为8位，所以每条指令需取两次。先取高8位，后取低8位。而当前取的是高8位还是低8位，由变量state记录。state为零表示取的高8位，存入高8位寄存器，同时将变量state置为1。下次再寄存时，由于state为1，可知取的是低8位，存入低8位寄存器中。

其VerilogHDL 程序见下面的模块：

```
//-------------------------------------------------------------
        module register(opc_iraddr,data,ena,clk1,rst);
        output [15:0] opc_iraddr;
        input [7:0] data;
        input ena, clk1, rst;
        reg [15:0] opc_iraddr;
        reg  state;

        always @(posedge clk1)
        begin
           if(rst)
             begin
                opc_iraddr<=16'b0000_0000_0000_0000;
                state<=1'b0;
             end
            else
             begin
                if(ena)           //如果加载指令寄存器信号load_ir到来，
                 begin            //分两个时钟每次8位加载指令寄存器
                   casex(state)   //先高字节，后低字节
```

```
                    1'b0:  begin
                           opc_iraddr[15:8]<=data;
                           state<=1;
                           end
                    1'b1:  begin
                           opc_iraddr[7:0]<=data;
                           state<=0;
                           end
                    default:  begin
                                opc_iraddr[15:0]<=16'bxxxxxxxxxxxxxxxx;
                                state<=1'bx;
                              end
              endcase
            end
          else
            state<=1'b0;
          end
        end
        endmodule
//-------------------------------------------------------
```

8.2.3.累加器



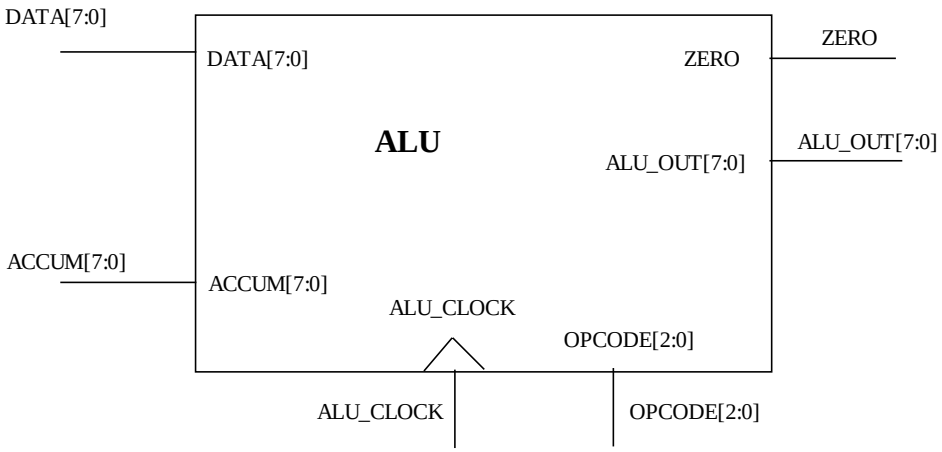
累加器用于存放当前的结果，它也是双目运算其中一个数据来源。复位后，累加器的值是零。当累加器通过ena口收到来自CPU状态控制器load_acc信号时，在clk1时钟正跳沿时就收到来自于数据总线的数据。

其VerilogHDL 程序见下面的模块:

```
//-------------------------------------------------------------
        module accum( accum, data, ena, clk1, rst);
          output[7:0]accum;
          input[7:0]data;
          input ena,clk1,rst;
          reg[7:0]accum;

          always@(posedge clk1)
            begin
              if(rst)
                accum<=8'b0000_0000;        //Reset
```

```
        else
          if(ena)        //当CPU状态控制器发出load_acc信号
            accum<=data;              //Accumulate
      end

endmodule
```

8.2.4.算术运算器



算术逻辑运算单元 根据输入的8种不同操作码分别实现相应的加、与、异或、跳转等8种基本操作运算。利用这几种基本运算可以实现很多种其它运算以及逻辑判断等操作。

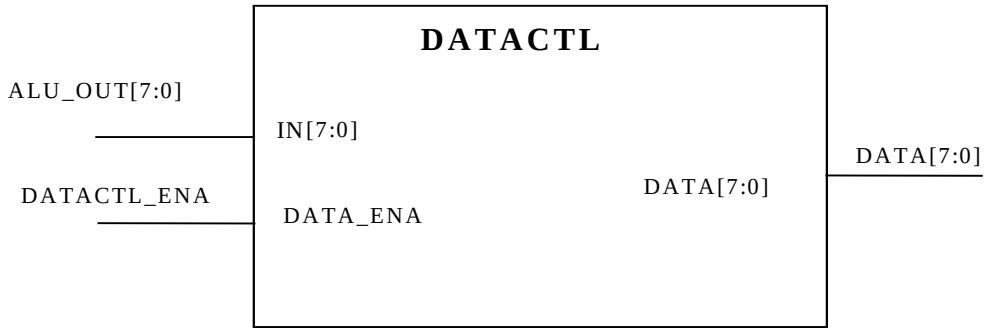
其VerilogHDL 程序见下面的模块：

```
//-------------------------------------------------------------------------
      module alu (alu_out, zero, data, accum, alu_clk, opcode);
      output [7:0]alu_out;
      output zero;
      input [7:0] data, accum;
      input [2:0] opcode;
      input alu_clk;
      reg [7:0] alu_out;

        parameter      HLT  =3'b000,
                       SKZ  =3'b001,
                       ADD  =3'b010,
                       ANDD =3'b011,
                       XORR =3'b100,
                       LDA  =3'b101,
                       STO  =3'b110,
                       JMP  =3'b111;

        assign zero = !accum;
        always @(posedgealu_clk)
          begin  //操作码来自指令寄存器的输出opc_iaddr<15..0>的低3位
            casex (opcode)
              HLT: alu_out<=accum;
              SKZ: alu_out<=accum;
              ADD: alu_out<=data+accum;
              ANDD: alu_out<=data&accum;
```

```
                XORR: alu_out<=data^accum;
                LDA: alu_out<=data;
                STO: alu_out<=accum;
                JMP: alu_out<=accum;
                default: alu_out<=8'bxxxx_xxxx;
              endcase
            end
        endmodule
//-------------------------------------------------------------------------
```

8.2.5.数据控制器

数据控制器的作用是控制累加器数据输出，由于数据总线是各种操作时传送数据的公共通道，

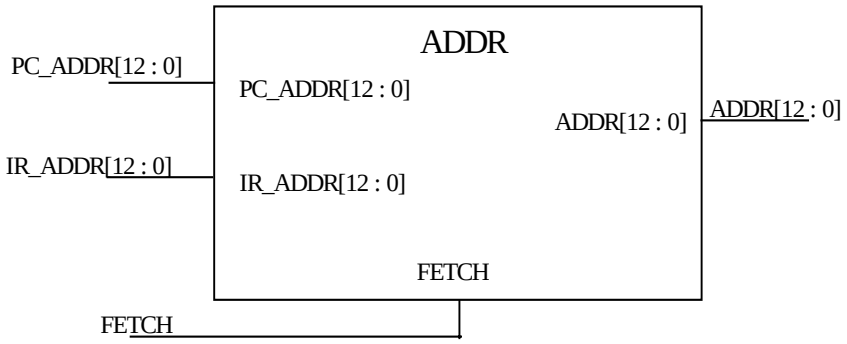


不同的情况下传送不同的内容。有时要传输指令，有时要传送RAM区或接口的数据。累加器的数据只有在需要往RAM区或端口写时才允许输出，否则应呈现高阻态，以允许其它部件使用数据总线。 所以任何部件往总线上输出数据时，都需要一控制信号。而此控制信号的启、停，则由CPU状态控制器输出的各信号控制决定。数据控制器何时输出累加器的数据则由状态控制器输出的控制信号datactl_ena决定。

其VerilogHDL 程序见下面的模块:

```
//-------------------------------------------------------------------
        module datactl (data,in,data_ena);
             output [7:0]data;
             input [7:0]in;
             input data_ena;

             assign  data = (data_ena)? In : 8'bzzzz_zzzz;

        endmodule
//-------------------------------------------------------------------
```

8.2.6.地址多路器



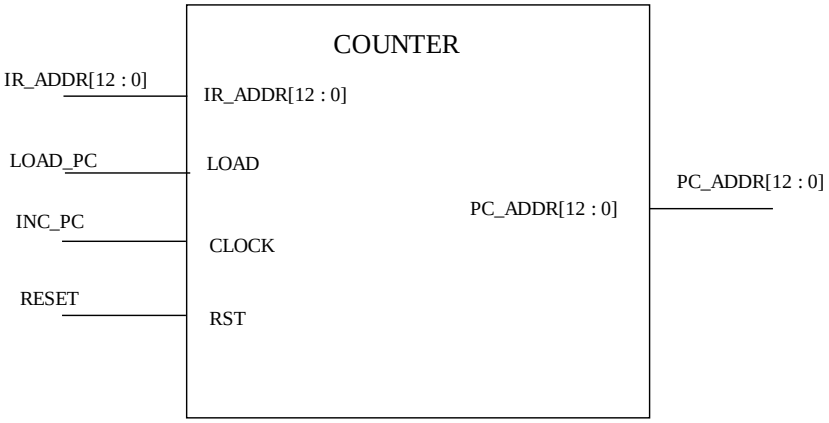
地址多路器用于选择输出的地址是PC（程序计数）地址还是数据/端口地址。每个指令周期的前4个时钟周期用于从ROM中读取指令，输出的应是PC地址。后4个时钟周期用于对RAM或端口的读写，该地址由指令中给出。地址的选择输出信号由时钟信号的8分频信号fetch提供。

其VerilogHDL 程序见下面的模块：

```
//-------------------------------------------------------------------------
        module  adr(addr,fetch,ir_addr,pc_addr);
        output [12:0] addr;
        input [12:0] ir_addr, pc_addr;
        input  fetch;

        assign  addr = fetch?  pc_addr : ir_addr;

        endmodule
//-------------------------------------------------------------------------
```

8.2.7.程序计数器

程序计数器用于提供指令地址。以便读取指令，指令按地址顺序存放在存储器中。有两种途径可形成指令地址：其一是顺序执行的情况，其二是遇到要改变顺序执行程序的情况，例如执行JMP指令后，需要形成新的指令地址。下面就来详细说明PC地址是如何建立的。



复位后，指令指针为零，即每次CPU重新启动将从ROM的零地址开始读取指令并执行。每条指令执行完需2个时钟，这时pc_addr已被增2，指向下一条指令。（因为每条指令占两个字节。）如果正执行的指令是跳转语句，这时CPU状态控制器将会输出load_pc信号，通过load口进入程序计数器。程序计数器（pc_addr）将装入目标地址（ir_addr），而不是增2。

其VerilogHDL 程序见下面的模块：

```
//-------------------------------------------------------------------------------
        module counter ( pc_addr, ir_addr, load, clock, rst);
          output [12:0] pc_addr;
          input [12:0] ir_addr;
          input load, clock, rst;
          reg [12:0] pc_addr;

          always @( posedge clock or posedge rst )
            begin
              if(rst)
                pc_addr<=13'b0_0000_0000_0000;
              else
                if(load)
                pc_addr<=ir_addr;
                else
                  pc_addr <= pc_addr + 1;
            end
        endmodule
//-------------------------------------------------------------------------------
```

8.2.8.状态控制器

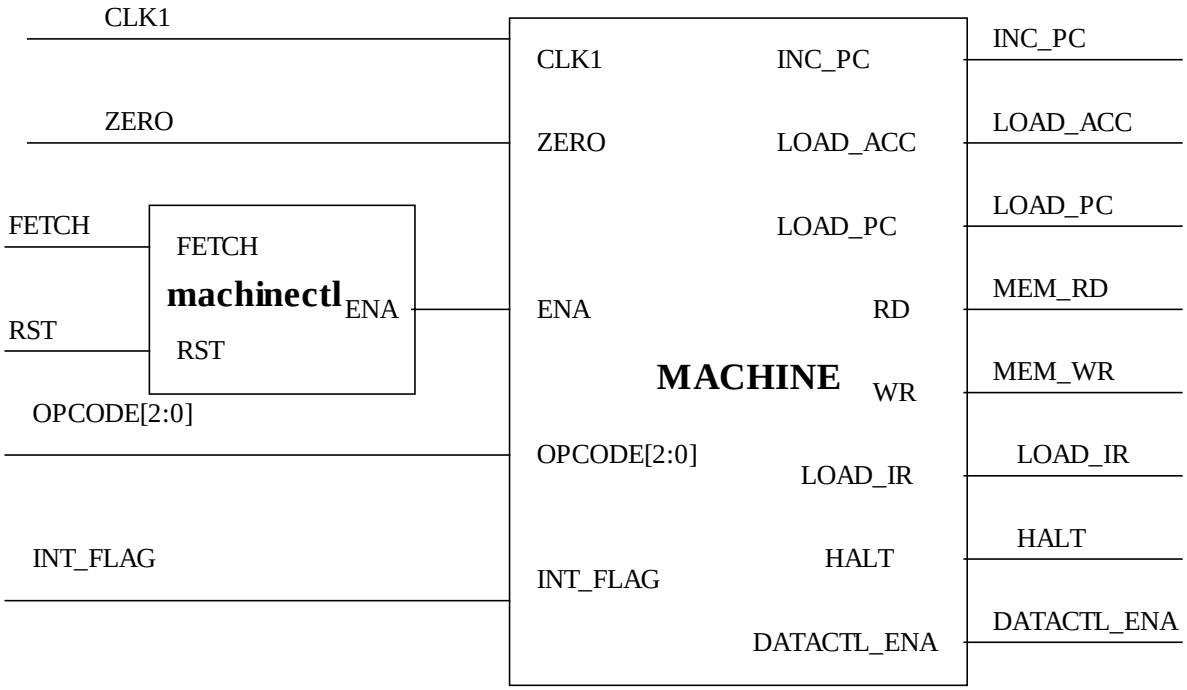


图8.2.8状态控制器

状态控制器由两部分组成：

1.状态机(上图中的MACHINE部分)
2.状态控制器(上图中的MACHINECTL部分)

状态机控制器接受复位信号RST，当RST有效时通过信号ena使其为0，输入到状态机中停止状态机的工作。

状态控制器的VerilogHDL程序见下面模块：

```
//-------------------------------------------------------------------------------
        module machinectl( ena, fetch, rst);
        output  ena;
        input  fetch, rst;
        reg ena;

          always @(posedge fetch or posedge rst)
            begin
              if(rst)
                ena<=0;
              else
                ena<=1;
            end

        endmodule
//-------------------------------------------------------------------------------
```

状态机是CPU的控制核心，用于产生一系列的控制信号，启动或停止某些部件。CPU何时进行读指令读写I/O端口，RAM区等操作，都是由状态机来控制的。状态机的当前状态，由变量state记录，state的值就是当前这个指令周期中已经过的时钟数（从零计起）。

指令周期是由8个时钟周期组成，每个时钟周期都要完成固定的操作。

1) 第0个时钟，因为CPU状态控制器的输出：rd和load_ir为高电平，其余均为低电平。指令寄存器寄存由ROM送来的高8位指令代码。

2) 第1个时钟，与上一时钟相比只是inc_pc从0变为1故PC增1，ROM送来低8位指令代码，指令寄存器寄存该8位代码。

3) 第2个时钟，空操作。

4) 第3个时钟，PC增1，指向下一条指令。若操作符为HLT，则输出信号HLT为高。如果操作符不为HLT，除了PC增一外（指向下一条指令），其它各控制线输出为零。

5) 第4个时钟，若操作符为AND、ADD、XOR或LDA，读相应地址的数据；若为JMP，将目的地址送给程序计数器；若为STO，输出累加器数据。

6) 第5个时钟，若操作符为ANDD、ADD或XORR，算术运算器就进行相应的运算；若为LDA，就把数据通过算术运算器送给累加器；若为SKZ，先判断累加器的值是否为0，如果为0，PC就增1，否则保持原值；若为JMP，锁存目的地址；若为STO，将数据写入地址处。

7) 第6个时钟，空操作。

8) 第7个时钟，若操作符为SKZ且累加器值为0，则PC值再增1，跳过一条指令，否则PC无变化。

状态机的VerilogHDL 程序见下面模块：

```
//-------------------------------------------------------------------------
module  machine( inc_pc, load_acc, load_pc, rd,wr, load_ir,
                         datactl_ena, halt, clk1, zero, ena, opcode );

        output inc_pc, load_acc, load_pc, rd, wr, load_ir;
        output datactl_ena, halt;
        input clk1, zero, ena;
        input [2:0] opcode;
        reg inc_pc, load_acc, load_pc, rd, wr, load_ir;
        reg datactl_ena, halt;
        reg [2:0] state;

        parameter   HLT  = 3 'b000,
                    SKZ  = 3 'b001,
                    ADD  = 3 'b010,
                    ANDD = 3 'b011,
                    XORR = 3 'b100,
                    LDA  = 3 'b101,
                    STO  = 3 'b110,
                    JMP  = 3 'b111;

          always @( negedge clk1 )
            begin
              if ( !ena )              //接收到复位信号RST，进行复位操作
                begin
```

```
                state<=3'b000;
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
              end
            else
              ctl_cycle;
          end
//-----------------begin of task ctl_cycle---------
        task ctl_cycle;
        begin
        casex(state)
    3'b000:          //load high 8bits in struction
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0001;
            {wr,load_ir,datactl_ena,halt}<=4'b0100;
            state<=3'b001;
          end
    3'b001:         //pc increased by one then load low 8bits instruction
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b1001;
            {wr,load_ir,datactl_ena,halt}<=4'b0100;
            state<=3'b010;
          end
    3'b010:         //idle
          begin
            {inc_pc,load_acc,load_pc,rd}<=4'b0000;
            {wr,load_ir,datactl_ena,halt}<=4'b0000;
            state<=3'b011;
          end

    3'b011:         //next instruction address setup 分析指令从这里开始
          begin
            if(opcode==HLT)  //指令为暂停HLT
              begin
              {inc_pc,load_acc,load_pc,rd}<=4'b1000;
              {wr,load_ir,datactl_ena,halt}<=4'b0001;
              end
              else
               begin
                 {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                 {wr,load_ir,datactl_ena,halt}<=4'b0000;
               end
            state<=3'b100;
          end
    3'b100:               //fetch oprand
          begin
            if(opcode==JMP)
              begin
                 {inc_pc,load_acc,load_pc,rd}<=4'b0010;
                 {wr,load_ir,datactl_ena,halt}<=4'b0000;
              end
              else
```

```
                if( opcode==ADD || opcode==ANDD ||
                    opcode==XORR || opcode==LDA)
                  begin
                    {inc_pc,load_acc,load_pc,rd}<=4'b0001;
                    {wr,load_ir,datactl_ena,halt}<=4'b0000;
                  end
              else
                if(opcode==STO)
                 begin
                   {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                   {wr,load_ir,datactl_ena,halt}<=4'b0010;
                 end
              else
                  begin
                   {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                   {wr,load_ir,datactl_ena,halt}<=4'b0000;
                  end
            state<=3'b101;
          end
3'b101:          //operation
          begin
            if ( opcode==ADD||opcode==ANDD||
                opcode==XORR||opcode==LDA )
              begin          //过一个时钟后与累加器的内容进行运算
                {inc_pc,load_acc,load_pc,rd}<=4'b0101;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
              end
              else
                if( opcode==SKZ && zero==1)
                  begin
                    {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                    {wr,load_ir,datactl_ena,halt}<=4'b0000;
                  end
                 else
                   if(opcode==JMP)
                    begin
                      {inc_pc,load_acc,load_pc,rd}<=4'b1010;
                      {wr,load_ir,datactl_ena,halt}<=4'b0000;
                    end
                    else
                      if(opcode==STO)
                       begin
                       //过一个时钟后把wr变1就可写到RAM中
                       {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                       {wr,load_ir,datactl_ena,halt}<=4'b1010;
                       end
                       else
                        begin
                          {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                          {wr,load_ir,datactl_ena,halt}<=4'b0000;
                        end
              state<=3'b110;
```

```
            end
          3'b110:          //idle
            begin
              if ( opcode==STO )
                begin
                  {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                  {wr,load_ir,datactl_ena,halt}<=4'b0010;
                end
              else
                if ( opcode==ADD||opcode==ANDD||
                     opcode==XORR||opcode==LDA)
                 begin
                   {inc_pc,load_acc,load_pc,rd}<=4'b0001;
                   {wr,load_ir,datactl_ena,halt}<=4'b0000;
                 end
                else
                 begin
                  {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                  {wr,load_ir,datactl_ena,halt}<=4'b0000;
                 end
             state<=3'b111;
            end

          3'b111:          //
            begin
              if( opcode==SKZ && zero==1 )
               begin
                {inc_pc,load_acc,load_pc,rd}<=4'b1000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
               end
              else
               begin
                {inc_pc,load_acc,load_pc,rd}<=4'b0000;
                {wr,load_ir,datactl_ena,halt}<=4'b0000;
               end
             state<=3'b000;
            end
          default:
            begin
              {inc_pc,load_acc,load_pc,rd}<=4'b0000;
              {wr,load_ir,datactl_ena,halt}<=4'b0000;
              state<=3'b000;
            end
          endcase
          end
        endtask
//-----------------end of task ctl_cycle---------

endmodule
//-------------------------------------------------------------------------
```

状态机和状态机控制器组成了状态控制器。它们之间的连接关系很简单。见本小节的图8.2.8。

8.2.9.外围模块

为了对RISC_CPU进行测试，需要有存储测试程序的ROM和装载数据的RAM、地址译码器。下面来简单介绍一下：

1.地址译码器

```
module addr_decode( addr, rom_sel, ram_sel);
   output rom_sel, ram_sel;
   input [12:0] addr;
   reg rom_sel, ram_sel;

   always @( addr )
   begin
      casex(addr)
         13'b1_1xxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b01;
         13'b0_xxxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b10;
         13'b1_0xxx_xxxx_xxxx:{rom_sel,ram_sel}<=2'b10;
         default:{rom_sel,ram_sel}<=2'b00;
      endcase
   end
endmodule
```

地址译码器用于产生选通信号，选通ROM或RAM。

FFFFH---1800H RAM

1800H---0000H ROM

2.RAM和ROM

```
module ram( data, addr, ena, read, write );
   inout [7:0] data;
   input [9:0] addr;
   input ena;
   input read, write;
   reg [7:0] ram [10'h3ff:0];

   assign data = ( read && ena )?  ram[addr] : 8'hzz;

   always @(posedge write)
      begin
         ram[addr]<=data;
      end
endmodule


module rom( data, addr, read, ena );
   output [7:0] data;
   input [12:0] addr;
   input read, ena;
   reg [7:0] memory [13'h1fff:0];
   wire [7:0] data;
```

```
    assign data= ( read && ena )? memory[addr] : 8'bzzzzzzzz;

endmodule
```

ROM用于装载测试程序，可读不可写。RAM用于存放数据，可读可写。

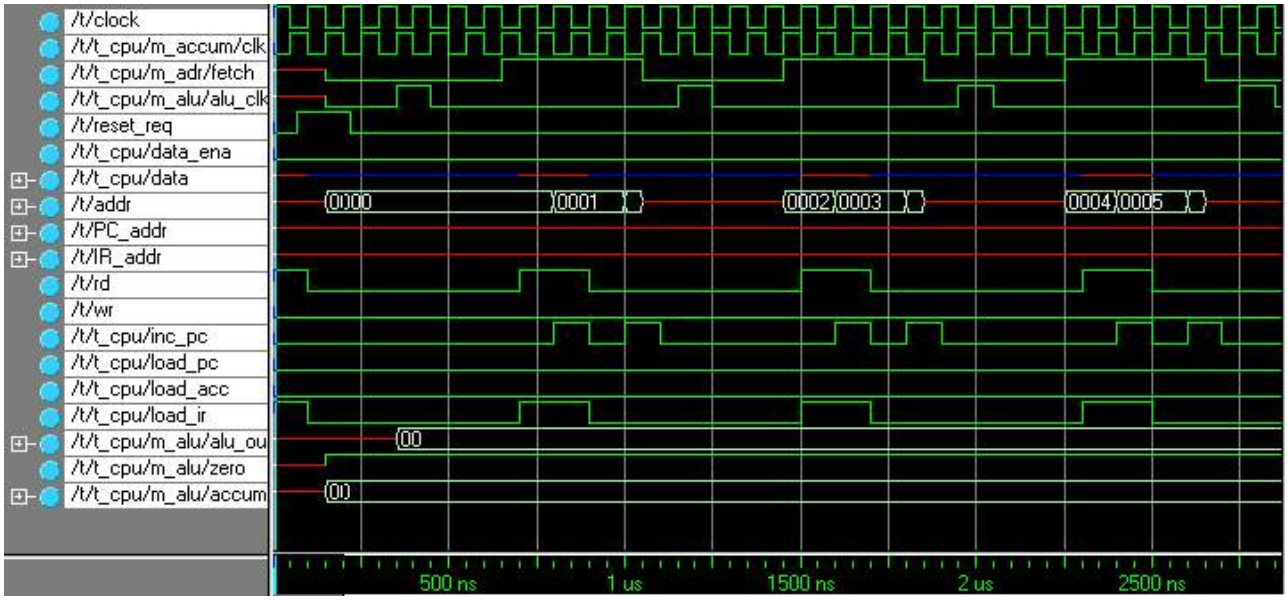
8.3. RISC_CPU 操作和时序

一个微机系统为了完成自身的功能，需要CPU执行许多操作。以下是RISC_CPU的主要操作：
1.系统的复位和启动操作
2.总线读操作
3.总线写操作
下面详细介绍一下每个操作：

8.3.1.系统的复位和启动操作

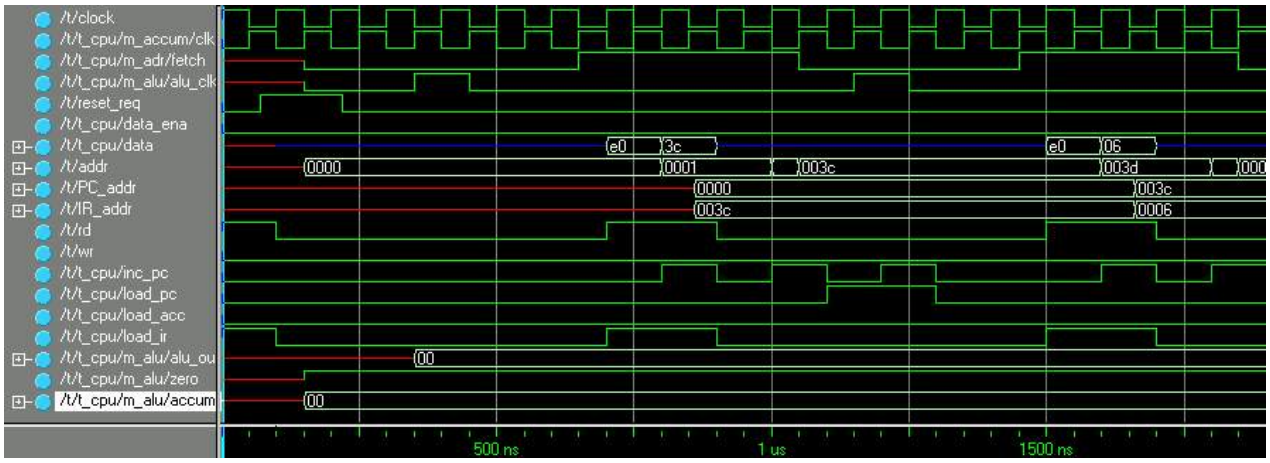
RISC_CPU的复位和启动操作是通过rst引脚的信号触发执行的。当rst信号一进入高电平，RISC_CPU就会结束现行操作，并且只要rst停留在高电平状态，CPU就维持在复位状态。在复位状态，CPU各内部寄存器都被设为初值，全部为零。数据总线为高阻态，地址总线为0000H，所有控制信号均为无效状态。rst回到低电平后，接着到来的第一个fetch上升沿将启动RISC_CPU开始工作，从ROM的000处开始读取指令并执行相应操作。波形图见8.3.1。虚线标志处为RISC_CPU启动工作的时刻。

8.3.2.总线读操作



RISC_CPU的复位和启动操作波形

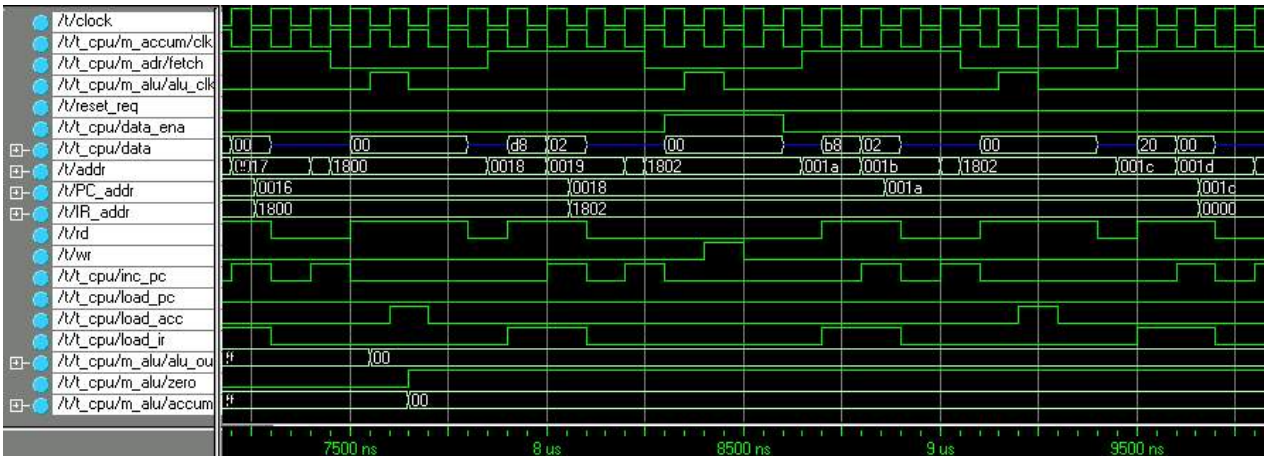
每个指令周期的前0--3个时钟周期用于读指令，在状态控制器一节中已详细讲述，这里就不再重复。第3.5个周期处，存储器或端口地址就输出到地址总线上，第4--6个时钟周期，读信号rd有效，数据送到数据总线上，以备累加器锁存，或参与算术、逻辑运算。第7个时钟周期，读信号无效，第7.5个周期，地址总线输出PC地址，为下一个指令做好准备。



CPU从存储器或端口读取数据的时序

8.3.3写总线操作

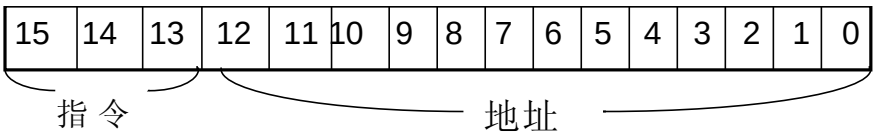
每个指令周期的第3.5个时钟周期处，写的地址就建立了，第4个时钟周期输出数据，第5个时钟周期输出写信号。至第6个时钟结束，数据无效，第7.5时钟地址输出为PC地址，为下一个指令周期做好准备。



CPU对存储器或端口写数据的时序

8.4.RISC_CPU寻址方式和指令系统

RISC_CPU的指令格式一律为：



它的指令系统仅由8条指令组成。

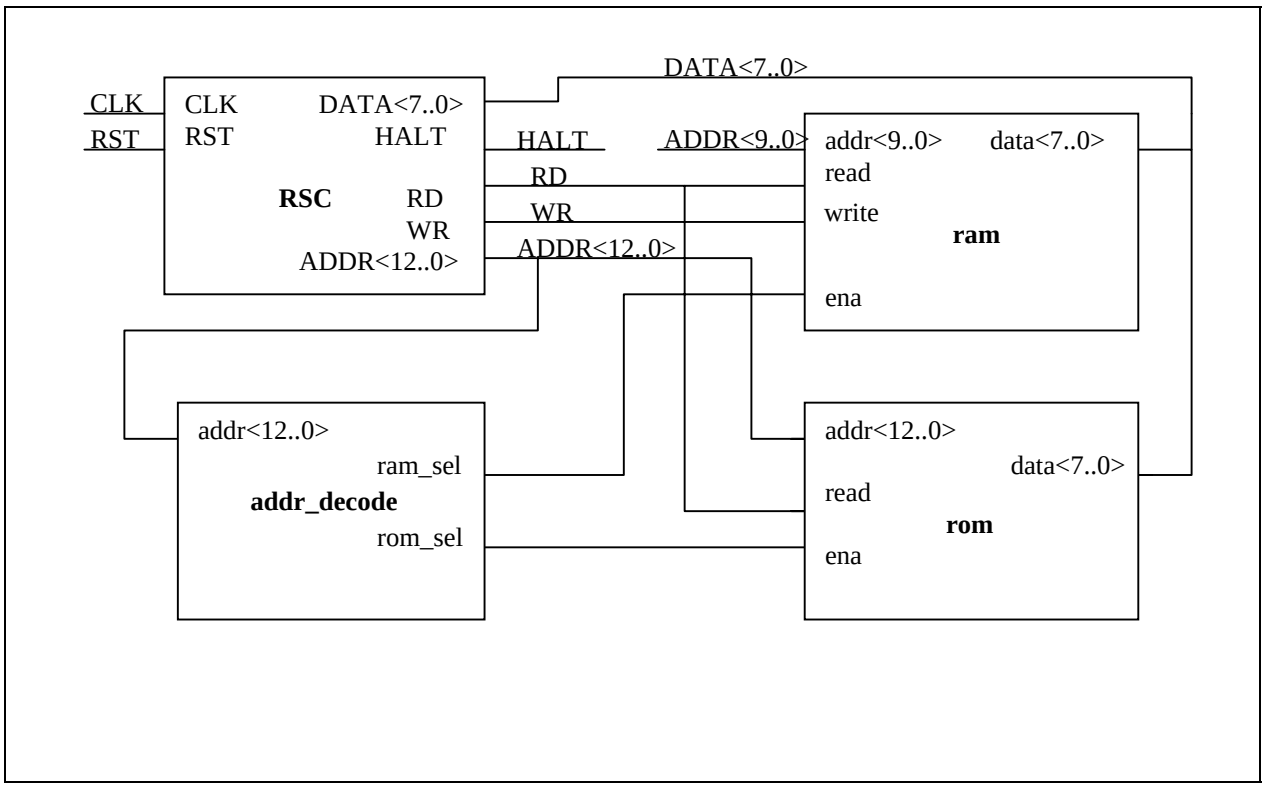
1) HLT 停机操作。该操作将空一个指令周期，即8个时钟周期。
2) SKZ 为零跳过下一条语句。该操作先判断当前alu中的结果是否为零，若是零就跳过下一条语句，否则继续执行。
3) ADD 相加。该操作将累加器中的值与地址所指的存储器或端口的数据相加，结果仍送回累加器中。
4) AND 相与。该操作将累加器的值与地址所指的存储器或端口的数据相与，结果仍送回累加器中。
5) XOR 异或。该操作将累加器的值与指令中给出地址的数据异或，结果仍送回累加器中。
6) LDA 读数据。该操作将指令中给出地址的数据放入累加器。
7) STO 写数据。该操作将累加器的数据放入指令中给出的地址。
8) JMP 无条件跳转语句。该操作将跳转至指令给出的目的地址，继续执行。

RISC_CPU是8位微处理器，一律采用直接寻址方式，即数据总是放在存储器中，寻址单元的地址由指令直接给出。这是最简单的寻址方式。

8.5. RISC_CPU模块的调试

8.5.1. RISC_CPU模块的前仿真

为了对所设计的RISC_CPU模型进行验证，需要把RISC_CPU包装在一个模块下，这样其内部连线就隐蔽起来，从系统的角度看就显得简洁，见图8.5.2。还需要建立一些必要的外围器件模型，例如储存程序用的ROM模型、储存数据用的RAM和地址译码器等。这些模型都可以用VerilogHDL描述，由于不需要综合成具体的电路只要保证功能和接口信号正确就能用于仿真。也就是说,用虚拟器件来代替真实的器件对所设计的RISC_CPU模型进行验证，检查各条指令是否执行正确，与外围电路的数据交换是否正常。这种模块是很容易编写的，上面8.2.9节中的ROM和RAM模块就是简化的虚拟器件的例子，可在下面的仿真中来代替真实的器件，用于验证RISC_CPU模型是否能正确地运行装入ROM和RAM的程序。在RISC_CPU的电路图上加上这些外围电路把有关的电路接通，见图8.5.1;也可以用VerilogHDL模块调用的方法把这些外围电路的模块连接上,这跟用真实的电路器件调试情况很类似，



RISC_CPU和它的外围电路

下面介绍的是在modelsim 5.4 下进行调试的仿真测试程序cputop.v。可用于对以上所设计的RISCCPU进行仿真测试，下面是前仿真的测试程序cputop.v。它的作用是按模块的要求执行仿真，并显示仿真的结果，测试模块cputop.v中的$display和$monitor等系统调用能在计算机的显示屏幕上显示部分测试结果，可以同时用波型观察器观察有关信号的波形。

```
`include "ram.v"
`include "rom.v"
`include "addrdecode.v"
`include "cpu.v"
`timescale 1ns / 100ps
`define PERIOD 100                  // matches clk_gen.v
module t;
  reg reset_req,clock;
  integer test;
  reg [(3*8):0] mnemonic;           //array that holds 3 8-bit ASCII characters
  reg [12:0] PC_addr,IR_addr;
  wire [7:0] data;
  wire [12:0] addr;
  wire rd,wr,halt,ram_sel,rom_sel;
//--------------------------------------------------------------------------
cpu   t_cpu (.clk(clock),.reset(reset_req),.halt(halt),.rd(rd),
                                    .wr(wr),.addr(addr),.data(data));

ram   t_ram  (.addr(addr[9:0]),.read(rd),.write(wr),.ena(ram_sel),.data(data));

rom   t_rom  (.addr(addr),.read(rd),.ena(rom_sel),.data(data));
```

```
addr_decode   t_addr_decode (.addr(addr),.ram_sel(ram_sel),.rom_sel(rom_sel));

//------------------------------------------------------------------------
initial
  begin
    clock=1;
   //display time in nanoseconds
    $timeformat ( -9,  1, " ns", 12);
    display_debug_message;
    sys_reset;
    test1;
    $stop;
    test2;
    $stop;
    test3;
    $stop;
end
  task display_debug_message;
    begin
      $display("\n*********************************************************");
      $display("*  THE FOLLOWING DEBUG TASK ARE AVAILABLE:           *");
      $display("* \"test1; \" to load the 1st diagnostic progran. *");
      $display("*  \"test2; \" to load the 2nd diagnostic program. *");
      $display("*  \"test3; \" to load the Fibonacci program.      *");
      $display("*********************************************************\n");
    end
  endtask
   task test1;
    begin
      test = 0;
      disable MONITOR;
      $readmemb ("test1.pro", t_rom.memory);
      $display("rom loaded   successfully!");
      $readmemb("test1.dat",t_ram.ram);
      $display("ram loaded   successfully!");
      #1 test = 1;
      #14800  ;
      sys_reset;
    end
   endtask

  task test2;
    begin
      test = 0;
      disable MONITOR;
      $readmemb("test2.pro",t_rom.memory);
      $display("rom loaded  successfully!");
      $readmemb("test2.dat",t_ram.ram);
      $display("ram loaded  successfully!");
      #1 test = 2;
      #11600;
      sys_reset;
```

```
    end
   endtask

  task test3;
    begin
      test = 0;
      disable MONITOR;
      $readmemb("test3.pro",t_rom.memory);
      $display("rom loaded  successfully!");
      $readmemb("test3.dat",t_ram.ram);
      $display("ram loaded  successfully!");
      #1 test = 3;
      #94000;
      sys_reset;
    end
   endtask


  task sys_reset;
    begin
      reset_req = 0;
      #(`PERIOD*0.7) reset_req = 1;
      #(1.5*`PERIOD) reset_req = 0;
    end
  endtask

   always @(test)
    begin: MONITOR
      case (test)
       1: begin                          //display results when running test 1
            $display("\n*** RUNNING CPUtest1 - The Basic CPU Diagnostic Program ***");
            $display("\n     TIME           PC       INSTR      ADDR     DATA  ");
            $display("    ----------      ----     -----     -----       ----- ");
             while (test == 1)
                  @(t_cpu.m_adr.pc_addr)//fixed
                  if ((t_cpu.m_adr.pc_addr%2 == 1)&&(t_cpu.m_adr.fetch == 1))//fixed
               begin
                 # 60    PC_addr <=t_cpu.m_adr.pc_addr -1 ;
                         IR_addr <=t_cpu.m_adr.ir_addr;
                # 340   $strobe("%t   %h     %s     %h
%h",$time,PC_addr,mnemonic,IR_addr,data );//HERE DATA HAS BEEN CHANGED
T-CPU-M-REGISTER.DATA
              end

          end

        2: begin
            $display("\n*** RUNNING CPUtest2 - The Advanced CPU Diagnostic Program ***");
              $display("\n     TIME           PC       INSTR      ADDR     DATA  ");
              $display("   ----------      ---       -----       -----    ---- ");
              while (test == 2)
                @(t_cpu.m_adr.pc_addr)
```

```
            if ((t_cpu.m_adr.pc_addr%2 == 1)
                                && (t_cpu.m_adr.fetch == 1))
            begin
             # 60    PC_addr  <= t_cpu.m_adr.pc_addr - 1 ;
                     IR_addr  <= t_cpu.m_adr.ir_addr;
             # 340   $strobe("%t  %h  %s  %h %h", $time, PC_addr,
                                          mnemonic, IR_addr, data );
            end

        end

      3: begin
           $display("\n***   RUNNING CPUtest3 - An Executable Program   ***");
           $display("*** This program should calculate the fibonacci  ***");
           $display("\n    TIME      FIBONACCI NUMBER");
           $display(  "  ---------   -----------------");
           while (test == 3)
            begin
               wait ( t_cpu.m_alu.opcode == 3'h1) // display Fib. No. at end of program loop
               $strobe("%t     %d", $time,t_ram.ram[10'h2]);
               wait ( t_cpu.m_alu.opcode != 3'h1);
            end
        end
    endcase

   end
//-------------------------------------------------------------------------------
--
always @(posedge halt)       //STOP when HALT instruction decoded
    begin
      #500
         $display("\n************************************************************");
         $display("*  A HALT INSTRUCTION WAS PROCESSED  !!!  *");
         $display("************************************************************\n");
    end
always #(`PERIOD/2) clock=~clock;
always  @(t_cpu.m_alu.opcode)
     //get an ASCII mnemonic for each opcode
    case(t_cpu.m_alu.opcode)
     3'b000  : mnemonic ="HLT";
     3'h1    : mnemonic = "SKZ";
     3'h2    : mnemonic = "ADD";
     3'h3    : mnemonic = "AND";
     3'h4    : mnemonic = "XOR";
     3'h5    : mnemonic = "LDA";
     3'h6    : mnemonic = "STO";
     3'h7    : mnemonic = "JMP";
     default : mnemonic = "???";
    endcase

endmodule
```

针对程序做如下说明：测试程序中用`include"“ 形式包含了“ rom.v ”，“ ram.v”和“addrdecode.v”三个外部模块，它们都是检测RISCCPU时必不可少虚拟设备。代表RAM ,ROM和地址译码器，对于RISCCPU，已将它做成一个独立的模块“cpu.v”。具体程序如下：

```
//-------------------------------------------------------------------
`include "clk_gen.v"
`include "accum.v"
`include "adr.v"
`include "alu.v"
`include "machine.v"
`include "counter.v"
`include "machinect1.v"
`include "register.v"
`include "datact1.v"

module cpu(clk,reset,halt,rd,wr,addr,data);
 input clk,reset;
 output rd,wr,addr,halt;
 inout data;
 wire clk,reset,halt;
 wire [7:0]  data;
 wire [12:0] addr;
 wire rd,wr;
 wire clk1,fetch,alu_clk;
 wire [2:0] opcode;
 wire [12:0] ir_addr,pc_addr;
 wire [7:0] alu_out,accum;
 wire zero,inc_pc,load_acc,load_pc,load_ir,data_ena,contr_ena;

clk_gen  m_clk_gen (.clk(clk),.clk1(clk1),.fetch(fetch),
                              .alu_clk(alu_clk),.reset(reset));

register  m_register (.data(data),.ena(load_ir),.rst(reset),
                                 .clk1(clk1),.opc_iraddr({opcode,ir_addr}));

accum      m_accum    (.data(alu_out),.ena(load_acc),
                                    .clk1(clk1),.rst(reset),.accum(accum));

alu        m_alu      (.data(data),.accum(accum),.alu_clk(alu_clk),
                             .opcode(opcode),.alu_out(alu_out),.zero(zero));

machinect1  m_machinec1(.ena(contr_ena),.fetch(fetch),.rst(reset));

machine    m_machine  (.inc_pc(inc_pc),.load_acc(load_acc),.load_pc(load_pc),
                       .rd(rd), .wr(wr), .load_ir(load_ir), .clk1(clk1),
                       .datactl_ena(data_ena), .halt(halt), .zero(zero),
                       .ena(contr_ena),.opcode(opcode));

datactl    m_datactl  (.in(alu_out),.data_ena(data_ena),.data(data));

adr        m_adr (.fetch(fetch),.ir_addr(ir_addr),.pc_addr(pc_addr),.addr(addr));

counter    m_counter  (.ir_addr(ir_addr),.load(load_pc),.clock(inc_pc),
```

```
                                        .rst(reset),.pc_addr(pc_addr));

endmodule
```

其中 contr_ena 用于machinect1与machine之间的ena的连接。cputop.v 中用到下面两条语句需要解释一下：

```
        $readmemb  ( "test1.pro",t_rom_.memory ); 和
        $readmemb  ( "test1.dat",t_ram_.ram);
```

即可把编译好的汇编机器码装人虚拟ROM,把需要参加运算的数据装人虚拟RAM就可以开始仿真。上面语句中的第一项为打开的文件名，后一项为系统层次管理下的ROM模块和RAM模块中的存储器memory和ram。

下面清单所列出是用于测试RISC_CPU基本功能而分别装人虚拟ROM和RAM的机器码和数据文件,其文件名分别为test1.pro,test1.dat,test2.pro,test2.dat,test3.pro,test3.pro和调用这些测试程序进行仿真的程序cputop.v文件：

```
//------------------ 文件  test1.pro ----------------------------------------
/*****************************************************************************
 * Test1 程序是用于验证RISC_ CPU的功能, 是设计工作的重要环节
 * 本程序测试RISC_ CPU的基本指令集，如果RISC_ CPU的各条指令执行正确，
 * 它应在地址为2E(hex)处，在执行HLT时停止运行。
 * 如果该程序在任何其他地址暂停运行，则必有一条指令运行出错。
 * 可参照注释找到出错的指令。
 *****************************************************************************/

机器码          地址       汇编助记符                注释

//------------------------------ test1.pro开始 -------------------------------------------------------
 @00                                                  //address statement
  111_00000     // 00       BEGIN:  JMP TST_JMP
  0011_1100
  000_00000     // 02               HLT             //JMP did not work at all
  0000_0000
  000_00000     // 04               HLT              //JMP did not load PC, it skipped
  0000_0000
  101_11000     // 06      JMP_OK: LDA  DATA_1
  0000_0000
  001_00000     // 08               SKZ
  0000_0000
  000_00000     // 0a               HLT               //SKZ or LDA did not work
  0000_0000
  101_11000     // 0c               LDA  DATA_2
  0000_0001
  001_00000     // 0e               SKZ
  0000_0000
  111_00000     // 10               JMP  SKZ_OK
  0001_0100
  000_00000     // 12               HLT                    //SKZ or LDA did not work
  0000_0000
  110_11000     // 14         SKZ_OK:  STO TEMP               //store non-zero value in TEMP
  0000_0010
  101_11000    // 16                LDA  DATA_1
  0000_0000
```

```
  110_11000    // 18                    STO  TEMP              //store zero value in TEMP
  0000_0010
  101_11000    // 1a                    LDA TEMP
  0000_0010
  001_00000    // 1c                    SKZ                    //check to see if STO worked
  0000_0000
  000_00000    // 1e                    HLT                    //STO did not work
  0000_0000
  100_11000    // 20                    XOR DATA_2
  0000_0001
  001_00000    // 22                    SKZ                    //check to see if XOR worked
  0000_0000
  111_00000    // 24                    JMP XOR_OK
  0010_1000
  000_00000    // 26                    HLT                    //XOR did not work at all
  0000_0000
  100_11000    // 28          XOR_OK:  XOR DATA_2
  0000_0001
  001_00000    // 2a                    SKZ
  0000_0000
  000_00000    // 2c                    HLT                    //XOR did not switch all bits
  0000_0000
  000_00000    // 2e            END:    HLT                    //CONGRATULATIONS - TEST1 PASSED!
  0000_0000
  111_00000    // 30                    JMP BEGIN              //run test again
  0000_0000

@3c
  111_00000    // 3c          TST_JMP:  JMP JMP_OK
  0000_0110
  000_00000    // 3e                    HLT                   //JMP is broken
//-----------------------------test1.pro的结束-----------------------------------------

/*****************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test1.pro使用。
*****************************************************************************************
****/
//-----------------------------test1.dat开始-----------------------------------------------
@00                                        //address  statement  at RAM
  00000000    // 1800    DATA_1:           //constant 00(hex)
  11111111    // 1801    DATA_2:           //constant FF(hex)
  10101010    // 1802    TEMP:             //variable - starts with AA(hex)
//-----------------------------test1.dat的结束-----------------------------------------



/*************************************************************************
 * Test 2程序是用于验证RISC_ CPU的功能, 是设计工作的重要环节
 * 本程序测试RISC_ CPU的高级指令集，如果RISC_ CPU的各条指令执行正确，
 * 它应在地址为20(hex)处，在执行HLT时停止运行。
 * 如果该程序在任何其他地址暂停运行，则必有一条指令运行出错。
 * 可参照注释找到出错的指令。
 * 注意：必须 先在RISC_ CPU 上运行 test1程序成功后，才可运行本程序。
 *************************************************************************/

 机器码          地址       汇编助记符                注释
//-----------------------------test2.pro开始-----------------------------------------------
```

```
@00
  101_11000     // 00  BEGIN:  LDA DATA_2
  0000_0001
  011_11000     // 02            AND DATA_3
  0000_0010
  100_11000     // 04            XOR DATA_2
  0000_0001
  001_00000     // 06            SKZ
  0000_0000
  000_00000     // 08            HLT                    //AND doesn't work
  0000_0000
  010_11000     // 0a            ADD DATA_1
  0000_0000
  001_00000     // 0c            SKZ
  0000_0000
  111_00000     // 0e            JMP ADD_OK
  0001_0010
  000_00000     // 10            HLT                    //ADD doesn't work
  0000_0000
  100_11000     // 12   ADD_OK:  XOR DATA_3
  0000_0010
  010_11000     // 14            ADD DATA_1             //FF plus 1 makes -1
  0000_0000
  110_11000     // 16            STO TEMP
  0000_0011
  101_11000     // 18            LDA DATA_1
  0000_0000
  010_11000     // 1a            ADD TEMP               //-1 plus 1 should make zero
  0000_0011
  001_00000     // 1c            SKZ
  0000_0000
  000_00000     // 1e            HLT                    //ADD Doesn't work
  0000_0000
  000_00000     // 20   END:     HLT                    //CONGRATULATIONS - TEST2 PASSED!
  0000_0000
  111_00000     // 22            JMP BEGIN              //run test again
  0000_0000

//----------------------------test2.pro结束-----------------------------------------


/*****************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test2.pro使用。
*****************************************************************************************
****/
//----------------------------test2.dat开始-----------------------------------------------
@00
  00000001      // 1800          DATA_1:                //constant  1(hex)
  10101010      // 1801          DATA_2:                //constant AA(hex)
  11111111      // 1802          DATA_3:                //constant FF(hex)
  00000000      // 1803          TEMP:
//----------------------------test2.dat结束 .---------------------------------------------


/*****************************************************************************
 * Test 3 程序是一个计算从0到144的Fibonacci 序列的程序，用于进一步验证RISC_ CPU的功能。
 * 所谓Fibonacci 序列就是一系列数其中每一个数都是它前面两个数的和（如：0，1，1，2，3，5，
 * 8，13，21，.....）。 这种序列常用于财务分析。
```

```
 * 注意：必须在成功地运行前两个测试程序后才运行本程序 。否则很难发现问题所在。
 ********************************************************************************/
机器码            地址        汇编助记符                  注释
//------------------------------test3.pro开始---------------------------------------------------------------

@00
  101_11000      // 00   LOOP:  LDA FN2              //load value in FN2 into accum
  0000_0001
  110_11000      // 02           STO TEMP             //store accumulator in TEMP
  0000_0010
  010_11000     // 04           ADD FN1              //add value in FN1 to accumulator
  0000_0000
  110_11000     // 06           STO FN2              //store result in FN2
  0000_0001
  101_11000     // 08           LDA TEMP             //load TEMP into the accumulator
  0000_0010
  110_11000      // 0a           STO FN1              //store accumulator in FN1
  0000_0000
  100_11000      // 0c           XOR LIMIT            //compare accumulator to LIMIT
  0000_0011
  001_00000      // 0e           SKZ                  //if accum = 0, skip to DONE
  0000_0000
  111_00000      // 10           JMP LOOP             //jump to address of LOOP
  0000_0000
  000_00000     // 12    DONE:  HLT                  //end of program
  0000_0000
//-----------------------------test3.pro结束-----------------------------------------

/*************************************************************************************
**
下面文件中的数据在仿真时需要用系统任务$readmemb读入RAM，才能被上面的汇编程序test3.pro使用。
*************************************************************************************
****/
//-----------------------------test3.dat开始-----------------------------------------

@00
   00000001     // 1800   FN1:          //data storage for 1st Fib. No.
  00000000      // 1801   FN2:           //data storage for 2nd Fib. No.
  00000000      // 1802   TEMP:         //temproray data storage
  10010000     // 1803    LIMIT:       //max value to calculate 144(dec)
//-----------------------------test3.pro结束-----------------------------------------
```

以下介绍前仿真的步骤，首先按照表示各模块之间连线的电路图编制测试文件，即定义Verilog的wire变量作为连线，连接各功能模块之间的引脚，并将输入信号引入，输出信号引出。如若需要，可加入必要的语句显示提示信息。例如，risc_cpu 的测试文件就是cputop.v。其次，使用仿真软件进行仿真，由于不同的软件使用方法可能有较大的差异，以下只简单的介绍modelsim的使用。在进入modelsim的环境之后，在file项选择change direction来确定编制的文件所在的目录，然后在design项选择或创建一个library，完成后即可开始编译。在design项选compile...项，进入编译环境，选定要编译的文件进行编译。Modelsim的编译器语法检查并不严格，有时会出现莫名其妙的逻辑错误，书写时应注意笔误。完成编译后，还是在compile...项，选择load new design项，选中编译后提示的top module的名字，然后开始仿真。在view项可选波形显示，信号选择，功能和操作简单明了，这里就不一一赘述。

仿真结果如下：

```
run -all
#
# ****************************************************
```

```
# *  THE FOLLOWING DEBUG TASK ARE AVAILABLE:           *
# * "test1; " to load the 1st diagnostic progran. *
# *  "test2; " to load the 2nd diagnostic program. *
# *  "test3; " to load the Fibonacci program.      *
# ****************************************************
#
# rom loaded   successfully!
# ram loaded   successfully!
#
# *** RUNNING CPUtest1 - The Basic CPU Diagnostic Program ***
#
#     TIME         PC        INSTR    ADDR     DATA
#    ----------  ---       -----    ----    -----
#    1200.0 ns   0000       JMP     003c       zz
#    2000.0 ns   003c       JMP     0006       zz
#    2800.0 ns   0006       LDA     1800       00
#    3600.0 ns   0008       SKZ     0000       zz
#    4400.0 ns   000c       LDA     1801       ff
#    5200.0 ns   000e       SKZ     0000       zz
#    6000.0 ns   0010       JMP     0014       zz
#    6800.0 ns   0014       STO     1802       ff
#    7600.0 ns   0016       LDA     1800       00
#    8400.0 ns   0018       STO     1802       00
#    9200.0 ns   001a       LDA     1802       00
#   10000.0 ns   001c       SKZ     0000       zz
#   10800.0 ns   0020       XOR     1801       ff
#   11600.0 ns   0022       SKZ     0000       zz
#   12400.0 ns   0024       JMP     0028       zz
#   13200.0 ns   0028       XOR     1801       ff
#   14000.0 ns   002a       SKZ     0000       zz
#   14800.0 ns   002e       HLT     0000       zz
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/Cputop.v line 109
run -continue
# rom loaded  successfully!
# ram loaded  successfully!
#
# *** RUNNING CPUtest2 - The Advanced CPU Diagnostic Program ***
#
#      TIME        PC     INSTR  ADDR  DATA
#    ----------   ----   -----  ----  -----
#   16200.0 ns   0000    LDA   1801   aa
#   17000.0 ns   0002    AND   1802   ff
#   17800.0 ns   0004    XOR   1801   aa
#   18600.0 ns   0006    SKZ   0000   zz
#   19400.0 ns   000a    ADD   1800   01
#   20200.0 ns   000c    SKZ   0000   zz
#   21000.0 ns   000e    JMP   0012   zz
```

```
#   21800.0 ns   0012    XOR   1802   ff
#   22600.0 ns   0014    ADD   1800   01
#   23400.0 ns   0016    STO   1803   ff
#   24200.0 ns   0018    LDA   1800   01
#   25000.0 ns   001a    ADD   1803   ff
#   25800.0 ns   001c    SKZ   0000   zz
#   26600.0 ns   0020    HLT   0000   zz
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/cputop.v line 111
run -continue
# rom loaded  successfully!
# ram loaded  successfully!
#
# ***   RUNNING CPUtest3 - An Executable Program   ***
# *** This program should calculate the fibonacci  ***
#
#     TIME      FIBONACCI NUMBER
#   ---------   -----------------
#   33250.0 ns       0
#   40450.0 ns       1
#   47650.0 ns       1
#   54850.0 ns       2
#   62050.0 ns       3
#   69250.0 ns       5
#   76450.0 ns       8
#   83650.0 ns      13
#   90850.0 ns      21
#   98050.0 ns      34
#  105250.0 ns      55
#  112450.0 ns      89
#  119650.0 ns     144
#
# ************************************************************
# *  A HALT INSTRUCTION WAS PROCESSED  !!!  *
# ************************************************************
#
# Break at H:/seda/w/cputop.v line 112
```

在运行了以上程序后，如仿真程序运行的结果正确，前仿真（即布局布线前的仿真）可告结束。

8.5.2. RISC_CPU模块的综合

在对所设计的RISC_CPU模型进行验证后，如没有发现问题就可开始做下一步的工作即综合。综合工作往往要分阶段来进行，这样便于发现问题。

所谓分阶段就是指：

第一阶段：先对构成RISC_CPU模型的各个子模块，如状态控制机模块（包括machine模块，machinectl模块）、指令寄存器模块（register模块）、算术逻辑运算单元模块（alu模块）等，分别加以综合以检查其可综合性，综合后及时进行后仿真，这样便于及时发现错误，及时改进。

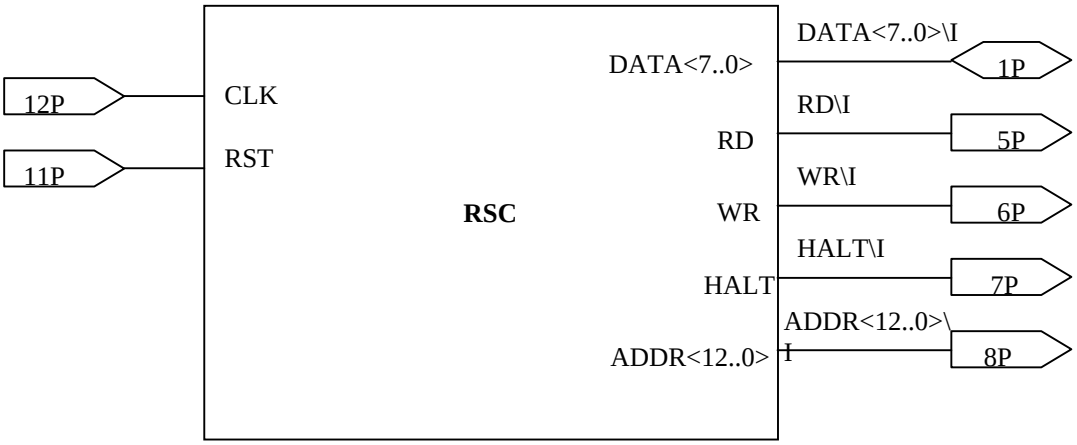


图8.5.2 用于综合的RISC_CPU模块（RSC）

第二阶段：把要综合的模块从仿真测试信号模块和虚拟外围电路模型（如ROM模块、RAM模块、显示部件模块等）中分离出来，组成一个独立的模块，其中包括了所有需要综合的模块。然后给这个大模块起一个名字，如本章中的例子，我们要综合的只是RISC_CPU并不包括虚拟外围电路，可以给这一模块起一个名字，例如称它为RSC_CHIP模块。如用电路图描述的话，我们还需给它的引脚加上标准的引脚部件并加标记，见图8.5.2。

第三阶段：加载需要综合的模块到综合器，本例所使用的综合器是 Synplify，选定的 FPGA 是 Altera FLEX10K，针对它的库进行综合。

综合器综合的结果会产生一系列的文件，其中有一个文件报告用了所使用的基本单元，各部件的时间参数以及综合的过程。见下面的报告,它就是这个RISC_CPU芯片所用的综合报告，综合所用的库为Altera FLEX10K系列的FPGA库。

```
$ Start of Compile
#Fri Jul 21 10:11:03 2000

Synplify Verilog Compiler, version 5.2.2, built Aug 20 1999
Copyright (C) 1994-1999, Synplicity Inc.  All Rights Reserved

@I::"h:\seda\w\cpu.v"
Verilog syntax check successful!
Selecting top level module cpu
Synthesizing module clk_gen
Synthesizing module register
Synthesizing module accum
Synthesizing module alu
Synthesizing module machinect1
Synthesizing module machine
Synthesizing module datactl
```

```
Synthesizing module adr
Synthesizing module counter
Synthesizing module cpu
@END
Process took 0.491 seconds realtime, 0.54 seconds cputime
Synplify Altera Technology Mapper, version 5.2.2, built Aug 31 1999
Copyright (C) 1994-1998, Synplicity Inc.  All Rights Reserved
Loading timing data for chip EPF10K10-3
List of partitions to map:
   view:work.cpu(verilog)
Automatic dissolve at startup in view:work.cpu(verilog) of m_counter(counter)
Automatic dissolve at startup in view:work.cpu(verilog) of m_adr(adr)
Automatic dissolve at startup in view:work.cpu(verilog) of m_datactl(datactl)
Automatic dissolve at startup in view:work.cpu(verilog) of m_machinec1(machinect1)
@N:"h:\seda\w\cpu.v":347:0:347:5|Found counter in view:work.cpu(verilog) inst
m_counter.pc_addr[12:0]
Automatic dissolve during optimization of view:work.cpu(verilog) of m_alu(alu)
Automatic dissolve during optimization of view:work.cpu(verilog) of m_accum(accum)
Loading timing data for chip EPF10K10-3
Found clock m_machine.inc_pc with period 100ns
Found clock m_clk_gen.fetch with period 100ns
Found clock m_clk_gen.alu_clk with period 100ns
Found clock clk with period 100ns

              ##### START TIMING REPORT #####
Set the Environment Variable SYNPLIFY_TIMING_REPORT_OLD to get the old timing report


              Performance Summary
             *********************

                        Requested     Estimated     Requested     Estimated
Clock                   Frequency     Frequency     Period        Period        Slack
--------------------------------------------------------------------------------
m_machine.inc_pc        10.0 MHz      95.2 MHz      100.0         10.5          89.5
m_clk_gen.alu_clk       10.0 MHz      59.5 MHz      100.0         16.8          83.2
clk                     10.0 MHz      16.8 MHz      100.0         59.5          40.5
================================================================================

              Interface Information
             ***********************

Input Ports:

Port        Reference                       User        Arrival     Required
Name        Clock                           Constraint  Time        Time        Slack
--------------------------------------------------------------------------------
----
clk         System                          0.0           0.0         >2000.0     NA
data[0]     m_clk_gen.alu_clk [rising]      0.0           0.0         85.9        85.9
data[1]     m_clk_gen.alu_clk [rising]      0.0           0.0         86.2        86.2
data[2]     m_clk_gen.alu_clk [rising]      0.0           0.0         86.5        86.5
```

```
data[3]     m_clk_gen.alu_clk [rising]     0.0          0.0         87.0         87.0
data[4]     m_clk_gen.alu_clk [rising]     0.0          0.0         88.8         88.8
data[5]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.1         89.1
data[6]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.4         89.4
data[7]     m_clk_gen.alu_clk [rising]     0.0          0.0         89.9         89.9
reset       clk [falling]                  0.0          0.0         91.9         91.9
================================================================================
====

Output Ports:

Port          Reference                      User       Arrival     Required
Name          Clock                       Constraint     Time        Time         Slack
--------------------------------------------------------------------------------
-----
addr[0]      clk [falling]                  0.0          6.9         100.0        93.1
addr[1]      clk [falling]                  0.0          6.9         100.0        93.1
addr[2]      clk [falling]                  0.0          6.9         100.0        93.1
addr[3]      clk [falling]                  0.0          6.9         100.0        93.1
addr[4]      clk [falling]                  0.0          6.9         100.0        93.1
addr[5]      clk [falling]                  0.0          6.9         100.0        93.1
addr[6]      clk [falling]                  0.0          6.9         100.0        93.1
addr[7]      clk [falling]                  0.0          6.9         100.0        93.1
addr[8]      clk [falling]                  0.0          6.9         100.0        93.1
addr[9]      clk [falling]                  0.0          6.9         100.0        93.1
addr[10]     clk [falling]                  0.0          6.9         100.0        93.1
addr[11]     clk [falling]                  0.0          6.9         100.0        93.1
addr[12]     clk [falling]                  0.0          6.9         100.0        93.1
data[0]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[1]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[2]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[3]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[4]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[5]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[6]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
data[7]      m_clk_gen.alu_clk [rising]     0.0          0.0         100.0        100.0
halt         clk [rising]                   0.0          1.0         100.0        99.0
rd           clk [rising]                   0.0          1.0         100.0        99.0
wr           clk [rising]                   0.0          1.0         100.0        99.0
================================================================================

             Detailed Timing Report for  clock : clk
             ****************************************
Requested Period      100.0 ns
Estimated Period      59.5 ns
Worst Slack          40.5 ns

Start Points for Paths with Slack Worse than 42.8 ns :

                                                                    Arrival
Instance                Type      Pin     Net                       Time        Slack
--------------------------------------------------------------------------------
```

```
m_machine.load_ir       S_DFF     Q       m_machine.load_ir       51.4        40.5
m_machine.load_acc      S_DFF     Q       m_machine.load_acc      51.0        41.1
================================================================================

End Points for Paths with Slack Worse than 42.8 ns :

                                                                      Required
Instance                    Type      Pin     Net                     Time        Slack
--------------------------------------------------------------------------------
----------
m_register.opc_iraddr[8]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[7]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[6]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[5]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[4]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[3]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[2]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[1]    S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[15]   S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
m_register.opc_iraddr[14]   S_DFFE    ENA     m_register.un1_un1_rst    97.8        40.5
================================================================================

A Critical Path with worst case slack = 40.5 ns:

Instance/Net                          Pin       Pin      Arrival     Delta     Fan
Name                        Type      Name      Dir      Time        Delay     Out
--------------------------------------------------------------------------------
m_machine.load_ir           S_DFF     Q         Out      51.4        51.4
m_machine.load_ir           Net                                              2
m_register.un1_un1_rst      S_LUT     I1        In       51.4
m_register.un1_un1_rst      S_LUT     OUT       Out      57.3        5.9
m_register.un1_un1_rst      Net                                              16
m_register.opc_iraddr[0]    S_DFFE    ENA       In       57.3
================================================================================
Setup requirement on this path is 2.2 ns.


            Detailed Timing Report for  clock : m_clk_gen.alu_clk
            ********************************************
Requested Period      100.0 ns
Estimated Period      16.8 ns
Worst Slack          83.2 ns

Start Points for Paths with Slack Worse than 85.5 ns :

                                                                      Arrival
Instance                    Type      Pin     Net                     Time        Slack
--------------------------------------------------------------------------------
m_accum.accum[1]            S_DFFE    Q       m_accum.accum[1]          3.0         83.2
m_accum.accum[0]            S_DFFE    Q       m_accum.accum[0]          2.6         83.3
m_accum.accum[2]            S_DFFE    Q       m_accum.accum[2]          2.6         83.9
m_accum.accum[3]            S_DFFE    Q       m_accum.accum[3]          2.6         84.4
```

```
m_register.opc_iraddr[14]    S_DFFE    Q     m_register.opc_iraddr[14]    4.4       85.5
================================================================================

End Points for Paths with Slack Worse than 85.5 ns :

                                                                  Required
Instance              Type      Pin     Net                       Time          Slack
--------------------------------------------------------------------------------
m_alu.alu_out[3]      S_DFF     D       m_alu.alu_out_11_5[3]     97.8          83.2
m_alu.alu_out[2]      S_DFF     D       m_alu.alu_out_11_5[2]     97.8          83.5
m_alu.alu_out[0]      S_DFF     D       m_alu.alu_out_11_5[0]     97.8          84.4
m_alu.alu_out[7]      S_DFF     D       m_alu.alu_out_11_5[7]     97.8          84.9
m_alu.alu_out[6]      S_DFF     D       m_alu.alu_out_11_5[6]     97.8          85.2
m_alu.alu_out[5]      S_DFF     D       m_alu.alu_out_11_5[5]     97.8          85.5
================================================================================

A Critical Path with worst case slack = 83.2 ns:

Instance/Net                          Pin      Pin      Arrival     Delta     Fan
Name                        Type      Name     Dir      Time        Delay     Out
--------------------------------------------------------------------------------
m_accum.accum[1]            S_DFFE    Q        Out      3.0         3.0
m_accum.accum[1]            Net                                              6
m_alu.un2_alu_out_add1      S_CAR     I1       In       3.0
m_alu.un2_alu_out_add1      S_CAR     COUT     Out      4.2         1.2
m_alu.un2_alu_out_carry_1   Net                                              1
m_alu.un2_alu_out_add2      S_CAR     CIN      In       4.2
m_alu.un2_alu_out_add2      S_CAR     COUT     Out      4.5         0.3
m_alu.un2_alu_out_carry_2   Net                                              1
m_alu.un2_alu_out_add3      S_CAR     CIN      In       4.5
m_alu.un2_alu_out_add3      S_CAR     OUT      Out      6.7         2.2
m_alu.un2_alu_out_add3      Net                                              1
m_alu.alu_out_11_1[3]       S_LUT     I1       In       6.7
m_alu.alu_out_11_1[3]       S_LUT     OUT      Out      9.6         2.9
m_alu.alu_out_11_1[3]       Net                                              1
m_alu.alu_out_11_2[3]       S_LUT     I2       In       9.6
m_alu.alu_out_11_2[3]       S_LUT     OUT      Out      12.5        2.9
m_alu.alu_out_11_2[3]       Net                                              1
m_alu.alu_out_11_5[3]       S_LUT     I1       In       12.5
m_alu.alu_out_11_5[3]       S_LUT     OUT      Out      14.6        2.1
m_alu.alu_out_11_5[3]       Net                                              1
m_alu.alu_out[3]            S_DFF     D        In       14.6
================================================================================
Setup requirement on this path is 2.2 ns.


             Detailed Timing Report for  clock : m_machine.inc_pc
             ****************************************
Requested Period      100.0 ns
Estimated Period      10.5 ns
Worst Slack          89.5 ns
```

Start Points for Paths with Slack Worse than 91.8 ns :

Instance	Type	Pin	Net	Arrival Time	Slack
m_machine.load_pc	S_DFF	Q	m_machine.load_pc	1.0	89.5
m_counter.pc_addr[0]	S_DFF	Q	m_counter.pc_addr[0]	1.4	90.5
m_counter.pc_addr[1]	S_DFF	Q	m_counter.pc_addr[1]	1.4	90.8
m_register.opc_iraddr[0]	S_DFFE	Q	m_register.opc_iraddr[0]	1.8	91.0
m_register.opc_iraddr[1]	S_DFFE	Q	m_register.opc_iraddr[1]	1.8	91.0
m_register.opc_iraddr[2]	S_DFFE	Q	m_register.opc_iraddr[2]	1.8	91.0
m_register.opc_iraddr[3]	S_DFFE	Q	m_register.opc_iraddr[3]	1.8	91.0
m_register.opc_iraddr[4]	S_DFFE	Q	m_register.opc_iraddr[4]	1.8	91.0
m_register.opc_iraddr[5]	S_DFFE	Q	m_register.opc_iraddr[5]	1.8	91.0
m_register.opc_iraddr[6]	S_DFFE	Q	m_register.opc_iraddr[6]	1.8	91.0

End Points for Paths with Slack Worse than 91.8 ns :

Instance	Type	Pin	Net	Required Time	Slack
m_counter.pc_addr[0]	S_DFF	D	m_counter.pc_addr_lm0	97.8	89.5
m_counter.pc_addr[1]	S_DFF	D	m_counter.pc_addr_lm1	97.8	89.5
m_counter.pc_addr[2]	S_DFF	D	m_counter.pc_addr_lm2	97.8	89.5
m_counter.pc_addr[3]	S_DFF	D	m_counter.pc_addr_lm3	97.8	89.5
m_counter.pc_addr[4]	S_DFF	D	m_counter.pc_addr_lm4	97.8	89.5
m_counter.pc_addr[5]	S_DFF	D	m_counter.pc_addr_lm5	97.8	89.5
m_counter.pc_addr[6]	S_DFF	D	m_counter.pc_addr_lm6	97.8	89.5
m_counter.pc_addr[7]	S_DFF	D	m_counter.pc_addr_lm7	97.8	89.5
m_counter.pc_addr[8]	S_DFF	D	m_counter.pc_addr_lm8	97.8	89.5
m_counter.pc_addr[9]	S_DFF	D	m_counter.pc_addr_lm9	97.8	89.5

A Critical Path with worst case slack = 89.5 ns:

Instance/Net Name	Type	Pin Name	Pin Dir	Arrival Time	Delta Delay	Fan Out
m_machine.load_pc	S_DFF	Q	Out	1.0	1.0	
m_machine.load_pc	Net					1
m_machine.load_pc_i	S_LUT	I0	In	1.0		
m_machine.load_pc_i	S_LUT	OUT	Out	6.8	5.8	
m_machine.load_pc_i	Net					13
m_counter.pc_addr_lm0	S_MUX21	SEL	In	6.8		
m_counter.pc_addr_lm0	S_MUX21	Z	Out	8.3	1.5	
m_counter.pc_addr_lm0	Net					1
m_counter.pc_addr[0]	S_DFF	D	In	8.3		

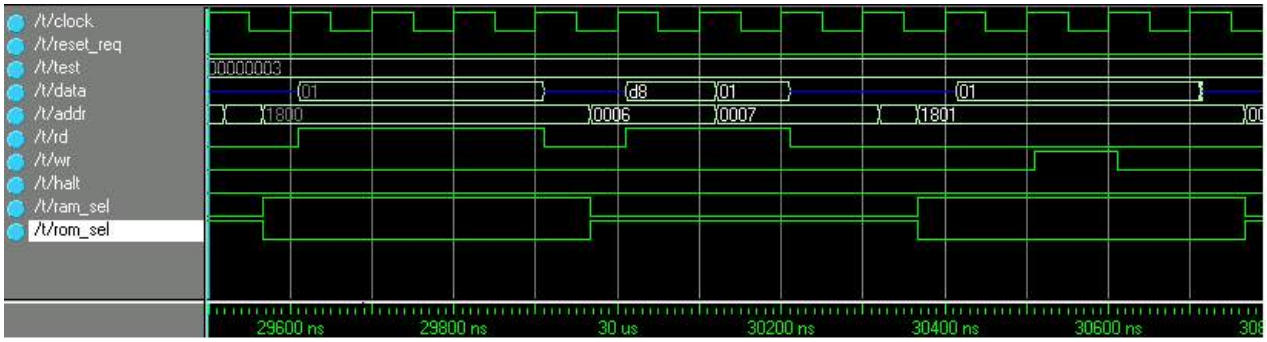
Setup requirement on this path is 2.2 ns.

```
                ##### END TIMING REPORT #####


---------------------------------------
Resource Usage Report

Synplify is performing all technology mapping
Post place and route resource use may vary a small
amount due to logic cell replication and register packing
decisions during place and route.

Design view:work.cpu(verilog)
Selecting part epf10k10lc84-3

Logic resources:  149 LCs of 576 (25%)
Number of Nets:   265
Number of Inputs: 926
Register bits:    69 (26 using enable)
I/O cells:       26

Details:
Cells in logic mode:   116
Cells in arith mode:   8
Cells in cascade mode: 10
Cells in counter mode: 13
DFFs with no logic:    2  (uses cell for routing)
LUTs driving both DFF and logic: 2


Found clock clk with period 100ns
Found clock alu_clk with period 100ns
Found clock fetch with period 100ns
Found clock inc_pc with period 100ns
Enabling timing driven placement for new ACF file.
All Constraints processed!
Mapper successful!
Process took 7.03 seconds realtime, 7.1 seconds cputime


//-------------------- RISC_CPU芯片综合结果报告结束----------------------------
```

8.5.3.RISC_CPU模块的优化和布局布线

选定部件库后就可以对所设计的RISC_CPU模型进行综合,综合后产生了一系列的文件,其中XXXXX.edf文件就是与所选定的厂家部件库对应的电子设计交换格式（Electronic Design Interchange Format）文件或是与某一类部件库（如通用FPGA库）对应的电子设计交换格式文件，这也就是在电路设计工业界常说的EDIF格式文件。在产生了XXXXX.edf文件之后，就要进行后仿真。以下介绍的是Altera Max+II 9.3 进行布线，在使用时在软件相应项选取所得到的cpu.edf文件，相对应的部件库（Altera FLEX10K）以及输出格式（verilog）。布线完成后得到两个文件cpu.vo和alt_max2.vo。cpu.vo是所设计的RISC_CPU的门级结构，即利用Verilog 语法描述的用alt_max2部件库中的基本元件构成的复杂电路连线网络，而alt_max2.vo是cpu.vo所引用的门级模型的库文件，包含各种基本类

型的电路的门级模型，它们的参数与真实器件完全一致，包括如延迟等参数。将这两个文件包含在cputop.v中，来代替原来的RTL级子模块，用仿真器再进行一次仿真，此时称为后仿真。实际上，后仿真与前仿真的根本区别在于测试文件所包含的模型的结构不同。前仿真使用的是一种RTL级模型，如cpu.v,而后仿真使用的是门级结构模型，其中不但有逻辑关系还包含实际门级电路和布线的延迟，还有驱动能力的问题。仔细观察后仿真波形就会发现与前仿真相比较，各信号的变化与时钟沿之间存在着延迟，这在前仿真时并未反映出来。



后仿真波形

下面的Verilog程序是由布局布线工具生成的，分别命名为cpu.vo和alt_max2.vo。由于cpu.vo是门级描述，共有上千行，而alt_max2.vo是仿真用库用UDP描述，也有几百行，无法在课本上全部列出，只能从中截取一小片段供同学参考。有兴趣的同学可以用 Verilog 语法中有关门级描述和用户自定义源语（UDP）来理解，由于是门级模型，又有布线的延迟，所以可以来验证电路结构是否符合设计要求。

```
/**********************cpu.vo开始*************************
// MAX+plus II Version 9.3 RC3 7/20/1999
// Sun Jul 30 10:53:36 2000

//
`timescale 100 ps / 100 ps


module cpu (
       addr,
       data,
       CLK,
       reset,
       halt,
       rd,
       wr);

output [12:0] addr;
inout [7:0] data;
input CLK;
input reset;
output halt;
output rd;
output wr;
supply0 gnd;
```

```
supply1 vcc;

wire
    \|accum:m_accum|accum_0_.CLK , \|accum:m_accum|accum_0_.D ,
\|accum:m_accum|accum_0_.ENA ,
    \|accum:m_accum|accum_0__Q , \|accum:m_accum|accum_1_.CLK ,
\|accum:m_accum|accum_1_.D ,
    …
    …
    …

    TRIBUF0_cpu TRIBUF_2
( .Y(data[0]), .IN1(N_126), .OE(\|datactl:m_datactl|data_0_.OE ) );
    TRIBUF0_cpu TRIBUF_4
( .Y(data[1]), .IN1(N_135), .OE(\|datactl:m_datactl|data_1_.OE ) );
    TRIBUF0_cpu TRIBUF_6
( .Y(data[2]), .IN1(N_144), .OE(\|datactl:m_datactl|data_2_.OE ) );
    TRIBUF0_cpu TRIBUF_8
( .Y(data[3]), .IN1(N_153), .OE(\|datactl:m_datactl|data_3_.OE ) );
    …
    …
AND1 AND1_49 ( \|datactl:m_datactl|data_0_.OE , N_124 );
DELAY DELAY_50 ( N_124, \|machine:m_machine|datactl_ena1_Q  );
defparam DELAY_50.TPD = 40;
DELAY DELAY_51 ( N_126, N_127 );
XOR2 XOR2_52 ( N_127, N_128, N_132 );
....

module DFF0_cpu ( Q, D, CLK, CLRN, PRN );
    input D;
    input CLK;
    input CLRN;
    input PRN;
    output Q;
    PRIM_DFF (Q, D, CLK, CLRN, PRN);

    wire legal;
    and(legal, CLRN, PRN);
    specify

        specparam TREG = 9;
        specparam TRSU = 13;
        specparam TRH  = 14;
        specparam TRPR = 10;
        specparam TRCL = 10;

        $setup  (  D, posedge CLK &&& legal, TRSU  ) ;
        $hold   (  posedge CLK &&& legal, D, TRH   ) ;

        ( negedge CLRN => (Q  +: 1'b0)) = ( TRCL, TRCL ) ;
        ( negedge PRN  => (Q  +: 1'b1)) = ( TRPR, TRPR ) ;
        ( posedge CLK  => (Q  +: D)) = ( TREG, TREG ) ;
```

```
    endspecify
endmodule
…
…
/******************cpu.vo结束********************

/*****************alt_max2.vo开始****************
//
// MAX+plus II Version 9.3 RC3 7/20/1999
// Sun Jul 30 10:53:36 2000

//

//`define SDF_IOPATH
`timescale 100 ps / 100 ps

primitive PRIM_DFF  (Q, D, CP, RB, SB);

    output Q;
    input  D, CP, RB, SB;
    reg    Q;

    initial Q = 1'b0;

    // FUNCTION : POSITIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
    //            ASYNCHRONOUS SET AND CLEAR. ( Q OUTPUT UDP ).

    table

    //  D   CP      RB  SB  :   Qt  :   Qt+1

        1   (01)    1   1   :   ?   :   1;  // clocked data
        1   (01)    1   x   :   ?   :   1;  // pessimism

        1    ?      1   x   :   1   :   1;  // pessimism

        0    0      1   x   :   1   :   1;  // pessimism
        0    x      1 (?x)  :   1   :   1;  // pessimism
        0    1      1 (?x)  :   1   :   1;  // pessimism
…….

primitive PRIM_LATCH  (Q, ENA, D);
    input  D;
    input  ENA;
    output Q; reg Q;

    table

    // ENA    D    Q   Q+
         0    ?  : ? : -;
         1    0  : ? : 0;
```

```
        1    1  : ? : 1;  //

    endtable

endprimitive
……

`celldefine
module AND1 ( Y, IN1 );
    parameter TPD = 0;
    input IN1;
    output Y;

    and #TPD (Y, IN1);


`ifdef SDF_IOPATH
    specify
        (IN1 => Y) = (0,0);
    endspecify
`endif

endmodule
…
/**********************alt_max2.vo结束******************
```

不同FPGA厂家的布局布线工具提供不同的后仿真解决方法。所以很难用一句话作全面的介绍，读者应阅读FPGA厂家的布局布线工具的说明书中有关章节，选用正确的Verilog门级结构的后仿真解决方案。如后仿真正确无误，就可以把布局布线后生成的一系列文件送ASIC厂家或加载到FPGA器件的编码工具，使其变为专用的电路芯片。如后仿真中发现有错误，可先降低测试信号模块的主时钟频率，如该问题解决了，则需要找到造成问题的关键路径，下一次在布局布线时应先布关键的路径（即在约束文件中注明该路径是关键路径后，再重做自动布局布线），若还有问题则需检查各模块中是否有个别模块没有按照同步设计的原则。若是，则需改写有关的VerilogHDL模块。重复以上工作，直到后仿真正确无误。以上所述的就是用VerilogHDL设计一个复杂数字电路系统的步骤。读者可以参考以上步骤，自己来设计一个可在FPGA上实现的小RISC_CPU系统。

思考题

1）请叙述一下设计一个复杂数字系统的步骤。
2） 综合一个大型的数字系统需要注意什么？
3） 请改进以上RISC_CPU，把指令数增至16，寻址空间降为4K。
4） 什么叫软硬件联合仿真?为什么说Verilog语言支持软硬件联合设计?

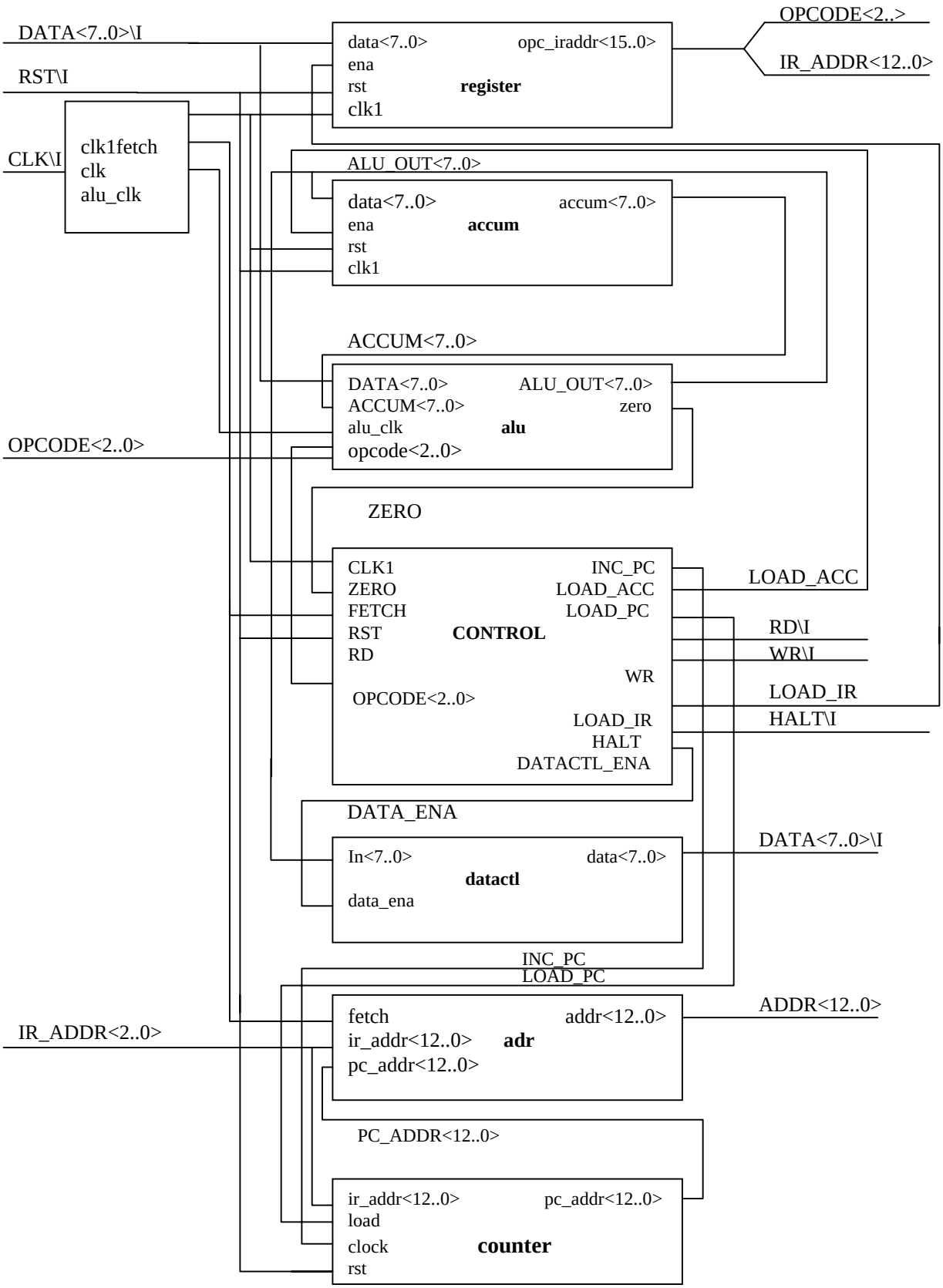


图8。1 RISC——CPU中各部件的相互连接关系