

```

begin
    link_read      <= YES;
    FF            <= 1;
    sh8in_state   <= sh8in_bit7;
end
else
    sh8in_state   <= sh8in_end;
default: begin
    link_read      <= NO;
    sh8in_state   <= sh8in_bit7;
end
endcase
end
endtask

//-----并行数据转换为串行数据任务-----
task shift8_out;
begin
casex(sh8out_state)
    sh8out_bit7:
        if(! SCL)
            begin
                link_sda      <= YES;
                link_write    <= YES;
                sh8out_state  <= sh8out_bit6;
            end
        else
            sh8out_state  <= sh8out_bit7;
    sh8out_bit6:
        if(! SCL)
            begin
                link_sda      <= YES;
                link_write    <= YES;
                sh8out_state  <= sh8out_bit5;
                sh8out_buf    <= sh8out_buf<<1;
            end
        else
            sh8out_state  <= sh8out_bit6;
    sh8out_bit5:
        if(! SCL)
            begin
                sh8out_state  <= sh8out_bit4;
                sh8out_buf    <= sh8out_buf<<1;
            end
        else
            sh8out_state  <= sh8out_bit5;
endcase
end

```

```
        end
    else
        sh8out_state      <= sh8out_bit5;
sh8out_bit4:
    if(! SCL)
        begin
            sh8out_state      <= sh8out_bit3;
            sh8out_buf       <= sh8out_buf<<1;
        end
    else
        sh8out_state      <= sh8out_bit4;
sh8out_bit3:
    if(! SCL)
        begin
            sh8out_state      <= sh8out_bit2;
            sh8out_buf       <= sh8out_buf<<1;
        end
    else
        sh8out_state      <= sh8out_bit3;
sh8out_bit2:
    if(! SCL)
        begin
            sh8out_state      <= sh8out_bit1;
            sh8out_buf       <= sh8out_buf<<1;
        end
    else
        sh8out_state      <= sh8out_bit2;
sh8out_bit1:
    if(! SCL)
        begin
            sh8out_state      <= sh8out_bit0;
            sh8out_buf       <= sh8out_buf<<1;
        end
    else
        sh8out_state      <= sh8out_bit1;
sh8out_bit0:
    if(! SCL)
        begin
            sh8out_state      <= sh8out_end;
            sh8out_buf       <= sh8out_buf<<1;
        end
    else
        sh8out_state      <= sh8out_bit0;
```

```

sh8out_end;
    if(! SCL)
        begin
            link_sda      <= NO;
            link_write    <= NO;
            FF           <= 1;
        end
    else
        sh8out_state   <= sh8out_end;
endcase
end
endtask

//----- 输出启动信号任务 -----
task shift_head;
begin
casex(head_state)
    head_begin:
        if(! SCL)
            begin
                link_write  <= NO;
                link_sda    <= YES;
                link_head   <= YES;
                head_state  <= head_bit;
            end
        else
            head_state  <= head_begin;
    head_bit:
        if(SCL)
            begin
                FF          <= 1;
                head_buf    <= head_buf<<1;
                head_state  <= head_end;
            end
        else
            head_state  <= head_bit;
    head_end:
        if(! SCL)
            begin
                link_head   <= NO;
                link_write  <= YES;
            end
        else

```

```

    head_state <= head_end;
endcase
end
endtask
//----- 输出停止信号任务 -----
task shift_stop;
begin
casex(stop_state)
stop_begin: if(! SCL)
begin
link_sda <= YES;
link_write <= NO;
link_stop <= YES;
stop_state <= stop_bit;
end
else
stop_state <= stop_begin;
stop_bit: if(SCL)
begin
stop_buf <= stop_buf<<1;
stop_state <= stop_end;
end
else
stop_state <= stop_bit;
stop_end: if(! SCL)
begin
link_head <= NO;
link_stop <= NO;
link_sda <= NO;
FF <= 1;
end
else
stop_state <= stop_end;
endcase
end
endtask
endmodule
//----- eeprom_wr.v 文件结束-----

```

eeprom_wr.v 程序模块最终通过 Synplify Pro8.1 和 QuartusII6.1 的综合，并在多种系列的 FPGA 上实现布局布线，通过布线后仿真验证。

(3) EEPROM 的信号源模块和顶层模块：完成串行 EEPROM 读写器件的设计后，我们

还需要做 EEPROM 读写器件的仿真。仿真可以分为前仿真和后仿真,前仿真时 Verilog HDL 的功能仿真,后仿真时 Verilog HDL 代码经过综合、布局布线后的时序仿真。为此,我们还要编写用于 EEPROM 读写器件的仿真测试的信号源程序。这个信号源能产生相应的读信号、写信号、并行地址信号和并行数据信号,并能接收串行 EEPROM 读写器件的应答信号(ACK),以此来调节发送或接收数据的速度。在这个程序中,为了保证串行 EEPROM 读写器件的正确性,可以进行完整的测试。写操作时输入的地址信号和数据信号的数据通过系统命令 \$readmemh 从 addr.dat 和 data.dat 文件中取得,而在 addr.dat 和 data.dat 文件中可以存放任意数据。读操作时从 EEPROM 读出的数据存入文件 eeprom.dat。对比 3 个文件的数据就可以验证程序的正确性。\$readmemh 和 \$fopen 等系统命令读者可以参考 Verilog HDL 的语法部分。最后我们把信号源、EEPROM 和 EEPROM 读写器用顶层模块连接在一起。下面的程序就是这个信号源的 Verilog HDL 模型和顶层模块。

```
/*
模块名称:Signal    文件名:signal.v
模块功能:用于产生测试信号,对所设计的 EEPROM_WR 模块进行测试。Signal 模块能对被测试模块产生的 ack 信号产生响应,发出模仿 MCU 的数据、地址信号和读/写信号;被测试的模块在接收到信号后会发出写/读 EEPROM 虚拟模块的信号。
模块说明:本模块为行为模块,不可综合为门级网表。而且本模块为教学目的做了许多简化,但功能不完整,不能用做商业目的。
*/

```

信号源的 Verilog HDL 模型:

```
'timescale 1ns/1ns
`define timeslice 200
module Signal(RESET,CLK,RD,WR,ADDR,ACK,DATA);
    output RESET;           //复位信号
    output CLK;             //时钟信号
    output RD,WR;           //读写信号
    output[10:0] ADDR;       //11 位地址信号
    input ACK;              //读写周期的应答信号
    inout[7:0] DATA;         //数据线

    reg RESET;
    reg CLK;
    reg RD,WR;
    reg W_R;                //低位:写操作;高位:读操作
    reg[10:0] ADDR;
    reg[7:0] data_to_eeprom;
    reg[10:0] addr_mem[0:255];
    reg[7:0] data_mem[0:255];
    reg[7:0] ROM[1:2047];
    integer i,j;
    integer OUTFILE;
```

```
Parameter test_number=50;  
assign DATA = (W_R) ? 8'bzz : data_to_eeprom ;  
  
//-----时钟信号输入-----  
always #(`timeslice/2)  
    CLK=~CLK; //时钟信号，由复位信号产生，时钟频率为1MHz  
  
//-----读写信号输入-----  
initial  
begin  
    RESET = 1;  
    i = 0;  
    j = 0;  
    W_R = 0;  
    CLK = 0;  
    RD = 0;  
    WR = 0; //将WR置为0，以确保在写操作完成后，再进行读操作  
    #1000; //延时1000ns，等待时序逻辑模块完成复位  
    RESET = 0;  
repeat(test_number) //连续写 test_number 次数据, 调试成功后可以增加到  
//全部地址并覆盖测试  
begin  
    # (5 * `timeslice);  
    WR = 1;  
    #(`timeslice); //将WR置为1，以启动写操作  
    WR = 0; //将WR置为0，以确保在写操作完成后，再进行读操作  
    @ (posedge ACK); //EEPROM_WR 转换模块请求写数据  
end  
#(10 * `timeslice);  
W_R = 1; //开始读操作  
repeat(test_number) //连续读 test_number 次数据  
begin  
    # (5 * `timeslice);  
    RD = 1; //将RD置为1，以启动读操作  
    #(`timeslice); //读取数据  
    RD = 0; //将RD置为0，以确保在读操作完成后，再进行写操作  
    @ (posedge ACK); //EEPROM_WR 转换模块请求读数据  
end  
//-----写操作-----  
initial  
begin  
    $display("writing----writing----writing----writing");
```

```

# (2 * `timeslice);
for(i=0;i<=test_number;i=i+1)
begin
    ADDR = addr_mem[i];           //输出写操作的地址
    data_to_eeprom = data_mem[i]; //输出需要转换的平行数据
    $fdisplay(OUTFILE,"@%0h %0h",ADDR, data_to_eeprom);
    //把输出的地址和数据记录在已经打开的 eeprom.dat 文件中
    @(posedge ACK);             //EEPROM_WR 转换模块请求写数据
end

initial
@(posedge W_R)
begin
    ADDR = addr_mem[0];
    $fclose(OUTFILE);           //关闭已经打开的 eeprom.dat 文件
    $readmemh("./eeprom.dat",ROM); //把数据文件的数据读到 ROM 中
$display("Begin READING----READING----READING----READING");
for(j = 0; j <= test_number; j = j+1)
begin
    ADDR = addr_mem[j];
    @(posedge ACK);
    if(DATA == ROM[ADDR]) //比较并显示发送的数据和接收到的数据是否一致
        $display("DATA %0h == ROM[%0h]--READ RIGHT",DATA,ADDR);
    else
        $display("DATA %0h != ROM[%0h]--READ WRONG",DATA,ADDR);
end
end

initial
begin
    OUTFILE = $fopen("./eeprom.dat"); //打开一个名为 eeprom.dat 的文件备用
    $readmemh("./addr.dat",addr_mem); //把地址数据存入地址存储器
    $readmemh("./data.dat",data_mem); //把准备写入 EEPROM 的数据存入数据存储器
end

endmodule
//----- signal.v 的结束-----

```

顶层模块：

```

/*
 * 本文件是顶层模块，主要完成对 EEPROM 的读写操作。
 */
`include "Signal.v"
`include "EEPROM.v"
`include "./EEPROM_WR.v" //可以用 EEPROM_WR 模块相应的 Verilog 门级网表来替换
`timescale 1ns/1ns
`define timesline 200

module Top;
    wire RESET;
    wire CLK;
    wire RD,WR;
    wire ACK;
    wire[10:0] ADDR;
    wire[7:0] DATA;
    wire SCL;
    wire SDA;

    Parameter test_numbers=123;
    initial #(`timeslice * 180 * test_numbers) $stop;
    Signal #(test_numbers) signal(.RESET(RESET),.CLK(CLK),.RD(RD),
        .WR(WR),.ADDR(ADDR),.ACK(ACK),.DATA(DATA));
    EEPROM_WR eeprom_wr(.RESET(RESET),.SDA(SDA),.SCL(SCL),.ACK(ACK),
        .CLK(CLK),.WR(WR),.RD(RD),.ADDR(ADDR),.DATA(DATA));
    EEPROM eeprom(.sda(SDA),.scl(SCL));

endmodule
//----- top.v 文件的结束-----

```

数据文件：

```

//-----addr.dat 文件的开始-----
00
01
02
03
04
05
06

```

```

..... // 所写的数据个数应该比 Signal.v 代码中 testnumber 多几个
//-----addr.dat 文件的结束-----
//-----data.dat 文件的开始 -----
00
01
02
03
04
05
06
..... // 所写的数据个数应该比 Signal.v 代码中 testnumber 多几个
//-----data.dat 文件的结束 -----

```

通过前后仿真可以验证程序的正确性。这里给出的是 EEPROM 读写时序的前仿真波形，如图 16.7 所示和图 16.8 所示。后仿真波形除 SCL 和 SDA 与 CLK 有些延迟外，信号的逻辑关系与前仿真一致。

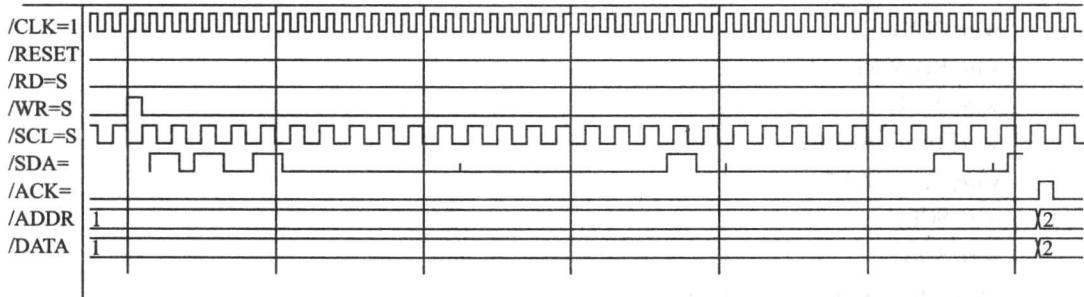


图 16.7 EEPROM 的写时序

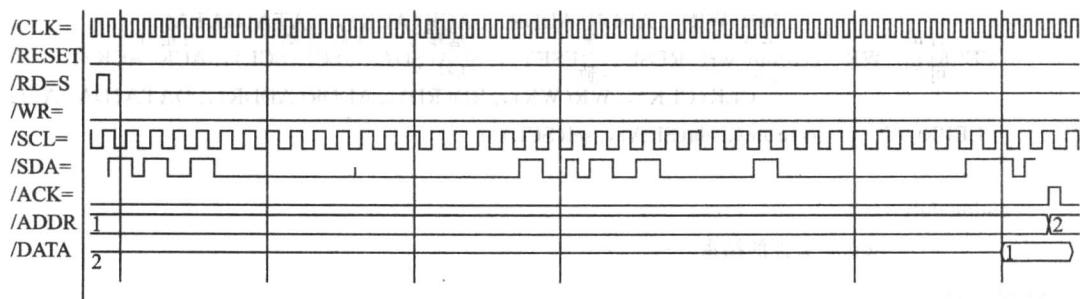


图 16.8 EEPROM 的读时序

说明：在 WINDOWS NT, XP 操作系统的 PC 机器上，应用 Synplify Pro、Actel Designer、Altera MaxplusII、QuartusII 6.1 及 ModelSim 6.1 等 EDA 工具，以上代码已通过 RTL 仿真、综合、布局布线、和布线后仿真验证；也在 Unix 环境下，用 Cadence Verilog-XL, NC-Verilog, Synopsys VCS, DC 等 EDA 工具通过前仿真、综合和后仿真（可综合到各种 FPGA 和 ASIC 工艺）。

总 结

用 Verilog 设计可实现的数字逻辑电路时, 必须对它的电路结构的总体有一个明确的想法。例如上面这个例子, 我们必须对如何控制 sda 串行总线有深入细致的认识。sda 总线既用于输出, 又用于输入, 它必须通过开关网络(组合逻辑电路)与寄存器发生联系。sda 有时与某个寄存器的输出连接, 有时又与另一个寄存器的输入连接。而这些连接过程与 MCU 发过来的命令和数据有关。发出的或接收的信号流又与串行通信协议有关, 这些关系都体现在一个或几个状态机中。状态机是这些开关逻辑正确协调操作的指挥者。为了设计好这些状态机必须要搞清楚信号数据的流动和协议。对接口时序的要求必须明确地在具体电路模块的设计中体现, 也必须在与其连接通信的虚拟模块中明确地用表示行为的 Verilog 语句体现。只有这样, 才能在仿真中帮助我们及时发现复杂状态机编写的漏洞, 从而设计出正确无误的电路系统。

思 考 题

1. 什么是同步状态机?
2. 设计有限同步状态机的一般步骤是什么?
3. 为什么说把具体问题抽象成嵌套的状态机的思考方式可以处理极其复杂的逻辑关系?
4. 为什么要用同步状态机来产生数据流动的开关控制序列?
5. 什么是 HDL RTL 级的描述方式? 它与行为描述方式有什么不同?
6. 什么是综合? 为什么要编写可综合模块?
7. 在设计中可综合模块和行为模块的作用分别是什么?
8. 可综合的 Verilog HDL RTL 级的描述方式的样板是什么?
9. 用 RTL 级描述方式的 Verilog HDL 模块是否都能综合? 保证能综合的要点是什么?
10. 可综合的 Verilog HDL RTL 级模块的编写中, 用阻塞赋值和非阻塞赋值的原则是什么?
11. 保证可综合模块 RTL 和布线后仿真一致性的关键是什么?
12. 读懂本讲中的例子, 如 EEPROM 读写器的设计, 并改写 Verilog 模块, 使得它不只能进行随机读/写还能进行连续的页面读/写, 还改写其他虚拟模块进行 RTL 级、门级网表和布线后仿真。

第 17 章 简化的 RISC_CPU 设计

概 述

在前面 16 章里我们已经学习了 Verilog HDL 的基本语法、简单组合逻辑和简单时序逻辑模块的编写, 学习了 Top-Down 设计方法, 还学习了可综合风格的组合逻辑和有限状态机的设计。其中 EEPROM 读写器的设计实际上是一个较复杂的嵌套的有限状态机的设计, 它是根据业已完成的实际工程项目为教学目的改写而来的, 已很接近真实的设计。在本章里, 将介绍一个经过简化的用于教学目的的精简指令集(RISC)CPU 的构造原理和设计方法。作者相信读者通过参考书上的程序和解释, 经过自己的努力, 就可以独立完成该 CPU 核的设计和验证, 以此来学习这种新设计方法, 并掌握利用 Verilog 硬件描述语言的高层次设计方法。

17.1 课题的来由和设计环境介绍

在本章中, 将通过自己动脑筋, 设计出一个 CPU 的软核和固核。这个 CPU 是一个简化的专门为教学目的而设计的 RISC_CPU。在设计中我们不但关心 CPU 总体设计的合理性, 而且还使得构成这个 RISC_CPU 的每一个模块不仅是可仿真的也都可以综合成门级网表。因而从物理意义上说, 这也是一个能真正通过具体电路结构而实现的 CPU。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, 把寻址空间规定为 8K(即 13 位地址线)字节。

下面让我们一步一步地来设计这样一个 CPU, 并进行 RTL 仿真和综合经过综合、布局布线后, 再进行一次仿真从中可以体会到这种设计方法的潜力。本章中的 Verilog HDL 程序都是为教学目的而编写的, 全部程序在 Cadence 公司的 NC-Verilog 环境 Synopsys VCS、Mentor 公司的 ModelSim 6.1 等环境下用 Verilog 语言进行了仿真。同时分别用 Synplify、Altera Quartus II 等工具, 针对不同的 FPGA 进行了综合。顺利地通过 RTL 级仿真、综合后门级逻辑网表仿真以及布线后的门级结构电路模型仿真。这个 CPU 模型只是一个教学模型, 设计也不一定很合理, 只是从原理上说明了简单的 RISC_CPU 是如何构成的。本章的内容是想达到以下四个目的:

- (1) 学习 RISC_CPU 的基本结构和原理;
- (2) 了解 Verilog HDL 仿真和综合工具的潜力;
- (3) 展示 Verilog 设计方法对软/硬件联合设计和验证的意义;
- (4) 学习并掌握一些常用的 Verilog 语法和验证方法。

作者希望本章的内容能引起对 CPU 和复杂数字逻辑系统设计有兴趣的电子工程师们的注意, 加入我国集成电路的设计队伍, 提高我国电子产品的档次。由于作者的经验与学识有限, 不足之处敬请读者批评、指正。

17.2 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并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:

① 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。

② 分析指令——即指令译码,这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。

③ 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 CPU 的功能进一步细化,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) 能与存储器和外设交换数据;
- (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) 指令寄存器和译码器;
- (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 20 世纪 80 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,而且还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 的不同之处在于: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 Verilog HDL 模型的设计和仿真过程。

17.3 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,可把它分成 8 个基本部件来考虑:

- (1) 时钟发生器；
- (2) 指令寄存器；
- (3) 累加器；
- (4) 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 17.1。其中时钟发生器利用外来时钟信号进行分频生成一

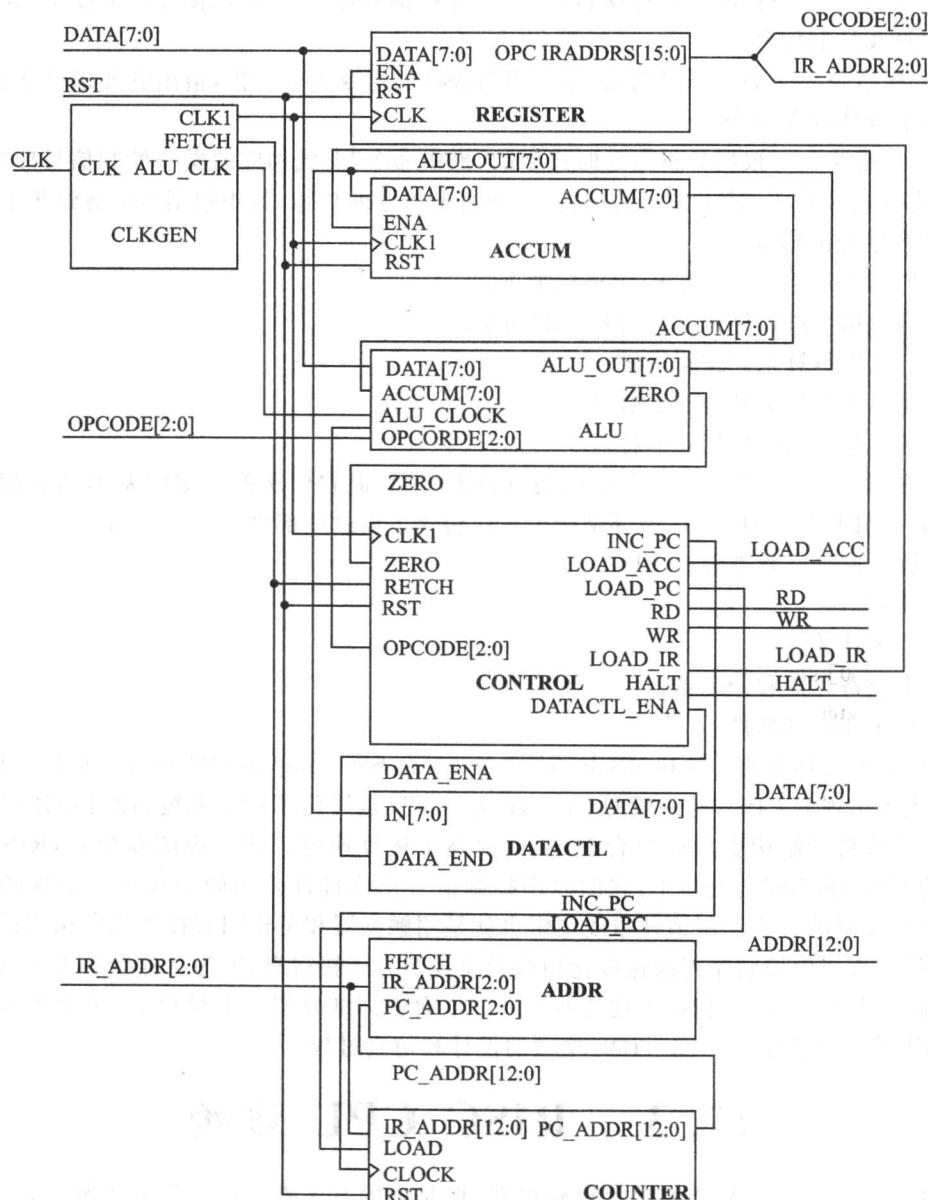


图 17.1 RISC_CPU 中各部件的相互连接关系

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各节中逐一进行介绍。

17.3.1 时钟发生器

时钟发生器 CLKGEN 利用外来时钟信号 clk 生成一系列时钟信号 clk1、fetch、alu_ena，并送往 CPU 的其他部件。其中，fetch 是控制信号，clk 的 8 分频信号。当 FETCH 高电平时，使 CLK 能触发 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。clk 信号用作指令寄存器、累加器、状态控制器的时钟信号。ALU_ENA 则用于控制算术逻辑运算单元的操作。图 17.2 为时钟发生器原理图。时钟发生器 clkgen 的波形如图 17.3 所示。

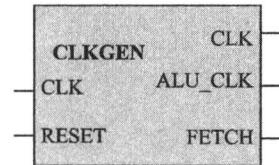


图 17.2 时钟发生器

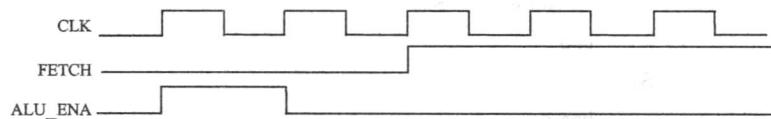


图 17.3 时钟发生器 CLKGEN 的波形

其 VerilogHDL 程序见下面的模块：

```

//-----clk_gen.v 的开始-----
'timescale 1ns/1ns
module clk_gen (clk,reset,fetch,alu_ena);
  input clk,reset;
  output fetch,alu_ena;
  wire clk,reset;
  reg fetch,alu_ena;
  reg[7:0] state;
  parameter S1 = 8'b00000001,
            S2 = 8'b00000010,
            S3 = 8'b000000100,
            S4 = 8'b000001000,
            S5 = 8'b000010000,
            S6 = 8'b001000000,
            S7 = 8'b010000000,
            S8 = 8'b100000000,
            idle = 8'b00000000;

  always @ (posedge clk)
    if(reset)
      begin
        fetch <= 0;
        alu_ena <= 0;
        state <= S1;
      end
    else
      begin
        if(state == S1)
          begin
            fetch <= 1;
            alu_ena <= 1;
            state <= S2;
          end
        else if(state == S2)
          begin
            fetch <= 0;
            alu_ena <= 0;
            state <= S3;
          end
        else if(state == S3)
          begin
            fetch <= 1;
            alu_ena <= 1;
            state <= S4;
          end
        else if(state == S4)
          begin
            fetch <= 0;
            alu_ena <= 0;
            state <= S5;
          end
        else if(state == S5)
          begin
            fetch <= 1;
            alu_ena <= 1;
            state <= S6;
          end
        else if(state == S6)
          begin
            fetch <= 0;
            alu_ena <= 0;
            state <= S7;
          end
        else if(state == S7)
          begin
            fetch <= 1;
            alu_ena <= 1;
            state <= S8;
          end
        else if(state == S8)
          begin
            fetch <= 0;
            alu_ena <= 0;
            state <= S1;
          end
        else
          begin
            fetch <= 0;
            alu_ena <= 0;
            state <= S1;
          end
      end
  end
endmodule

```

```

state<=idle;
end
else
begin
    case(state)
        S1:
        begin
            alu_ena<=1;
            state <= S2;
        end
        S2:
        begin
            alu_ena<=0;
            state <= S3;
        end
        S3:
        begin
            fetch<=1;
            state <= S4;
        end
        S4:
        begin
            state <= S5;
        end
        S5: state <= S6;
        S6: state <= S7;
        S7: begin
            fetch<=0;
            state <= S8;
        end
        S8: begin
            state <= S1;
        end
        idle: state <= S1;
        default: state <= idle;
    endcase
end
endmodule
-----clk_gen.v 的结束-----

```

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 fetch,alu_ena 在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。

17.3.2 指令寄存器

顾名思义,指令寄存器用于寄存指令,如图 17.4 所示。

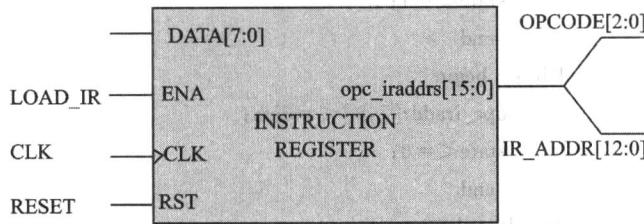


图 17.4 指令寄存器结构

指令寄存器的触发时钟是 clk,在 clk 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 clk 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。

每条指令为两个字节,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 0 表示取的是高 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:

```

`timescale 1ns/1ns
module register(opc_iraddr,data,ena,clk,rst);
    output [15:0] opc_iraddr;
    input [7:0] data;
    input ena, clk, rst;
    reg [15:0] opc_iraddr;
    reg state;

    always @ (posedge clk)
    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
                        ...
                    end
            end
    end
endmodule

```

```

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'bxxxxxxxxxxxxxx;
        state<=1'b0;
    end
endcase
end
else begin
    state<=1'b0;
end
end
endmodule
//-

```

17.3.3 累加器

累加器用于存放当前的结果,它也是双目运算中的一个数据来源(见图 17.5)。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。

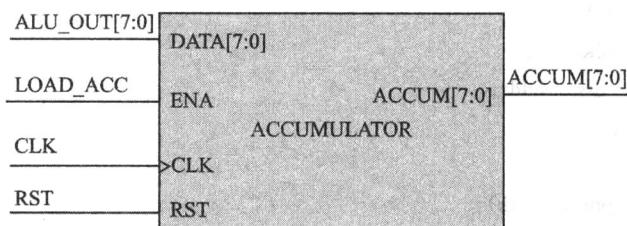


图 17.5 累加器结构

其 VerilogHDL 程序见下面的模块:

```

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

```

```

always@(posedge clk)
begin
    if(rst)
        accum<=8'b0000_0000;           //Reset
    else
        if(ena)                      //CPU 状态控制器发出 load_acc 信号
            accum<=data;             //Accumulate
end

endmodule

```

17.3.4 算术运算器

算术逻辑运算单元如图 17.6 所示。它根据输入的 8 种不同操作码分别实现相应的加、与、异或、跳转等基本操作运算。利用这几种基本运算可以实现很多种其他运算以及逻辑判断等操作。

其 Verilog HDL 程序见下面的模块：

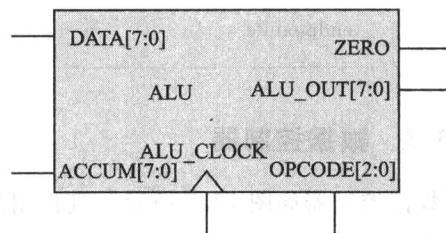


图 17.6 算术运算器结构

```

'timescale 1ns/1ns
module alu (alu_out, zero, data, accum, alu_ena, opcode);
    output [7:0]alu_out;
    output zero;
    input [7:0] data, accum;
    input [2:0] opcode;
    input alu_ena;
    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 @ (posedge clk)
        if(alu_ena)

```

```

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
//-----

```

17.3.5 数据控制器

数据控制器如图 17.7 所示。其作用是控制累加器的数据输出,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,以允许其他部件使用数据总线。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。

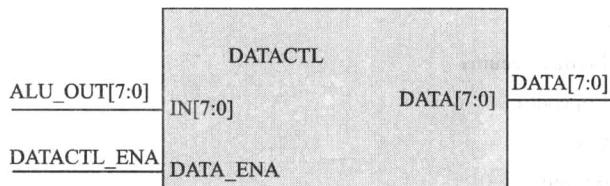


图 17.7 数据控制器结构

其 Verilog HDL 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
//-----

```

17.3.6 地址多路器

地址多路器如图 17.8 所示。它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。

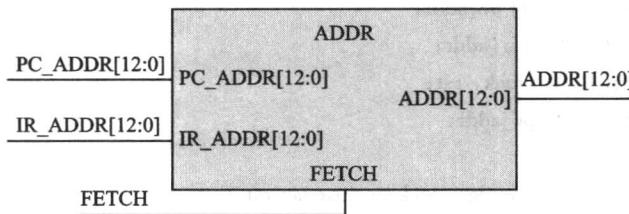


图 17.8 地址多路器结构

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
//-----

```

17.3.7 程序计数器

程序计数器如图 17.9 所示。它用于提供指令地址, 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。

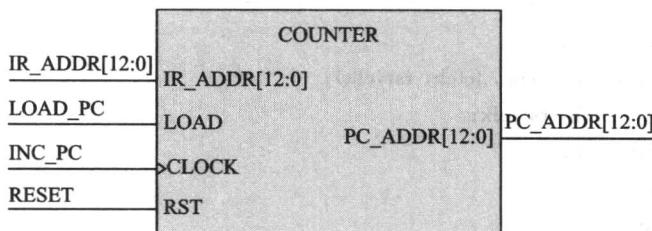


图 17.9 程序计数器结构

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。

每条指令执行完需两个时钟,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)。如果正在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 Verilog HDL 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  
//-----
```

17.3.8 状态控制器

状态控制器如图 17.10 所示。它由两部分组成:

- (1) 状态机(图中的 MACHINE 部分);
- (2) 状态控制器(图中的 MACHINECTL 部分)。

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

状态控制器的 Verilog HDL 程序见下面模块:

```
//-----  
'timescale 1ns/1ns  
module machinectl( ena, fetch, rst, clk );  
    input fetch, rst, clk;  
    output ena;  
    reg ena;  
    reg state;  
  
    always @ (posedge clk)  
    begin
```

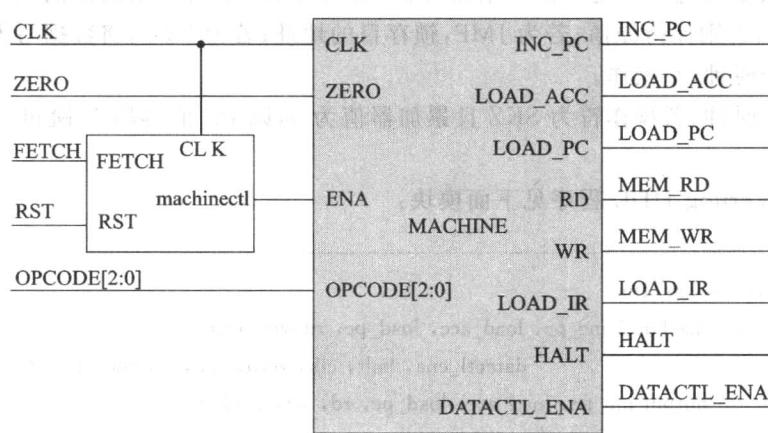


图 17.10 状态控制器

```

if(rst)
begin
    ena<=0;
end
else
begin
    if(fetch)
begin
    ena<=1;
end
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 何时进行读指令来读写 I/O 端口及 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,即

(1) 第 0 个时钟,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),其他各控制线输出为零。

(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 Verilog HDL 程序见下面模块:

```
//-----  
'timescale 1ns/1ns  
  
module machine( inc_pc, load_acc, load_pc, rd, wr, load_ir,  
                datactl_ena, halt, clk, zero, ena, opcode );  
    output inc_pc, load_acc, load_pc, rd, wr, load_ir;  
    output datactl_ena, halt;  
    input clk, 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 clk)  
        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-----  
    taskctl_cycle;  
        begin  
            casex(state)  
                3'b000:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b001:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b010:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b011:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b100:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b101:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b110:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;  
                3'b111:  
                    inc_pc = 1'b1;  
                    load_pc = 1'b1;  
                    load_ir = 1'b1;  
                    datactl_ena = 1'b1;  
                    halt = 1'b1;
```

```
3'b000;                                //load high 8bits instruction
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 operand
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
        end
      end
    end
  end
end

```

```
        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: //zero
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 17.10。

17.3.9 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 @(*)
    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。

1FFFH —— 1800H RAM

17FFH —— 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'hffff:0];
    wire [7:0] data;

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

endmodule

```

ROM 用于装载测试程序, 可读不可写; 而 RAM 用于存放数据, 可读可写。

17.4 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。

下面详细介绍每个操作, 即系统的复位与启动, 总线的读与写等操作。

17.4.1 系统的复位和启动操作

RISC_CPU 的复位和启动操作是通过 reset 引脚的信号触发执行的。当 reset 信号一进入高电平, RISC_CPU 就会结束现行操作, 并且只要 reset 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。reset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。波形见图17.11,虚线标志处为RISC_CPU启动工作的时刻。

图17.11为RISC_CPU的复位和启动操作波形图,虚线标志处为RISC_CPD启动工作的时刻。

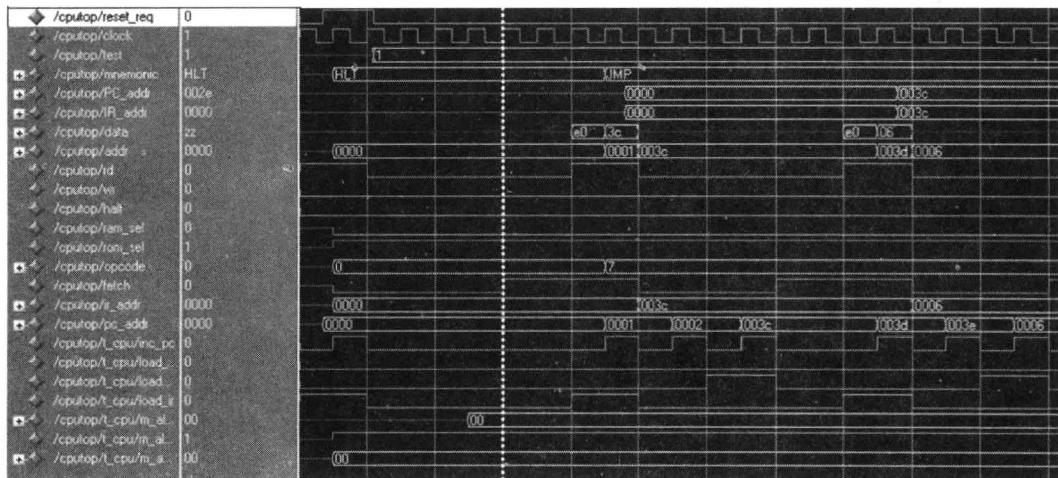


图 17.11 RISC_CPU 的复位和启动操作波形

17.4.2 总线读操作

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个时钟周期,地址总线输出PC地址,为下一个指令做好准备。图17.12为CPU从存储器或端口读取数据的时序。

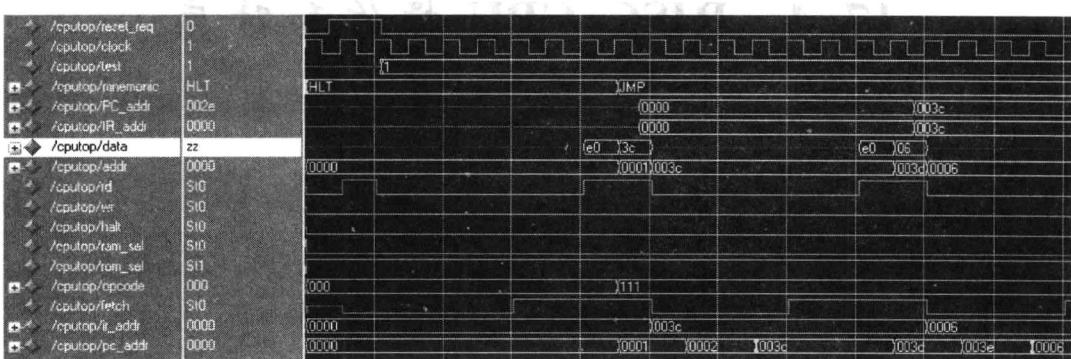


图 17.12 CPU 从存储器或端口读取数据的时序

17.4.3 总线写操作

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 个时钟地址输出为 PC 地址,为下一个指令周期做好准备。图 17.13 为 CPU 对存储器或端口写数据的时序。

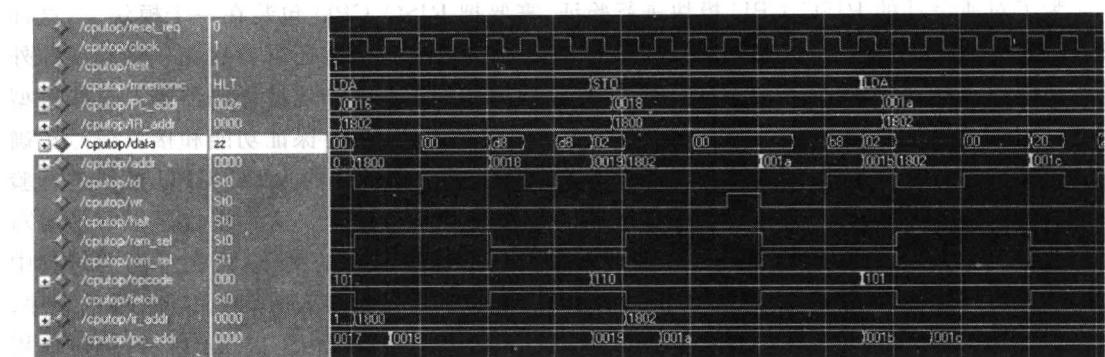
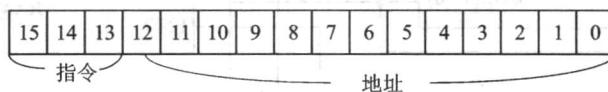


图 17.13 CPU 对存储器或端口写数据的时序

17.5 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。

17.6 RISC_CPU 模块的调试

17.6.1 RISC_CPU 模块的前仿真

为了对所设计的 RISC_CPU 模块进行验证,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 17.14。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 Verilog HDL 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 模块进行验证,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 17.3.9 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 模块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,如图 17.14 所示;也可以用 Verilog HDL 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。

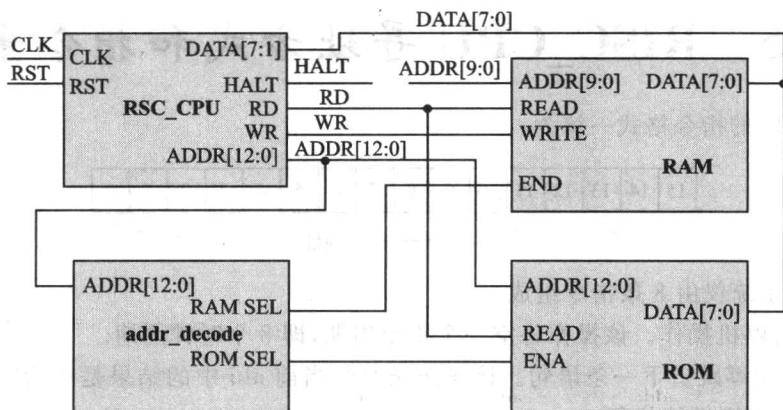


图 17.14 RISC_CPU 及其外围电路

下面介绍的是在 modelsim 6.1 下进行调试的仿真测试程序 cputop.v。可用于对以上所设计的 RISC_CPU 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,可以同时用波形观察器观察有关信号的波形。

```
//----- cputop.v 文件的开始-----
/*
*** 模块功能:cputop模块通过运行3个不同的汇编程序对cpu模块进行完整的逻辑测试
*** 和验证。仿真程序在加载后会自动执行,运行的结果在人机交互式显示屏上
*** 显示。本模块是不可综合的行为模块,用于对CPU模块进行RTL级和门级
*** 逻辑功能的全面仿真验证。本测试模块也可以为布线后仿真,只需要将代码
```

```

***          的第一句 `include "cpu.v" 改写为 `include "cpu.vo" 即可。
*****//若改为 `include "cpu.vo" 便可做布线后仿真

`include "cpu.v"
`include "ram.v"
`include "rom.v"
`include "addr_decode.v"
*****`timescale 1ns / 1 ns
`define PERIOD 100 // matches clk_gen.v
module cputop;
    reg reset_req, clock;
    integer test;
    reg [(3 * 8):0] mnemonic; //array that holds 3 8bit 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;
    wire [2:0] opcode; //为布线后仿真做的专门添加的 CPU 内部信号线
    wire fetch; //为布线后仿真做的专门添加的 CPU 内部信号线
    wire [12:0] ir_addr, pc_addr; //为布线后仿真做的专门添加的 CPU 内部信号线

//----- CPU 模块与地址译码器和 ROM, RAM 的连接部分-----
cpu t_cpu (.clk(clock), .reset(reset_req), .halt(halt), .rd(rd),
           .wr(wr), .addr(addr), .data(data), .opcode(opcode), .fetch(fetch),
           .ir_addr(ir_addr), .pc_addr(pc_addr);

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));

//----- CPU 模块与地址译码器和 ROM, RAM 的连接部分结束-----
initial
begin
    clock=1;
    //display time in nanoseconds
    $timeformat(-9, 1, "ns", 12);
    display_debug_message;
    sys_reset;
    test1;
    $stop;
    test2;

```

```

$ stop;
test3;
$ finish; //simulation is finished here.
end
task display_debug_message;
begin
$ display("\n*****");
$ display(" * THE FOLLOWING DEBUG TASK ARE AVAILABLE: * ");
$ display(" * \\"test1; \" to load the 1st diagnostic program. * ");
$ 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. pc_addr)//fixed
                if ((t_cpu. pc_addr%2 == 1)&&(t_cpu. fetch == 1))//fixed
                    begin
                        # 60  PC_addr <= t_cpu. pc_addr - 1 ;
                        IR_addr <= t_cpu. 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 ***");
        end
    endcase
end
endtask
```

```

$ display("\n      TIME      PC      INSTR      ADDR      DATA      ");
$ display("("----- ----- ----- ----- ----- ----- -----");
while (test == 2)
  @ (t_cpu.pc_addr)
  if ((t_cpu.pc_addr%2 == 1) && (t_cpu.fetch == 1))
    begin
      # 60    PC_addr <= t_cpu.pc_addr - 1 ;
      IR_addr <= t_cpu.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.opcode == 3'h1) //display Fib. No. at end of program loop
      $ strobe("%t      %d", $time, t_ram.ram[10'h2]);
      wait ( t_cpu.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.opcode)
  //get an ASCII mnemonic for each opcode
  case(t_cpu.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
//----- cputop.v 文件的结束 -----

```

针对程序做如下说明：测试程序中用宏指令 `include“模块文件”名包含了 rom.v , ram.v 和 addrdecode.v3 3 个外部模块。它们都是检测 RISC_CPU 时必不可少的虚拟部件却代表了 RAM,ROM 和地址译码器；对于 RISC_CPU 需要综合成电路的部分，则已通过 CPU.v 程序将它组合成一个独立的 CPU 模块。具体程序如下：

```

//----- cpu.v 文件的开始 -----
/* **** */
模块功能:CPU 模块是本讲所设计的 RISC_CPU 的核心，共有 10 个可综合模块的
连接组成。它本身是可综合的模块，已经过门级后仿真验证。
***** */

`include "clk_gen.v"
`include "accum.v"
`include "adr.v"
`include "alu.v"
`include "machine.v"
`include "counter.v"
`include "machinectl.v"
`include "register.v"
`include "datactl.v"
// ****

`timescale 1ns/1ns
module cpu(clk,reset,halt,rd,wr,addr,data,opcode,fetch,ir_addr,pc_addr);
    input clk,reset;
    output rd,wr,halt;
    output[12:0]addr;
    output[2:0]opcode;
    output fetch;
    output[12:0]ir_addr,pc_addr;
    inout[7:0]data;
    wire clk,reset,halt;
    wire [7:0] data;
    wire [12:0] addr;

```

```

wire rd,wr;
wire clk,fetch,alu_ena;
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),.reset(rest),.fetch(fetch),.alu_ena(alu_ena));

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

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

alu m_alu (.data(data),.accum(accum),.clk(clk);alu_ena(aln_ena)),
            .opcode(opcode),.alu_out(alu_out),.zero(zero));

machinectl m_machinectl(clk(clk),.rst(reset),.fetch(fetck),.ena(control_ena));
machine m_machine (.inc_pc(inc_pc),.load_acc(load_acc),.load_pc(load_pc),
                   .rd(rd),.wr(wr),.load_ir(load_ir),.clk(clk),
                   .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 (.clock(inc_pc),.rst(reset),.ir_addr(ir_addr),.load(load_pc)),
                  .pc_addr(pc_addr));

endmodule
//-----cpu.v文件的结束-----

```

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

```

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

```

即可把编译好的汇编机器码装入虚拟 ROM, 把需要参加运算的数据装入虚拟 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。

下面清单所列出的是用于测试 RISC_CPU 基本功能而分别装入虚拟 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.dat,还有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 逻辑功能的机器代码,是根据汇编语言由人工编译的。
*** 本汇编程序用于测试 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。
*** @符号后的十六进制数表示存储器的地址,以下的二进制数为机器码。
*** 每行//符号后表示自己为 RISC_CPU 设计的汇编程序和程序注释。
****

-----test1.pro 的开始-----
机器码 地址 汇编助记符 注释
@00
//address statement
111_00000 // 00 BEGIN: JMP TST_JMP
0011_1100 // 01
000_00000 // 02 HLT //JMP did not work at all
0000_0000 // 03
000_00000 // 04 HLT //JMP did not load PC, it skipped
0000_0000 // 05
101_11000 // 06 JMP_OK: LDA DATA_1
0000_0000
001_00000 // 08 SKZ
0000_0000 // 09
000_00000 // 0a HLT //SKZ or LDA did not work
0000_0000 // 0b
101_11000 // 0c LDA DATA_2
0000_0001
001_00000 // 0d SKZ
0000_0000 // 0e
111_00000 // 0f JMP SKZ_OK
0001_0100 // 10
000_00000 // 11 HLT //SKZ or LDA did not work
0000_0000 // 12
110_11000 // 13 SKZ_OK: STO TEMP //store non-zero value in TEMP
0000_0010 // 14
101_11000 // 15 LDA DATA_1
0000_0000 // 16
110_11000 // 17 SKZ_OK: STO TEMP //store zero value in TEMP
0000_0010 // 18
```

```

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。

//-----test2.pro 开始-----

机器码	地 址	汇编助记符	注 释
@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:           //constant 0(hex)
//-----test2. dat 结束 .
```

```

/ ****
* Test 3 程序是一个计算从 0~144 的 Fibonacci 序列的程序, 用于验证 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
@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	VLDA 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:          //temporaray 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 modelsim6.1 的使用。

(1) 建一个目录存放编写的设计代码文件, 注意所有 Verilog 设计代码都必须用扩展名为.v 的文本型文件存档(注意: 在计算机环境中将.v 扩展名文件的打开方式设置为 Modelsim)。

(2) 双击 cputop.v 便能自动地启动 modelsim 6.1 进入文件所在的目录。

(3) 单击 modelsim 6.1 台头菜单: File→New→Library 弹出一个配置框, 给新的 Library 命名, 例如键入 work, 单击 OK, 在 Transcript 框内出现一些文字告诉操作者所在的目录和处理的文件和新建立的 Library 的名字, 这次编译的很多信息将储存在这个由用户起名的 Library 中。

(4) 单击 modelsim 6.1 台头菜单上类似多层文件的图标, 弹出配置框, 可设置 Library, 并可以把需要编译的文件选中进行编译。

(5) 编译成功后, 用户将在 workspace 的框内发现用户命名的 Library 下面出现了编译通过的文件名, 如果设计仿真所需要的所有文件已经编译成功, 就可以加载仿真代码。只需要双击 cputop 即可, 如果加载成功, Modesim 自动地进入可以仿真的状态, 只要配置好需要观察波形的信号, 就可以单击台头菜单上的开始仿真的图标。至于如何配置需要观察的信号, 不同的版本有些差别, 试验几次就可以明白, 这里就不再一一赘述。由于 cputop.v 编写了很好的输

出显示,所以在 Transcript 框内将显示以下信息,这些信息说明仿真工作正确无误。

仿真结果如下:

```
run-all
#
# *****
# * THE FOLLOWING DEBUG TASK ARE AVAILABLE:
# * "test1;" to load the 1st diagnostic program.
# * "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,RTL 仿真(即布局布线前的仿真)可告结束。

17.6.2 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。

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块(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 CPUC.v 模块。见前面测试程序解释时介绍的 CPU.v 模块。如用电路图描述的话,还需给它的引脚加上标准的引脚部件并加标记,如图 17.15 所示。

第三阶段:把需要的综合的模块加载到综合器,本例所使用的综合器是独立的 Synplify 8.1,选定的 FPGA 是 Altera Stratixii,针对它的库进行综合。

也可以使用 QuartusII 或其他综合工具进行综合。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。下面的报告就是综合 cpu.v 时生成的综合报告,综合所用的库为 Altera Stratixii 系列的 FPGA 库,约定的时钟频率为 80 MHz。

```

//----- RISC_CPU 芯片综合结果报告开始-----
# Program: Synplify Pro 8.1
# OS: Windows_NT

```

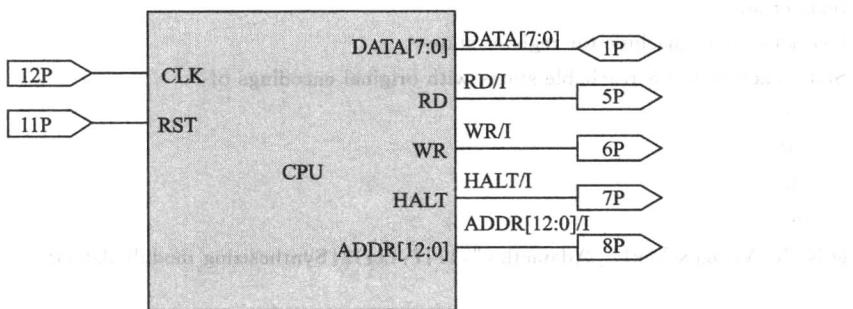


图 17.15 用于综合的 RISC_CPU 模块 (CPU.v)

```

$ Start of Compile
# Wed May 23 17:02:14 2007
Synplicity Verilog Compiler, version 3.1.0, Build 049R, built May 3 2005
Copyright (C) 1994–2005, Synplicity Inc. All Rights Reserved

@I: : "C:\Program Files\Synplicity\fpga_81\lib\altera\altera.v"
@I: : "C:\Program Files\Synplicity\fpga_81\lib\altera\altera_mf.v"
@I: : "C:\Program Files\Synplicity\fpga_81\lib\altera\altera_lpm.v"
@I: : "C:\vlogexe\ex17_2\cpu.v"
@I: "C:\vlogexe\ex17_2\cpu.v": "C:\vlogexe\ex17_2\clk_gen.v"
@I: "C:\vlogexe\ex17_2\cpu.v": "C:\vlogexe\ex17_2\accum.v"
.....
.....
Verilog syntax check successful!
Compiler output is up to date. No re-compile necessary
Selecting top level module cpu
@N: "C:\vlogexe\ex17_2\clk_gen.v":2:7:2:13|Synthesizing module clk_gen
@N: CL201 :"C:\vlogexe\ex17_2\clk_gen.v":18:0:18:5 | Trying to extract state machine for
register state
Extracted state machine for register state
State machine has 9 reachable states with original encodings of:
00000000
00000001
00000010
.....
.....
@N: "C:\vlogexe\ex17_2\register.v":4:7:4:14|Synthesizing module register
.....
.....
@N: CL201 :"C:\vlogexe\ex17_2\machine.v":44:0:44:5 | Trying to extract state machine for

```

register state

Extracted state machine for register state

State machine has 8 reachable states with original encodings of:

000

001

010

...

@N: "C:\vlogexe\ex17_2\datactl.v":11:7;11:13|Synthesizing module datactl

@N: "C:\vlogexe\ex17_2\adr.v":10:8;10:10|Synthesizing module adr

.....

.....

@END

Process took 0h:00m:01s realtime, 0h:00m:01s cputime

Wed May 23 17:02:14 2007

#####
[

Version 8.1

Synplicity Altera Technology Mapper, Version 8.1.0, Build 539R, Built May 6 2005

Copyright (C) 1994–2005, Synplicity Inc. All Rights Reserved

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)

.....

.....

RTL optimization done.

@N: "c:\vlogexe\ex17_2\counter.v":19:0;19:5|Found counter in view:work.cpu(verilog) inst m_counter.pc_addr[12:0]

Encoding state machine work.clk_gen(verilog)–state[8:0]

original code → new code

00000000 → 000000000

00000001 → 000000011

.....

Encoding state machine work.machine_synplcty(verilog)–state[7:0]

original code → new code

000 → 00000000

001 → 00000011

.....

Writing Analyst data base C:\vlogexe\ex17_2\rev_1\cpu.srm

Writing Verilog Netlist and constraint files

Writing .vqm output for Quartus

Writing Cross reference file for Quartus to C:\vlogexe\ex17_2\rev_1\cpu.xrf

Writing Verilog Simulation files

Found clock cpu|clk with period 12.50ns

```

Found clock machine|inc_pc_derived_clock with period 12.50ns

##### START OF TIMING REPORT #####
# Timing Report written on Wed May 23 17:02:16 2007

Top view:          cpu
Requested Frequency: 80.0 MHz
Wire load mode:    top
Paths requested:   5
Constraint File(s): @N: MT195 | This timing report estimates place and route data. Please look at the place and route timing report for final timing.

@N: MT197 | Clock constraints cover only FF-to-FF paths associated with the clock.

Performance Summary
*****
Worst slack in design: 10.158



| Starting Clock               | Requested Frequency | Estimated Frequency | Requested Period | Estimate Period | Slack  | Clock Type | Clock Group         |
|------------------------------|---------------------|---------------------|------------------|-----------------|--------|------------|---------------------|
| cpu clk                      | 80.0 MHz            | 427.0 MHz           | 12.500           | 2.342           | 10.158 | inferred   | Inferred_clkgroup_0 |
| machine inc_pc_derived_clock | 80.0 MHz            | 427.0 MHz           | 12.500           | 2.342           | 10.661 | derived    | Inferred_clkgroup_0 |



Clock Relationships
*****
Detailed Report for Clock: cpu|clk

Starting Points with Worst Slack
*****


| Instance         | Starting Reference Clock | Type               | Pin    | Net     | Arrival Time | Slack  |
|------------------|--------------------------|--------------------|--------|---------|--------------|--------|
| m_accum.accum[0] | cpu clk                  | stratixii_lcell_ff | regout | accum_0 | 0.095        | 10.158 |
| m_accum.accum[1] | cpu clk                  | stratixii_lcell_ff | regout | accum_1 | 0.095        | 10.194 |


Ending Points with Worst Slack

```

***** Worst Path Information *****

Worst Path Information

Path information for path number 1:

Requested Period:	12.500
— Setup time:	0.403
= Required time:	12.097
— Propagation time:	1.939
= Slack (critical) :	10.158

Number of logic level(s):

9

Starting point: m_accum. accum[0] / regout

Ending point: m_alu. alu_out[7] / adatasdata

The start point is clocked by cpu|clk [rising] on pin clk

The end point is clocked by cpu|clk [rising] on pin clk

Instance/Net	Type	Pin	Pin	Arrival	No. of
Name		Name	Dir	Delay	Fan Out(s)
m_accum. accum[0]	stratixii_lcell_ff	regout	Out	0.095	0.095
accum_0	Net	—	—	0.621	—
m_alu. un2_alu_out_carry_0	stratixii_lcell_comb	dataf	In	—	0.716
m_alu. un2_alu_out_carry_0	stratixii_lcell_comb	cout	Out	0.312	1.028
un2_alu_out_carry_0	Net	—	—	0.000	—
m_alu. un2_alu_out_carry_1	stratixii_lcell_comb	cin	In	—	1.028
.....					
.....					

Total path delay (propagation time + setup) of 2.342 is 1.307(55.8%) logic and 1.035(44.2%) route.

Detailed Report for Clock: machine_synthpcty|inc_pc_derived_clock

Starting Points with Worst Slack

Starting	Reference	Type	Pin	Net	Arrival Time	Slack
Instance	Clock					
m_counter. pc_addr[0]	inc_pc_clock	stratixii_lcell_ff	regout	pc_addr_c_0	0.095	10.661
m_counter. pc_addr[1]	inc_pc_clock	stratixii_lcell_ff	regout	pc_addr_c_1	0.095	10.697
.....						

Ending Points with Worst Slack

Path information for path number 1:

Requested Period:	12.500
— Setup time:	0.247
= Required time:	12.253
 — Propagation time:	1.592
= Slack (non-critical) :	10.661

Number of logic level(s): 13

Starting point: m_counter.pc_addr[0] / regout

Ending point: m_counter.pc_addr[12] / datain

The start point is clocked by machine_synpcty|inc_pc_derived_clock [rising] on pin clk

The end point is clocked by machine_synpcty|inc_pc_derived_clock [rising] on pin clk

Instance / Net		Pin	Pin	Arrival	No. of	
Name	Type	Name	Dir	Delay	Time	Fan Out(s)
m_counter.pc_addr[0]	stratixii_lcell_ff	regout	Out	0.095	0.095	—
pc_addr_c_0	Net	—	—	—	0.621	— 3
m_counter.pc_addr_c0	stratixii_lcell_comb	datad	In	—	0.716	—
m_counter.pc_addr_c0	stratixii_lcell_comb	cout	Out	0.354	1.070	—
pc_addr_c0_cout	Net	—	—	—	0.000	— 1

Total path delay (propagation time + setup) of 1.839 is 1.218(66.2%) logic and 0.621(33.8%) route.

#####END OF TIMING REPORT #####

#####START OF AREA REPORT #####

Design view: work.cpu(verilog)

Selecting part EP2S15F484C3

@N: FA174 | The following device usage report estimates place and route data. Please look at the place and route report for final resource usage.

Total combinational functions 76

ALUT usage by number of inputs

7 input functions 0

6 input functions 8

5 input functions 7

4 input functions 6

```

<=3 input functions 55
ALUTs by mode
    normal mode      55
    extended LUT mode 0

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-----

在以上的报告文件中,揭示了综合器对综合过程和结果的分析,有极其重要的意义。它能帮助设计者了解系统运行的最高时钟、关键路径的最大延迟,使用的逻辑部件的种类和数目。当出现问题时会提示人们:Verilog 源代码的哪个模块第几行有错误或警告。这些资料的熟练应用和分析是很重要的。它能提高设计的工作效率,使综合器综合出更加合理的电路。关于如何利用综合器能处理的综合指令,作者将在高级教程中介绍。综合指令和属性是 Verilog 源代码中的一种符合特殊规定的注释行,仿真工具不处理这样的注释行,而综合器却能识别这些符合特殊要求的注释行,根据设计者通过综合指令提出的要求,使综合器综合出更好的符合设计者要求的电路结构。

17.6.3 RISC_CPU 模块的优化和布局布线

选定元件库后就可以对所设计的 RISC_CPU 模型进行综合,综合工作是把 Verilog RTL 代码通过综合工具,产生一系列由现存元件的逻辑网表组成的文件。在综合工具上通过选择项可以配置生成逻辑网表文件的格式。逻辑网表文件可以是:Verilog Netlist、VHDL Netlist 或者电子设计交换格式(Electronic Design Interchange Format),也就是在电路设计工业界常说的 EDIF 格式文件。在产生了这些文件之后,就可以进行综合后的网表仿真。网表仿真的 Verilog 模型只是对应库逻辑元件的行为模型,并不涉及器件和布局布线的连接线延迟,因此与实际电路的行为还存在着差异,这种仿真模型没有明显的延迟。为了知道实现电路真实的带延迟的行为,还必须进行布局布线操作,以便生成实际电路和连接线带延迟的行为模型。

下面将介绍如何用 Altera QuartusII 进行综合和布局布线,由 RTL 代码产生由对应元件库(Altera StritixII)Verilog 网表组成的仿真模型以及该网表所提取的延迟参数文件。用 QuartusII 进行综合和布线布线的步骤如下:

- (1) 双击 QuartusII 图标,启动 QuartusII 工具。
- (2) 在 QuartusII 主窗口的台头工具栏中选择 File→New Project Wizard...,随即弹出对话框,在相应的空格栏选取或者填入工作目录名、项目名和被综合模块组的顶层模块名。
- (3) 单击 Next,随即弹出另外一个对话框,在相应的空格栏中选取或者填入希望被综合的文件名,单击 ADD,添加该文件进入综合环境;
- (4) 单击 Next,随即又弹出一个对话框,在相应的空格栏中选取或者填入实现逻辑的器

件型号。

(5) 单击 Next, 随即又弹出一个对话框, 选取 EDA 仿真工具, 在出现的空格框内选取 ModelSim 和 Verilog 格式。

(6) 单击 Next, 仔细阅读设计者已经配置环境的情况, 然后再单击 finish, 结束综合环境的配置过程。

(7) 在 QuartusII 主窗口的台头工具栏中选择 Processing → start compilation, 或者直接单击三角形的图标随即开始编译过程。

(8) 在工作目录中会出现一个新的名为 simulation 的目录打开这个目录, 可以看到一个名为 Modelsim 的目录, 再打开这个目录, 可以看到 3 个文件, 分别为 xxx.v, xxx_modelsim.xrf 和 xxx_v.sdo。

(9) 将 xxx.v 和 xxx_v.sdo 复制到工作目录, 将 xxx.v 文件替换原来的 xxx.v 文件再进行一次仿真就能将 xxx_v.sdo 的延迟信息带入, 得到布局布线后的仿真结果。

(10) 需要注意的是布局布线后的仿真还必须有已经选择库的仿真模型才能进行。这些库究竟在哪里呢? 我们可以在 Altera QuartusII 的安装目录里寻找。如果 Altera QuartusII 安装在硬盘 C 上, 则在 C:\altera\61\quartus\eda\sim_lib 目录下可以看到许多型号 FPGA 的元件库的仿真模型, 如果用的是 Verilog 模型, 只需要在扩展名为.v 的文件中寻找即可, 把相应型号的 FPGA 库元件的仿真模型复制到自己的工作目录进行编译就能进行布局布线后的仿真了。

综合和布局布线完成后得到两个文件 cpu.v, cpu_v.sdo 和 cpu_modelsim.xrf。cpu.v 是所设计的 RISC_CPU 的门级结构, 即利用 Verilog 语法描述的用 stratixii 型号 FPGA 库中的基本逻辑电路元件构成的复杂电路连线网络, 而 cpu_v.sdo 是布局布线的延迟参数文件, stratixii_atoms.v 是 cpu.v 所引用的 Verilog 门级模型的库文件, 包含了各种基本逻辑电路的门级模型, 它们的参数与真实器件完全一致, 包括如延迟等参数。

需要注意的是: 必须在布线工具的相关界面上选取生成输出文件的格式为 Verilog 后, cpu.v 和 cpu_v.sdo 这两个文件才会产生。将这两个文件和 stratixii_atoms.v 包含在 cpu_top.v 中, 来代替原来的 RTL 模块 cpu.v。其他外围测试行为模块相同, 用仿真器再进行一次仿真, 此时称为布局布线后仿真。实际上, 后仿真与前仿真的根本区别在于测试文件所包含模型的结构不同。前仿真使用的是 RTL 级模型, 如 cpu.v, 而后仿真使用的是真实的门级结构模型, 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有一些不同, 各信号的变化与时钟沿之间存在着延迟, 这些仿真信息在前仿真时并未反映出来, 后仿真波形如图 17.16 所示。

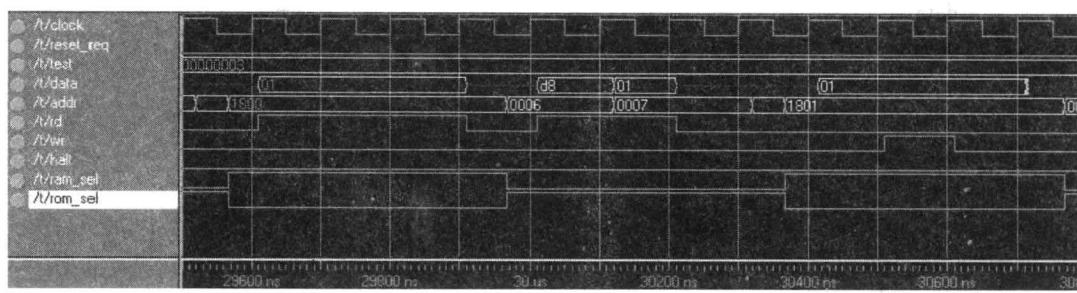


图 17.16 后仿真波形

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 cpu_v.sdo。由于 cpu.vo 是门级描述, 共有上千行, 而 cpu_v.sdo 是延迟参数文件, 也有几百行。而 stratixii_atoms.v 是库元件, 包含的逻辑元件非常多, 也无法在课本上列出其全部程序, 只能从中截取一小片段供同学参考。有兴趣的同学可以查看生成的代码并参考 Verilog 语法手册中有关门级描述和用户自定义源语(UDP)来理解这些代码。由于这些代码是 Verilog 的门级模型, 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。下面列出了这三个可用于布局布线后仿真的, 用来代替 RTL 描述的 cpu.v 的 Verilog 文件片段供同学参考, 帮助同学理解布线后门级仿真的原理。

```

/ **** * cpu.vo 开始 **** *
// Copyright (C) 1991 - 2006 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
-----
-----
// VENDOR "Altera"
// PROGRAM "Quartus II"
// VERSION "Version 6.1 Build 201 11/27/2006 SJ Full Version"

// DATE "05/20/2007 14:15:10"

// Device: Altera EP2S15F484C3 Package FBGA484

// This Verilog file should be used for ModelSim (Verilog) only

`timescale 1 ps/ 1 ps

module cpu (
    clk,
    reset,
    halt,
    rd,
    wr,
    addr,
    data,
    opcode,
    fetch,
    ir_addr,
    pc_addr);
    input    clk;
    input    reset;
    output   halt;
    output   rd;
    output   wr;
    output  [12:0] addr;

```

```
inout [7:0] data;
output [2:0] opcode;
output fetch;
output [12:0] ir_addr;
output [12:0] pc_addr;

wire gnd = 1'b0;
wire vcc = 1'b1;

tril devclr;
tril devpor;
tril devoe;

// synopsys translate_off
initial $ sdf_annotation("cpu_v.sdo");
// synopsys translate_on

wire \m_machine|inc_pc ;
wire \m_machine|always0~73 ;
wire \m_machine|always0~74 ;
.....
.....
// atom is at LCFF_X21_Y13_N27
stratixii_lcell_ff \m_machine|inc_pc~I (
    .clk(\clk~clkctrl),
    .datain(\m_machine|Selector0~168),
    .adatasdata(gnd),
    .aclr(gnd),
    .aload(gnd),
    .sclr(! \m_machinecl|ena),
    .sload(gnd),
    .ena(vcc),
    .devclr(devclr),
    .devpor(devpor),
    .regout(\m_machine|inc_pc));
.....
.....
// atom is at PIN_C11
stratixii_io \pc_addr[12]~I (
    .datain(\m_counter|pc_addr[12]),
    .ddiodatain(gnd),
    .oe(vcc),
    .outclk(gnd),
```

```

        .outclk(vcc),
        .inclk(gnd),
        .inclk(vcc),
        .areset(gnd),
        .....
        ....);
// synopsys translate_off
defparam \pc_addr[12]~I.ddio_mode = "none";
defparam \pc_addr[12]~I.ddioinclk_input = "negated_inclk";
.....
.....
// synopsys translate_on
endmodule
/******cpu.v结束***** */

/*****cpu_v.sdo 开始***** */
// Copyright (C) 1991 - 2006 Altera Corporation
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its documentation, is governed
// by the terms and conditions of an applicable license agreement,
// revision 20060110
.....
.....
// Device: Altera EP2S15F484C3 Package FBGA484
.....
.....
// This SDF file should be used for ModelSim (Verilog) only
.....
(DELAYFILE
  (SDFVERSION "2.1")
  (DESIGN "cpu")
  (DATE "05/20/2007 14:15:10")
  (VENDOR "Altera")
  (PROGRAM "Quartus II")
  (VERSION "Version 6.1 Build 201 11/27/2006 SJ Full Version")
  (DIVIDER .)
  (TIMESCALE 1 ps)

  (CELL
    (CELLTYPE "stratixii_lcell_ff")
    (INSTANCE m_machine\|inc_pc\~I)
    (DELAY
      (ABSOLUTE
        (PORT clk (1240:1240:1240) (1285:1285:1285))
        (PORT datain (155:155:155) (155:155:155))
        (PORT sclr (1174:1174:1174) (1124:1124:1124))
        (IOPATH (posedge clk) regout (94:94:94) (94:94:94)))
      )
    )
  )
)

```

```

)
(TIMINGCHECK
  (SETUP datain (posedge clk) (90;90;90))
  (SETUP sclr (posedge clk) (90;90;90))
  (HOLD datain (posedge clk) (149;149;149))
  (HOLD sclr (posedge clk) (149;149;149))
)
)

(CELL
  (CELLTYPE "stratixii_lcell_comb")
  (INSTANCE m_machine\|always0\~73_I)
  (DELAY
    (ABSOLUTE
      (PORT dataa (279;279;279) (282;282;282))
      .....
      (IOPATH dataa combout (366;366;366) (366;366;366))
      .....
    )
  )
  .....
  .....
  .....

(CELL
  (CELLTYPE "stratixii_asynch_io")
  (INSTANCE pc_addr\[12\]\~I_inst1)
  (DELAY
    (ABSOLUTE
      (PORT datain (1606;1606;1606) (1956;1956;1956))
      (IOPATH datain padio (1998;1998;1998) (1998;1998;1998))
    )
  )
)
)

//----- cpu_v.sdo 的结束 -----



//-----stratixii_atom.v 的开始 -----
// Copyright (C) 1991–2006 Altera Corporation
.....
.....
// ***** PRIMITIVE DEFINITIONS *****
`timescale 1 ps/1 ps

```

```

// ***** DFFE

primitive STRATIXII_PRIM_DFFE (Q, ENA, D, CLK, CLRN, PRN, notifier);
    input D;
    input CLRN;
    input PRN;
    input CLK;
    input ENA;
    input notifier;
    output Q; reg Q;

    initial Q = 1'b0;

    table
        // ENA  D   CLK   CLRN  PRN  notifier  :  Qt  :  Qt+1
        // -----
        // (??) ? ? 1 1 ? : ? : -; // pessimism
        // x ? ? 1 1 ? : ? : -; // pessimism
        // 1 1 (01) 1 1 ? : ? : 1; // clocked data
        ..... .
        .....
    endtable

    endprimitive

module stratixii_dffe ( Q, CLK, ENA, D, CLRN, PRN );
    input D;
    input CLK;
    input CLRN;
    input PRN;
    input ENA;
    output Q;

    wire D_ipd;
    wire ENA_ipd;
    wire CLK_ipd;
    wire PRN_ipd;
    wire CLRN_ipd;

    buf (D_ipd, D);
    buf (ENA_ipd, ENA);
    buf (CLK_ipd, CLK);
    buf (PRN_ipd, PRN);
    buf (CLRN_ipd, CLRN);

```

```

wire    legal;
reg     viol_notifier;

STRATIXII_PRIM_DFFE ( Q, ENA_ipd, D_ipd, CLK_ipd, CLRN_ipd, PRN_ipd, viol_notifier );
and(legal, ENA_ipd, CLRN_ipd, PRN_ipd);

specify

    specparam TREG = 0;
    specparam TREN = 0;
    specparam TRSU = 0;
    specparam TRH = 0;
    specparam TRPR = 0;
    specparam TRCL = 0;

    $ setup ( D, posedge CLK &&& legal, TRSU, viol_notifier ) ;
    $ hold   ( posedge CLK &&& legal, D, TRH, viol_notifier ) ;
    $ setup ( ENA, posedge CLK &&& legal, TREN, viol_notifier ) ;
    $ hold   ( posedge CLK &&& legal, ENA, 0, viol_notifier ) ;

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

endspecify
endmodule

.....
//-----
`timescale 1 ps/1 ps

module stratixii_termination (
    rup,
    rdn,
    terminationclock,
    terminationclear,
    ....
    ....);
    input      rup;
    input      rdn;
    input      terminationclock;

```

```

input      terminationclear;
.....
....;
parameter runtime_control = "false";
parameter use_core_control = "false";
.....
....;

// BUFFERED BUS INPUTS
wire      rup_in;
wire      rdn_in;
wire      clock_in;
.....
....;

// TMP OUTPUTS
wire      incrup_out;
wire      incrdown_out;
wire [13:0] control_out;
.....
....;
// FUNCTIONS
....;
....;

// INTERNAL NETS AND VARIABLES
....;
....;

// TIMING HOOKS
buf (rup_in, rup);
buf (rdn_in, rdn);
buf (clock_in,terminationclock);
....;

specify
  (posedge terminationclock => (terminationcontrol +: control_out)) = (0,0);
  (posedge terminationclock => (terminationcontrolprobe +: controlprobe_out)) = (0,0);
endspecify
....;
....;

// output driver
buf buf_ctrl_out [13:0] (terminationcontrol,control_out);
buf buf_ctrlprobe_out [6:0] (terminationcontrolprobe,controlprobe_out);
....;
....;

// MODEL
....;
....;

```

```
assign incrup = incrup_out;
assign incrdown = incrdown_out;
assign incrup_out = (power_down == "true") ? (enable_in & rup_in) : rup_in;
.....
.....
stratixii_termination_digital rup_block(
    .rin(incrup_out),
    .clk(clock_in),
    .clr(clear_in),
    .ena(enal),
    .padder(pulldown_in),
    .devpor(devpor),
    .devclrn(devclrn),
    .ctrlout(rup_control_out)
);
defparam rup_block.runtime_control = runtime_control;
defparam rup_block.use_core_control = use_core_control;
.....
.....
stratixii_termination_digital rdn_block(
    .rin(incrdown_out),
    .clk(clock_in),
    .clr(clear_in),
    .ena(enal),
    .padder(pullup_in),
    .devpor(devpor),
    .devclrn(devclrn),
    .ctrlout(rdn_control_out)
);
defparam rdn_block.runtime_control = runtime_control;
defparam rdn_block.use_core_control = use_core_control;
.....
.....
endmodule
//-----
// Module Name : stratixii_routing_wire
// Description : Simulation model for a simple routing wire
//-----
`timescale 1ps / 1ps
```

```

module stratixii_routing_wire (datain, dataout );
    // INPUT PORTS
    input datain;
    // OUTPUT PORTS
    output dataout;
    // INTERNAL VARIABLES
    wire dataout_tmp;

    specify
        (datain => dataout) = (0, 0) ;
    endspecify

    assign dataout_tmp = datain;
    and (dataout, dataout_tmp, 1'b1);

endmodule // stratixii_routing_wire

```

//-----stratixii_atom.v 的结束-----

从上面提供的带门延迟的布局布线门级网表,可以了解布局布线后产生的带.vo 扩展名的网表的实质。从本质上说,这是一种比综合器产生的更接近实际电路结构的 Verilog 门级和源语基础部件级的源代码。这种代码有自己的仿真行为,但也有确定的电路制造参数与之对应,所以是可以实现的,上面列出的代码明确地说明了这个问题。对于源语基础部件级 Verilog 语法的深入了解是微电子工艺师和电路系统设计师都必须了解和掌握的,但苦于时间和篇幅的问题,我们将在高级教程里作深入的讲解。

不同的 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 Verilog HDL 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 上实现的小型 RISC_CPU 系统。

小 结

从上面的例子可以看到,复杂的 RISC_CPU 设计其实是一个从抽象到具体的逐步接近的

分析和实现过程。一个大型的设计先从概念出发,用 Verilog 写出抽象的功能块描述,把许多复杂的细节掩盖起来。然后,从行为级分析功能块之间的关系,通过仿真逐步验证,发现问题,改动模块代码使其逐步趋向合理,并最后可以用 RTL 级 Verilog 源代码模块来表示。接下去就可以通过自动综合工具把 RTL 级 Verilog 源代码模块综合成电路网表,再通过布局布线工具让它们更具体化。在基础器件源语级基础上的系统精确仿真结果正确,可以使系统电路芯片的制作有 90%以上的一次流片成功把握。这就是我们为什么要学习 Verilog 高层次先进设计方法的原因。

思 考 题

1. 请叙述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. 改进本章中的 RISC_CPU 系统,把指令数增至 16,寻址空间降为 4 KB,并书写设计报告,实现三个层次的仿真运行。
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?

第 18 章 虚拟器件/接口、IP 和基于平台的设计方法及其在大型数字系统设计中的作用

概 述

在现代数字系统芯片设计制造技术中最重要的基本概念之一是采取什么手段能确保如此复杂系统设计能赶上瞬息万变的市场变化和逻辑设计的精确，并提高一次流片的成功率，以降低设计和制造成本。商业化的软核和硬核、宏单元、虚拟器件和接口的应用普及，大大提高了设计制造效率，降低了设计和生产成本。推广知识产权模块(即 IP)重用技术，学习编写可以被国际电子工商业界认可的 IP 代码是我国电子工业起飞的关键之一。本章通过实例介绍这些已经被国际同行所公认的基本概念，希望同学们在日后在工作中积极学习和应用。

18.1 软核和硬核、宏单元、虚拟器件、设计和验证 IP 以及基于平台的设计方法

宏组件(Macrocells 或 Megacells)或核(Cores)是预先设计好的，其功能经过验证的、由总数超过 5000 个门构成的一体化的电路模块，这个模块可以是以软件为基础的，也可以是以硬件为基础的。这就是我们在第 1 章中曾经讨论过的软核和硬核，这种具有知识产权的模块在集成电路设计行业常被称为 IP(intellectual property)，IP 通常分为设计和验证 IP。所谓设计 IP，即虚拟组件/芯片(Virtual Chips)可以是用软核/硬核构成的器件，即用 RTL/Netlist 级 Verilog 或 VHDL 语言描述的电路模型。通过参数配置可以将该模型转换成系列化的具体电路组件。在新系统的研制过程中，借助 EDA 综合工具，虚拟组件(即宏组件)可以很容易地调整或改变参数，也能很容易地与其他外部逻辑结合为一体，并加以验证，从而大大扩展了设计者可选用的资源范围。掌握 IP 的重用技术可大大缩短设计周期，加快高技术新芯片的投产和上市。而所谓验证 IP 则是用系统级 Verilog 或 VHDL 语言描述的常用大规模集成电路(如 ROM 和 RAM)、总线接口和 CPU 的行为模型等，往往是不可综合的，也没有必要综合成具体电路(其电路制造版图需要申请，得到许可后才允许使用)，但其所有对外的性能与真实的器件或接口完全一致，在仿真时可用来代替真实的部件，用以验证所设计的电路(必须综合的部分)是否正确。

在美国和电子工业先进的国家，各种微处理器芯片(如 8051, ARM 系列 CPU 等)、通用串行接口芯片(如 8251 等)、中断控制器(如 8259 等)、并行输入输出接口(如 PIO 等)、直接存储器存取(如 DMA 控制器等)、数字信号处理器(DSP)、SDRAM、NAND Flash、USB 控制器以及 PCI 总线控制接口等都有其相对应的商品化的软/硬设计 IP 和验证 IP 可供选用。有的 IP 核能免费提供门级网表和 RTL 级的 Verilog HDL 或 VHDL 源代码，而大多数只提供行为模

型,而虚拟接口模型往往只提供系统级行为代码。这是因为门级和 RTL 级的 Verilog 或 VHDL 是可综合的,它与具体的逻辑电路有着精确的对应关系,往往需要申请使用许可证,并付一定费用,签订合同后,才能提供 RTL 级源代码或者仿真用代码。在 FPGA 工具中也有许多宏组件可以用,有些是免费的,有些需要付费才能使用。

近年来,在现代数字系统设计领域中发展最快的一个部门就是提供虚拟器件和虚拟接口模型的设计和服务。目前国际上有一个称为虚拟接口联盟(VSIA)的组织,它是协调虚拟器件和虚拟接口模型的设计标准和服务工作的国际组织。该组织认为虚拟器件和虚拟接口模型必须符合通用的工业标准和达到一定的质量水准,才能发布。这对选用虚拟器件和虚拟接口模型来设计复杂系统的工程师们无疑有很大的帮助。若采用虚拟器件和虚拟接口模型技术来设计复杂的数字系统,则必将大大缩短设计周期并提高设计的质量,也为千万门级 SoC 芯片的实现铺平了道路。在 Quartus II 6.1 中可以通过主窗口菜单 Tools→MegaWizard Plug-In Manager…弹出对话框,做适当选择后便可以进入一个选择菜单。设计者可以在 Installed Plug-Ins 和 IP MegaStore 之间做选择,前者是一些常用的规模不很大的 IP,后者包括了嵌入式处理器 IP、复杂接口和外围电路 IP、复杂 DSP 和通信 IP。几乎包括了所有设计者可以想到的各种知识产权的模块。但绝大部分必须付费后才能得到。究竟是自己独立开发还是利用现成的 IP,取决于系统设计者可以利用的财务资源和设计工作的进度要求,不同的设计项目有不同的应对策略。

进入 21 世纪后,国际上还成立了一个系统寄存器语言(Register Description Language)联盟,该组织为 SoC 芯片的设计提供了一种 SystemRDL 语言,该语言的配置空间寄存器(Configuration Space Registers,英文缩写为 CSR's)可以用来储存定义 IP 核或者芯片操作的成千上万个关键参数。经过 SystemRDL 语言处理的 IP 可允许系统架构工程师、软件开发人员、硬件设计师和验证工程师利用计算机可识别的共同的 CSR's 格式,正确地布置电路结构,并行地实施 SoC 芯片的开发流程,大大加快了 SoC 芯片的开发过程,降低了开发成本。

国际上,系统芯片(SoC)的设计方法学经历了以时序为主导的设计(Time Driven Design,简称 TDD)和基于块的设计(Block-Based Design,简称 BBD)的前两个发展阶段,编写行为的和可综合的 Verilog HDL 模块是这两个发展阶段中的主要工作,工作范围也只局限于公司内部。而进入 21 世纪以来,SoC 设计方法学开始进入了基于平台的设计方法。基于平台的设计(Platform-Based Design,简称 PBD)方法学,要求我们把尽可能多的优秀的可重复使用的设计资源(IP 核)集中到一个统一的平台上,供设计人员选用。只有依赖这样一个设计平台,才能高效率、低成本地完成现代复杂 SoC 系统的设计、验证、分析和优化,为 SoC 的成功投片提供保障。而目前我国大陆的设计公司大多数正处在 TDD 阶段,少数开始进入 BBD 的发展阶段。

以上这些设计方法的改变对于有经验的系统工程师而言只是实现手段略有改变而已。新的 PBD 设计方法从本质而言,只是又回归到基于处理器的利用现成集成电路块的实现方式,而不再需要专门设计很多定制电路。之所以能实现这样的回归,是因为利用计算机平台可以根据应用系统的需求,对处理器核进行必要的配置,并利用已积累的各种可重用可配置的 IP 核,自动生成远比由通用处理器芯片和通用外围集成电路芯片构成的线路板性能/价格比更好的 SoC 芯片。

特别需要指出的是:在国际高档电子消费品市场上,采用落后设计手段设计的产品是绝对

不可能赢得竞争胜利的。现代高档次消费类电子产品必须采用 PBD 方法来设计系统芯片(SoC),才有可能在国际市场中取得一席之地,在市场竞争中赢得胜利。

18.2 设计和验证 IP 供应商

在这一节中列出几个在 SoC 设计行业有良好声誉的设计和验证 IP 供应服务商的网页地址(见表 8.1),并简单介绍它们所能提供的产品和服务,供读者参考。

类似的 IP 设计服务公司可以通过这几个公司的合作伙伴找到,现代高性能复杂 SoC 芯片的设计离不开各具特色的公司之间的合作。我国电子设计工作者为了更快地跟上国际的设计水平,必须注意开展国际合作,同时注意保持自己的产品特色和市场方向。

表 18.1 虚构器件和接口公司简介

公司名	公司简介	提供的 IP 和服务特色
Denali 网页: http://www.denali.com	主要业务是为电子工业宽带通信、网络和其他部门提供高级储存器系统设计关键技术和软件设计服务。该公司推出的“存储器建模器-高级验证”(MMAV)已经成为电子行业存储器元件建模和仿真的工业标准	PCI Express IP; DDR DRAM 控制器 IP; NAND Flash 存储器控制器 IP; NAND Flash 文件系统 IP MMAV 存储器接口功能验证 IPs
ARC International 网页: http://www.arc.com	世界领先的嵌入式专用高性能 32 位 RISC/DSP 处理器 IP 核开发公司,并提供一体化的开发工具,实时操作系统和软件。该公司提供的 IP 解决方案,可以帮助用户迅速地开发下一代无线、网络和高级消费电子产品,降低完全依靠自己开发 SoC 芯片的风险	微处理器:ARC 700 系列 ARC 600 系列 子系统:ARC 游戏机子系统 ARC 高级音响子系统 ARC 视频子系统
Artisan Components, Inc 网页: http://www.artisan.com/	是 ARM 公司为用户提供 IP 解决方案的部门。该公司可为用户提供下一代电子产品所需要的处理器 IP 核、物理 IP 核、高速缓存和 SoC 设计、专用的标准产品、有关的软件和开发工具等设计和咨询服务	为基于 ARM 核的系统设计提供技术支持和咨询;派遣有经验的 SoC 工程师帮助用户集成带 ARM IP 核的 SoC 芯片;以用户指定的工艺将 ARM 软核和库中的 RAM 和外围 IP 等制成 ASIC 电路;ARM 硬核设计移植树到用户指定的工艺;多 ARM CPU 核和 AMBA™ 多层结构总线子系统和专用 SoC 芯片的开发支持。高级验证服务,包括 AMBA 兼容的测试平台环境的支持。评价板和 FPGA 的开发;第三方 IP 与 AMBA 总线的接口;操作系统的移植 Windows CE, Linux 和其他主要的实时操作系统

续表 18.1

公司名	公司简介	提供的 IP 和服务特色
Virage logic Corporation 网页： www.viragelogic.com	该公司为复杂集成电路设计提供高级的存储器 IP，无论在技术上还是在市场占有率上，该公司在全世界都是领先的。目前在半导体行业，该公司的 IP 已经得到同行的认可和信任。在嵌入式存储器、逻辑和 I/O 接口等领域，成为全球 IP 平台的领袖。公司提供的各种不同的 IP，其性能更高、功耗更低、密度更大，针对不同的制造工艺，对 IP 的设计进行了优化，无论在消费电子产品，还是在通信、网络手持设备和计算机市场上，该公司的 IP 都已得到了广泛的认可	Virage Logic 公司的逻辑 IPs 针对不同的市场应用，用手工进行优化，以满足最高的质量和性能标准。该公司提供的 ASAP 逻辑单元库基于公司专利的布线方法学和单元结构，已在 2500 多个产品的设计中得到了验证。许多消费类芯片的产量很大。每个 ASAP 逻辑结构都是针对特定目标工艺开发的，无论电路设计、布线、功耗和面积都经过手工优化。硅片电路的性能是有保障的 其金属可编程单元库可提供价格低廉的掩膜。适用于高速、高密度结构，对专用的标准部件、电路功能的快速实现、芯片产量规模中等或者较小的应用方案非常合适 其标准单元库的部件布线已经过改进，性能、面积和功耗均比传统的标准单元库有显著的改进，用户可以根据不同的应用目标选择高密度、超高密度和超低功耗的标准单元结构
Rambus 网页： www.rambus.com	该公司掌握芯片与芯片接口产品和服务的领先技术和工程专业知识，可以帮助 SoC 公司解决棘手的 I/O 接口问题，将工业界领先的 SoC 芯片和系统推向市场。该公司的产品可以在无数的高性能计算、消费电子产品和网络产品中找到	整体储存器解决方案：XDR™；目前世界上最快的处理器总线 FlexIO™ 数据率高达 12.5 Gbps 的高速 SerDes 解决方案 Advanced Backplane

18.3 虚拟模块的设计

我国大陆地区由于复杂芯片的设计工作开展较晚，经费也比较少，目前许多单位有还不能及时得到商业化的虚拟模块和接口，因此就有必要自己来设计虚拟接口模型。下面的例子说明了怎样根据数据手册和波形图来编写虚拟接口模型，从而完成与商业芯片的接口设计。

[例 18.1] 模数转换器 AD7886 仿真模型(虚拟模块)的设计。

下面介绍的名为 ADC 的 Verilog 模块在设计中可以用来模拟实际的模数转换器(下面简称 A/D)AD7886。因此，该仿真模型的输入与各输出信号间的逻辑关系，必须严格按照数据手册描述的波形编写，信号间的时间关系也必须完全符合手册要求，这样才能起到虚拟模块的作用。只有这样在设计电路的测试中才能用它来代替实际器件。同时，虚拟模块还应具备实际电路所没有的功能，即对于不符合要求的输入信号还能产生错误提示。在实际的电路中，我们很难控制 A/D 的输出数据，然而在该设计中，可以编写数据文件，得到想要的各种类型的数据来测试后续电路的功能，并可以随时根据测试要求更改数据，非常方便。虚拟模块的编写是

Verilog 语言应用的重要方面。它为 ASIC 设计投片一次成功提供了可能。

在实际电路中, A/D 包括模拟部分和许多必要的控制和参考电平输入。为了简单和说明问题起见, 只介绍 A/D 模块有关数字接口的一部分功能, 把这部分功能编写成虚拟模块。其中只包括了 A/D 控制信号的输入、数据总线和“忙”信号的输出。为了进一步简化还假设选片信号 CS 和读信号 RD 总是为低电平(有效)。因此, 该模型实际上是为教学目的而编写的简化虚拟模块, 在仿真时仅能代替真实 A/D 的一部分功能。它类似一个数据发生器, 根据输入控制信号和 A/D 自身的特性输出一个字节(8位)数据和“忙”信号。同时根据手册规定, 不断检测输入信号是否符合要求。虚拟模型的精确与否, 直接影响到设计是否能够一次投片成功。因此在 ASIC 系统芯片的设计中应予以充分的重视。

AD7886 是具有一个高速的 8 位三态数据输出接口的模数转换器, 转换的过程由输入信号 CONVST 控制, 数据的存取由选片信号 CS 和读信号 RD 输入信号控制(低电平有效)。下面的 Verilog 源代码描述了该 A/D 的转换启动和数据读出功能(假设 CS 和 RD 都为低电平), 根据手册的说明, 输入和输出波形如图 18.1 所示。

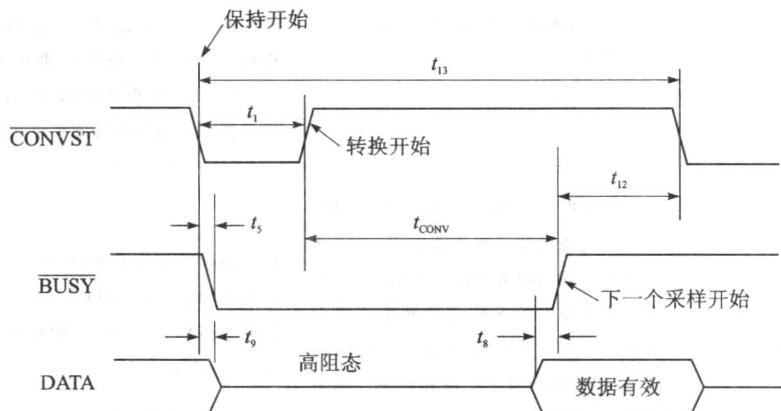


图 18.1 A/D 转换启动和数据读出时序 ($\overline{CS}=\overline{RD}=0$)

为使所设计的虚拟模块对输入信号有检测功能, 还在模块中加入了提示输入信号有错的语句。输出的 8 位数据可以根据要求自己编制, 从数据文件 AD.DATA 中读取。下面是一个名为 ADC.V 的文件, 描述了该 A/D 转换器波形所示的这一部分功能。其仿真模块的具体源代码如下:

```
//+++++  
'timescale 100ps/100ps          // 定义时间单位和时间分辨率  
module adc (nconvst, nbusy, data);  
    input nconvst;                // A/D 启动脉冲 ST, 见图 18.1 中  
    output nbusy;                 // A/D 工作标志, 见图 18.1 中  
    output data;                  // 数据总线, 从 AD.DATA 文件中读取数据后经端口输出  
    reg[7:0] databuf,i;          // 内部寄存器  
    reg nbusy;  
    wire[7:0] data;  
    reg[7:0] data_mem[0:255];
```

```

reg      link_bus;
integer   tconv,
          t5,
          t8,
          t9,
          t12;
integer   width1,
          width2,
          width;
//时间参数定义(依据 AD7886 手册):
always @(negedge nconvst)
begin
  tconv = 9500 + { $random } % 500; // (type 950 ns, max 1 000 ns) Conversion Time
  t5 = { $random } % 1000; // (max 100ns) CONVST to BUSY Propagation Delay
                           // CL = 10pF
  t8 = 200;           // (min 20) CL=20pF Data Setup Time Prior to BUSY
                     // (min 10) CL=100pF
  t9 = 100 + { $random } % 900; // (min 10ns, max 100ns) Bus Relinquish Time After
                               // CONVST
  t12 = 2500;         // (type) BUSY High to CONVST Low, SHA Acquisition Time
end

initial
begin
  $readmemh("adc.data",data_mem); // 从数据文件 adc.data 中读取数据
  i = 0;
  nbusy = 1;
  link_bus = 0;
end

assign data = link_bus? databuf:8'bzz; // 三态总线
/* -----
在信号 nconvst 的负跳降沿到来后,隔 t5 s nbusy 信号置为低,tconv 是 AD 将模拟信号转换为数字信号的时间,在信号 nconvst 的正跳降沿到来后经过 tconv 时间后,输出 nbusy 信号变为高。
----- */
always @(negedge nconvst)
fork
  # t5    nbusy = 0;
  @(posedge nconvst)
  begin
    # tconv  nbusy = 1;
  end
end

```

```

join
/* -----
nconvst 信号的下降沿触发, 经过  $t_9$  延时后, 把数据总线输出关闭置为高阻态。
nconvst 信号的上升沿到来后, 经过( $t_{conv} - t_8$ )时间, 输出一个字节(8位数据)到 databuf, 该数据来自
于 data_mem。而 data_mem 中的数据是初始化时从数据文件 AD.DATA 中读取的。此时应启动总
线的三态输出。
----- */
always @(negedge nconvst)
begin
  @(posedge nconvst)
  begin
    #(tconv - t8) databuf = data_mem[i];
  end

  if(wideth < 10000 && wideth > 500)
  begin
    if(i == 255) i = 0;
    else i = i + 1;
  end
  else i = i;
end

//在模数转换期间关闭三态输出, 转换结束时启动三态输出
always @(negedge nconvst)
fork
  #t9 link_bus = 1'b0; //关闭三态输出, 不允许总线输出
  @(posedge nconvst)
  begin
    #(tconv - t8) link_bus = 1'b1;
  end
join
/* -----
当 nconvst 输入信号的下一个转换的下降沿与 nbusy 信号上升沿之间时间延迟小于  $t_{12}$  时, 将会出现
警告信息, 通知设计者请求转换的输入信号频率太快, A/D 器件转换速度跟不上。
仿真模型不仅能够实现硬件电路的输出功能, 同时能够对输入信号进行检测。当输入信号不符合
手册要求时, 显示警告信息。
----- */
//检查 A/D 启动信号的频率是否太快
always @(posedge nbusy)
begin
  #t12;
  if (! nconvst)
  begin

```

```
$ display("Warning! SHA Acquisition Time is too short!");
end
// else $ display(" SHA Acquisition Time is enough!");
end
//检查 A/D 启动信号的负脉冲宽度是否足够和太宽

always @(negedge nconvst)
begin
wideth= $ time;
@(posedge nconvst) wideth= $ time-wideth;
if (wideth<=500 || wideth > 10000)
begin
$ display("nCONVST Pulse Width = %d",wideth);
$ display("Warning! nCONVST Pulse Width is too narrow or too wide!");
// $ stop;
end
end
endmodule
//+++++
```

对商业化的虚拟模块有着严格的要求,不但要求在系统设计的仿真中能完全来代替真实的器件,而且还希望能提示产生错误的原因。虚拟模块的精确与否,直接决定设计的成败。ASIC 的投片成本很高,编写虚拟模块时任何小的疏忽都有可能造成投片的失败,造成大量资金的浪费。因此编写这样的模块是一件复杂而细致的工作,需要极其认真的工作态度和作风,必须认真对待。为了简单起见,本节介绍的模块只具有 AD7886 的一部分功能,所以还不能称为 AD7886 完整的虚拟模块。

通过上述简单的例子能了解虚拟模块是如何设计的,对大多数的电路系统工程师来说,应该尽量利用商业化的虚拟模块来设计自己的电路系统。只有在没有办法得到商业化的虚拟模块时,才利用器件手册来编写虚拟模块,因为编写精确的虚拟模块需要花费很多时间和精力。

18.4 虚拟接口模块的实例

下面介绍两个常用的大规模集成芯片:通用串行收发控制器 USART8251 和 Intel8085 微处理器 CPU 的虚拟接口模块。这两个用 Verilog HDL 描述的虚拟接口的行为模块是由 Verilog 语言的创始人 P. R. Moorby 和 D. E. Thomas 合作编写的(这是我们从 Internet 网络上下载得到的)。

因为商品化的虚拟器件和虚拟接口模型是知识产权(简称 IP),必须保证设计所需的参数绝对正确,因此价格非常昂贵,不可能免费得到。下面的模块从严格意义上说来并非是真正的虚拟接口模型,因为它们并不对用户设计的成败负责。把它们列在这里只是拿它们作为学习

编写较复杂的 Verilog HDL 行为模块的样本而已。

[例 18.2] “商业化”的虚拟模块之一: Intel USART 8251A (通用串行异步收发器芯片)

为节省篇幅本书再版中省略了这段代码。感兴趣的读者可以自己在网址上寻找通用串行异步收发器 8251 或其他智能接口的 Verilog HDL 行为代码。

[例 18.3] “商业化”的虚拟模块之二: Intel 8085a 微处理器的行为描述模块。

为节省篇幅本书再版中省略了这段代码。感兴趣的读者可以自己在网络上寻找 Intel 8085a 或者其他处理器的 Verilog HDL 行为代码。

上面两个例子是常用的微处理器 CPU 和外围芯片。在系统芯片的设计中,可以用虚拟模型来代替真实的器件对自己所设计的电路功能进行仿真,全面精确地验证自己所设计的部分是否正确。在 ASIC 的制造过程中可以利用现存的与之对应的门级结构的电路实体来实现电路的功能。这样就能用较快的速度把许多人的劳动成果集合在一起,把一个极其复杂的数字系统集成在一个很小的硅片上。上面两个例子的代码也可从本见第一版中查阅而得。

小 结

推广商业化 IP 模块的编写,普及 IP 重用技术,推广基于平台的设计(PBD)方法学对于提高我国微电子产品的档次,降低设计成本将会产生重大影响。在 IP 模块的编写中对接口指标的描述和处理特别重要。只有对这一点有深刻的理解才可能编写出有实用价值的硬核虚拟模块和 RTL 级的软核模块,因为系统芯片其设计和验证过程已经变得非常复杂,设计质量的控制必须分级加以管理,这样接口就成为系统验证的瓶颈。当接口信号连接时出现问题能清晰地报告和说明故障所在的 IP 模块才有使用价值。要做到这一点不但需要设计师有高度的工作责任心还需要有复杂的质量管理体系的保证,设计这个质量保证系统的管理者必须是很有经验的高级系统设计师。我国目前还缺少这样的人才,需要借鉴国外的经验加速培养。

思 考 题

1. 为什么要设计虚拟模块?
2. 虚拟模块有几种类型?
3. 为什么在 ASIC 设计中要尽量利用商业化的虚拟模块和 IP 技术?
4. 为什么说编写完整精确的虚拟模块,编写者不但需要全面熟练地掌握 Verilog 语言,还需要有高度的责任性,并且需要有一个严格的质量保证体系来确保与工艺的电路的一致性?
5. 什么是基于平台的设计方法学,为什么该设计方法具有最高的设计效率?

第三部分 设计示范与实验练习

概 述

在第一部分和第二部分学习的基础上,通过 Verilog 设计教程第三部分实践篇——设计示范和上机习题之 12 个阶段练习,一定能逐步掌握 Verilog HDL 设计的要点。可以先理解设计示范模块中每一条语句的作用,进行功能仿真来加深理解;然后对示范模块进行综合;再分别进行生成的逻辑网表和布线后生成的带布线延迟的门级器网表时序仿真,以加深印象和深入理解。在此基础上再独立完成每一阶段规定的练习。当 12 个阶段的练习做完后,便可以设计一些简单的逻辑电路和系统。最好利用一个月的集中训练时间,能很快过渡到设计相当复杂的数字逻辑系统。当然,复杂的数字系统的设计和验证,不但需要系统结构知识和丰富经验的积累,还需要了解更多的语法现象和掌握高级的 Verilog HDL 系统任务,以及与 C 语言模块接口的方法(即 PLI),这些已超出本书的范围。有兴趣的同学可以阅读 Verilog 硬件描述语言参考资料和有关文献,开始自学。

练习一 简单的组合逻辑设计

目的：

- (1) 掌握基本组合逻辑电路的实现方法；
- (2) 初步了解两种基本组合逻辑电路的生成方法；
- (3) 学习测试模块的编写；
- (4) 通过综合和布局布线了解不同层次仿真的物理意义。

下面的模块描述了一个可综合的数据比较器。从语句可以很容易地看出，其功能是比较数据 a 与数据 b 的结果，如果两个数据相同，则给出结果 1，否则给出结果 0。在 Verilog HDL 中，描述组合逻辑时常用的 assign 结构。注意 $\text{equal} = (\text{a} == \text{b}) ? 1 : 0$ ，这是一种在组合逻辑实现分支判断时常用的格式。

模块源代码的方法之一：

```
//-----文件名 compare.v -----  
module compare(equal,a,b);  
    input a,b;  
    output equal;  
    assign equal = (a==b)? 1 : 0;  
    //a 等于 b 时, equal 输出为 1;a 不等于 b 时, equal 输出为 0  
endmodule
```

模块源代码的方法之二：

```
module compare(equal,a,b);  
    input a,b;  
    output equal;  
    reg equal;  
    always @ (a or b)  
        if(a==b)      //a 等于 b 时, equal 输出为 1  
            equal = 1;  
        else          //a 不等于 b 时, equal 输出为 0  
            equal = 0;  //思考:如果不写 else 部分会产生什么逻辑  
  
endmodule
```

测试模块用于检测模块设计是否正确。它给出模块的输入信号，观察模块的内部信号和输出信号，如果发现结果与预期有偏差，则需要对设计模块进行修改。

测试模块源代码的方法之一：

```
'timescale 1ns/1ns           //定义时间单位  
'include  "./compare.v"      //包含模块文件,在有的仿真调试环境中并不需要此语句,而  
                                //需要从调试环境的菜单中键入有关模块文件的路径和名称  
module t;
```

```

reg a,b;
wire equal;
initial //initial 常用于仿真时信号的给出
begin
    a=0;
    b=0;
# 100 a=0; b=1;
# 100 a=1; b=1;
# 100 a=1; b=0;
# 100 a=0; b=0;
# 100 $ stop; //系统任务,暂停仿真以便观察仿真波形
end

compare m(.equal(equal),.a(a),.b(b)); //调用被测试模块 t.m

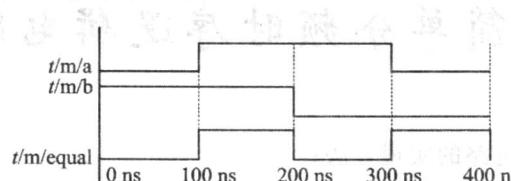
endmodule

```

综合就是把 compare.v 文件送到 Synplify 或其他综合器处理,在选定实现器件和选取生成 Verilog 网表的前提下,启动综合器的编译。综合器会自动生成一系列文件,向操作者报告综合的结果。其中生成的 Verilog Netlist 文件(扩展名为. vm),表示自动生成的门级逻辑结构网表,仍然用 Verilog 语句表示,但比输入的源文件更具体,可以用测试模块调用它做同样的仿真,运行的结果更接近实际器件。

布局布线就是把综合后生成的另一种文件(EDIF),在布线工具控制下进行处理,启动布线工具的编译。布局布线工具会自动生成一系列文件,向操作者报告布局布线的结果。其中生成的 Verilog Netlist 文件(扩展名为. vo),表示自动生成的具体基本门级结构和连接的延迟,仍然用 Verilog 基本部件结构语句和连接线的延迟参数的重新定义表示,库中的基本部件也更进一步具体化了,比综合后的扩展名为. vm 的文件更具体。可以用同一个测试模块调用它做同样的仿真,运行结果与实际器件运行结果几乎完全一致。

组合逻辑仿真波形部分如实验图 1 所示。



实验图 1 组合逻辑仿真波形

测试模块源代码的方法之二:

```

`timescale 1ns/1ns      //定义时间单位
`include ". /compare.v" //包含模块文件。在有的仿真调试环境中并不需要此语句
                           //而需要从调试环境的菜单中键入有关模块文件的路径和名称
module t;
reg a,b;

```

```

reg clock;
wire equal;
initial // initial 常用于仿真时信号的给出
begin
    a=0;
    b=0;
    clock = 0; // 定义一个时钟变量
end
always # 50 clock = ~clock; // 产生周期性的时钟
always @ (posedge clock) // 在每次时钟正跳变沿时刻产生不同的 a 和 b
begin
    a = { $random } % 2; // 每次 a 是 0 还是 1 是随机的
    b = { $random } % 2; // 每次 b 是 0 还是 1 是随机的
end
initial
begin    # 100000 $stop; end // 系统任务,暂停仿真以便观察仿真波形

compare m(.equal(equal),.a(a),.b(b)); // 调用被测试模块 t.m
endmodule

```

【练习题 1】设计一个字节(8位)的比较器。

要求:比较两个字节的大小,如 $a[7:0]$ 大于 $b[7:0]$,则输出高电平,否则输出低电平;并改写测试模型,使其能进行比较全面的测试。观测 RTL 级仿真、综合后门级仿真和布线后仿真有什么不同,并说明这些不同的原因。从文件系统中查阅自动生成的 compare.v, compare.vo 文件和 compare.v 做比较,说出它们的不同点和相同点。

【思考题 1】在测试方法二中,第二个 initial 块有什么用?它与第一个 initial 块有什么关系?如果在第二个 initial 块中,没有写 # 100000 或者 \$stop,仿真会如何进行?比较两种测试方法,那一种测试方法更全面?

练习二 简单分频时序逻辑电路的设计

目的:

- (1) 掌握最基本时序电路的实现方法;
- (2) 学习时序电路测试模块的编写;
- (3) 学习综合和不同层次的仿真。

在 Verilog HDL 中,相对于组合逻辑电路,可综合成具体电路结构的时序逻辑电路也有标准的表述方式。在可综合的 Verilog HDL 模型,通常使用 always 块和 @(posedge clk) 或 @(negedge clk) 的结构来表述时序逻辑。下面是一个 1/2 分频器的可综合模型。

```

//----- 文件名:half_clk.v -----
module half_clk(reset,clk_in,clk_out);
    input clk_in,reset;

```

```

module half_clk(.reset(reset),.clk_in(clk),.clk_out(clk_out));
    output clk_out;
    reg clk_out;
    reg clk_in;
    always @ (posedge clk_in)
    begin
        if (!reset) clk_out = 0;
        else clk_out = ~clk_out;
    end
endmodule

```

在 always 块中,被赋值的信号都必须定义为 reg 型,这是由时序逻辑电路的特点所决定的。对于 reg 型数据,如果未对它进行赋值,仿真工具会认为它是不定态。为了能正确地观察到仿真结果,并确定时序电路的起始相位,在可综合风格的模块中,通常定义一个复位信号 reset,当 reset 为低电平时,对电路中的寄存器进行复位。

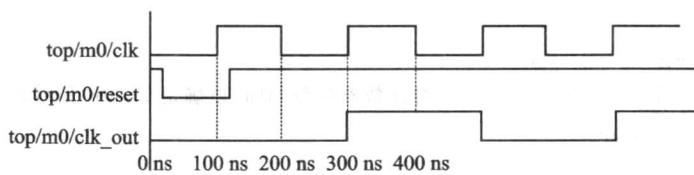
测试模块的源代码:

```

//----- 文件名 top.v -----
`timescale 1ns/100 ps
`define clk_cycle 50
module top;
reg clk,reset;
wire clk_out;
initial
begin
    clk = 0;
    reset = 1;
#10 reset = 0;
#110 reset = 1;
#100000 $stop;
end
half_clk m0(.reset(reset),.clk_in(clk),.clk_out(clk_out));
endmodule

```

简单的分频时序仿真波形如实验图 2 所示。



实验图 2 简单的分频时序波形

【练习题 2】 依然作 clk_in 的 2 分频 clk_out, 要求输出时钟的相位与上面的 1/2 分频器的输出正好相反。编写测试模块, 给出仿真波形。改变输入时钟的频率, 观测 RTL 级仿真、综合后门级仿真和布线后仿真的不同, 并写出报告。

【思考题 2】 如果没有 reset 信号, 能否控制 2 分频 clk_out 信号的相位? 只用 clk 时钟沿的触发(即不用 2 分频产生的时钟沿)如何直接产生 4 分频、8 分频或者 16 分频的时钟? 如何只用 clk 时钟沿的触发(不用 2 分频产生的时钟沿)直接产生占空比不同的分频时钟?

练习三 利用条件语句实现计数分频时序电路

目的:

- (1) 掌握条件语句在简单时序模块设计中的使用;
- (2) 学习在 Verilog 模块中应用计数器;
- (3) 学习测试模块的编写、综合和不同层次的仿真。

与常用的高级程序语言一样, 为了描述较为复杂的时序关系, Verilog HDL 提供了条件语句供分支判断时使用。在可综合风格的 Verilog HDL 模型中, 常用的条件语句有 if...else 和 case...endcase 两种结构, 用法和 C 程序语言中类似。两者相比, if...else 用于不很复杂的分支关系, 实际编写可综合风格的模块, 特别是用状态机构成的模块时, 更常用的是 case...endcase 风格的代码。这一节给的是有关 if...else 的范例, 有关 case...endcase 结构的代码以后会经常用到。

下面给出的范例也是一个可综合风格的分频器, 可将 10 MHz 的时钟分频为 500 kHz 的时钟。基本原理与 1/2 分频器是一样的, 但是需要定义一个计数器, 以便准确获得 1/20 分频。

模块源代码:

```
// ----- fdivision.v -----
module fdivision(RESET,F10M,F500K);
    input F10M,RESET;
    output F500K;
    reg F500K;
    reg [7:0]j;
    always @ (posedge F10M)
        if(! RESET)           //低电平复位
            begin
                F500K <= 0;
                j <= 0;
            end
        else
            begin
                if(j == 19)      //对计数器进行判断, 以确定 F500K 信号是否反转
                    begin
                        j <= 0;
                        F500K <= ~F500K;
                    end
                end
            end
endmodule
```

```

    end
else
    j <= j+1;
end
endmodule

```

测试模块源代码：

```

//----- fdivision_Top.v -----
`timescale 1ns/100 ps
`define clk_cycle 50

module division_Top;
reg F10M,RESET;
wire F500K_clk;

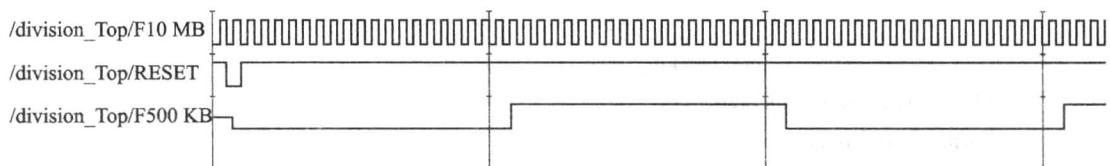
always # `clk_cycle F10M_clk = ~ F10M_clk;

initial
begin
    RESET=1;
    F10_M=0;
    # 100 RESET=0;
    # 100 RESET=1;
    # 10000 $ stop;
end

fdivision fdivision (.RESET(RESET),.F10_MB(F10M),.F500K(F500K_clk));
endmodule

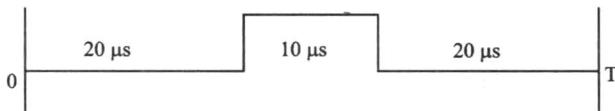
```

计数分频器的仿真波形如实验图 3 所示。



实验图 3 计数分频器波形

【练习题 3】 利用 10 MHz 的时钟,设计一个单周期形状的周期波形。



练习四 阻塞赋值与非阻塞赋值的区别

目的：

- (1) 通过实验,掌握阻塞赋值与非阻塞赋值的概念和区别;
- (2) 了解非阻塞和阻塞赋值的不同使用场合;
- (3) 学习测试模块的编写、综合和不同层次的仿真。

阻塞赋值与非阻塞赋值,在教材中已经了解了它们之间在语法上的区别以及综合后所得到的电路结构上的区别。在 always 块中,阻塞赋值可以理解为赋值语句是顺序执行的,而非阻塞赋值可以理解为赋值语句是并发执行的。时序逻辑设计中,通常都使用非阻塞赋值语句,而在实现组合逻辑的 assign 结构中,或者 always 块结构中都必须采用阻塞赋值语句。

下例两个模块(blocking.v 和 non_blocking.v)分别采用阻塞赋值语句和非阻塞赋值语句编写,看上去,但两者的含义存在重要区别。

模块源代码：

```
// ----- blocking.v -----
module blocking(clk,a,b,c);
    output [3:0] b,c;
    input  [3:0] a;
    input        clk;
    reg      [3:0] b,c;
    always @(posedge clk)
    begin
        b = a;
        c = b;
        $display("Blocking: a = %d, b = %d, c = %d.",a,b,c);
    end
endmodule

//----- non_blocking.v -----
module non_blocking(clk,a,b,c);
    output [3:0] b,c;
    input  [3:0] a;
    input        clk;
    reg      [3:0] b,c;
    always @(posedge clk)
```

```

begin
    b <= a;
    c <= b;
    $display("Non_Blocking: a = %d, b = %d, c = %d.", a, b, c);
end

endmodule

```

测试模块源代码：

```

//----- compareTop.v -----
`timescale 1 ns/100 ps
`include "./blocking.v"
`include "./non_blocking.v"

module compareTop;

    wire [3:0] b1,c1,b2,c2;
    reg    [3:0] a;
    reg          clk;

    initial
    begin
        clk = 0;
        forever # 50 clk = ~clk; //思考：如果在本句后还有语句，能否执行？为什么？
    end

    initial
    begin
        a = 4'h3;
        $display("____");
        # 100 a = 4'h7;
        $display("____");
        # 100 a = 4'hf;
        $display("____");
        # 100 a = 4'ha;
        $display("____");
        # 100 a = 4'h2;
        $display("____");
        # 100 $display("____");
        $stop;
    end

    non_blocking non_blocking(clk,a,b2,c2);

```

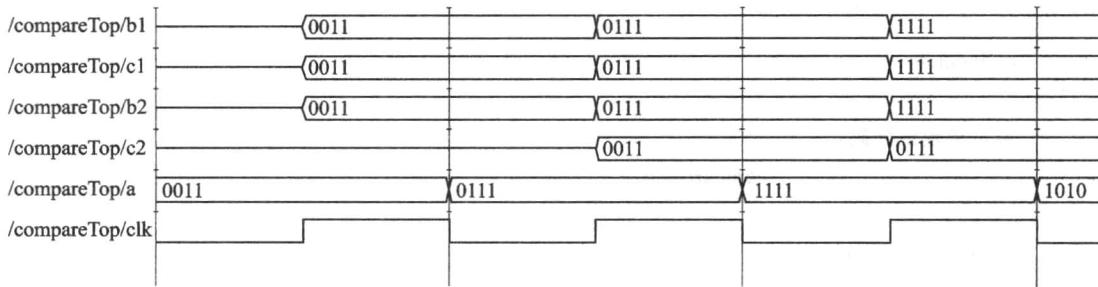
```

blocking      blocking(clk,a,b1,c1);

endmodule

```

阻塞值与非阻塞值的仿真波形(部分)如实验图4所示。



实验图4 阻塞值与非阻塞值的仿真波形

【思考题3】 在 blocking 模块中按如下两种写法, 仿真与综合的结果会有什么样的变化? 作出仿真波形, 分析综合结果。

(1) always @ (posedge clk)

```
begin
```

```
    c = b;
```

```
    b = a;
```

```
end
```

(2) always @ (posedge clk) b=a;

```
always @ (posedge clk) c=b;
```

练习五 用 always 块实现较复杂的组合逻辑电路

目的:

- (1) 掌握用 always 实现较大组合逻辑电路的方法;
- (2) 进一步了解 assign 与 always 两种组合电路实现方法的区别和注意点;
- (3) 学习测试模块中随机数的产生和应用;
- (4) 学习综合不同层次的仿真, 并比较结果。

使用 assign 结构来实现组合逻辑电路, 如果逻辑关系比较复杂, 不容易理解语句的功能。而适当地采用 always 来设计组合逻辑, 使源代码语句的功能容易理解。

下面是一个简单的指令译码电路的设计示例。该电路通过对指令的判断, 对输入数据执行相应的操作, 包括加、减、与、或和求反, 并且无论是指令作用的数据还是指令本身发生变化, 结果都要作出及时的反应。显然, 这是一个较为复杂的组合逻辑电路, 如果采用 assign 语句, 表达起来非常复杂。示例中使用了电平敏感的 always 块, 所谓电平敏感的触发条件, 是指在 @ 后的括号内电平列表中的任何一个电平发生变化(与时序逻辑不同, 它在 @ 后的括号内没有沿敏感关键词, 如 posedge 或 negedge), 就能触发 always 块的动作, 并且运用了 case 结构来进行分支判断, 不但设计思想得到直观的体现, 而且代码看起来非常整齐、便于理解。

```

//-----文件名 alu.v -----
`define plus      3'd0
`define minus     3'd1
`define band      3'd2
`define bor       3'd3
`define unegate   3'd4

module alu(out,opcode,a,b);
    output[7:0] out;
    reg[7:0]    out;
    input[2:0]   opcode;
    input[7:0]   a,b;           //操作数

always@(opcode or a or b)          //电平敏感的 always 块
begin
    case(opcode)
        'plus:  out = a+b;    //加操作
        'minus: out = a-b;    //减操作
        'band:   out = a&b;   //求 与
        'bor:    out = a|b;   //求 或
        'unegate: out=~a;    //求 反
        default:  out=8'hx;  //未收到指令时,输出任意态
    endcase
end
endmodule

```

同一组合逻辑电路分别用 always 块和连续赋值语句 assign 描述时,代码的形式大相径庭,但是在 always 中,若适当运用 default(在 case 结构中)和 else(在 if...else 结构中)语句时,通常可以综合为纯组合逻辑,尽管被赋值的变量一定要定义为 reg 型;若不使用 default 或 else 对默认项进行说明,则易生成意想不到的锁存器,这一点一定要加以注意。

指令译码器的测试模块源代码:

```

//----- alutest.v -----
`timescale 1ns/1ns
`include "../alu.v"
module alutest;
    wire[7:0] out;
    reg[7:0]  a,b;
    reg[2:0]  opcode;
    parameter times=5;
    initial
    begin
        a={ $random }%256;           //Give a radom number blongs to [0,255].
        b={ $random }%256;           //Give a radom number blongs to [0,255].
    end
endmodule

```

```

opcode=3'h0;
repeat(times)
begin
# 100    a={$random}%256; //Give a random number.
        b={$random}%256; //Give a random number.
        opcode=opcode+1;
end

# 100  $stop;

end

alu    alu(out,opcode,a,b);

endmodule

```

复杂组合逻辑电路的仿真波形(部分)如实验图 5 所示。

/alutest/out	a5	a6	0d	77
/alutest/a	24	09	0d	65
/alutest/b	81	63	8d	12
/alutest/opcode	0	1	2	3

实验图 5 复杂组合逻辑电路波形

【练习题 4】 运用 always 块设计一个 8 路数据选择器。要求:每路输入数据与输出数据均为 4 位 2 进制数,当选择开关(至少 3 位)或输入数据发生变化时,输出数据也相应地变化。

练习六 在 Verilog HDL 中使用函数

目的:

- (1) 了解函数的定义和在模块设计中的使用;
- (2) 了解函数的可综合性问题;
- (3) 了解许多综合器不能综合复杂的算术运算。

与一般的程序设计语言一样,Verilog HDL 也可使用函数以适应对不同变量采取同一运算的操作。Verilog HDL 函数在综合时被理解成具有独立运算功能的电路,每调用一次函数相当于改变这部分电路的输入以得到相应的计算结果。

下例是函数调用的一个简单示范。它采用同步时钟触发运算的执行,每个 clk 时钟周期都会执行一次运算,并且在测试模块中,通过调用系统任务 \$display 及在时钟的下降沿显示每次计算的结果。

模块源代码:

//----- 文件名 tryfunct.v -----

```

module tryfunct(clk,n,result,reset);
    output[31:0] result;
    input[3:0] n;
    input reset,clk;
    reg[31:0] result;

    always @ (posedge clk) //clk 的上升沿触发同步运算
        begin
            if (!reset) //reset 为低时复位
                result <= 0;
            else
                begin
                    result <= n * factorial(n)/((n * 2)+1);
                end //verilog 在整数除法运算结果中不考虑余数
        end

    function [31:0] factorial; //函数定义,返回的是一个 32 位的数
        input [3:0] operand; //输入只有一个 4 位的操作数
        reg [3:0] index; //函数内部计数用中间变量
        begin
            factorial = operand ? 1:0; //先定义操作数为零时函数的输出为零,不为零时为 1
            for(index = 2; index <= operand; index = index + 1)
                factorial = index * factorial; //表示阶乘的算术迭代运算
        end
    endfunction

endmodule

```

测试模块源代码：

```

`include "./tryfunct.v"
`timescale 1ns/100 ps
`define CLK_CYCLE 50

```

```

module tryfuctTop;

```

```

    reg[3:0] n,i;
    reg reset,clk;

```

```

    wire[31:0] result;

```

```

initial

```

```

begin
    clk = 0;

```

```

n=0;
reset=1;
# 100 reset=0;           //产生复位信号的负跳变沿
# 100 reset=1;           //复位信号恢复高电平后才开始输入 n
for(i=0;i<=15;i=i+1)
begin
  # 200 n=i;
end
# 100 $ stop;
end

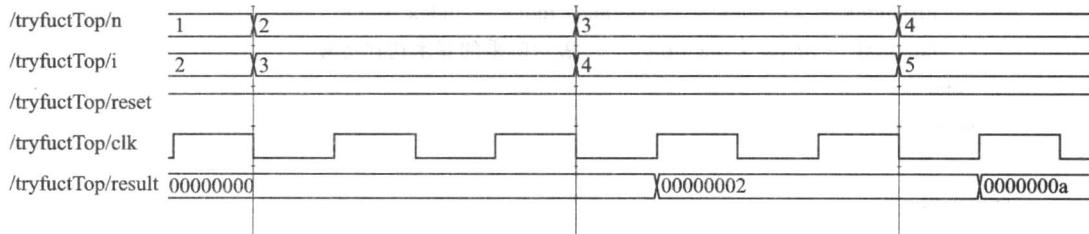
always # `clk_cycle clk=~clk;

tryfunct m(.clk(clk),.n(n),.result(result),.reset(reset));
endmodule

```

上例中函数 factorial(n)实际上就是阶乘运算。必须提醒大家注意的是,许多综合器不能综合 tryfunct.v 模块。因此,在实际可综合电路结构的设计中,要尽量避免复杂的算术运算,把复杂的运算拆分成几个步骤,通过寄存器存储中间数据,在几个时钟周期内完成。

函数调用仿真波形(部分)如实验图 6 所示。



实验图 6 函数调用仿真波形

【练习题 5】 设计一个带控制端的逻辑运算电路,分别完成正整数的平方、立方和最大数为 5 的阶乘的运算,要求可综合。编写测试模块,并给出各种层次的仿真波形,比较它们的不同。

练习七 在 Verilog HDL 中使用任务(task)

目的:

- (1) 掌握任务在 Verilog 模块设计中的应用;
- (2) 学会在电平敏感列表的 always 中使用拼接操作、任务和阻塞赋值等语句,并生成复杂组合逻辑的高级方法。

仅有函数并不能完全满足 Verilog HDL 中的运算需求。当我们希望能够将一些信号进行运算并输出多个结果时,采用函数结构就显得非常不方便,而任务结构在这方面的优势则十