第11章 验证、设计实例和Verilog综合

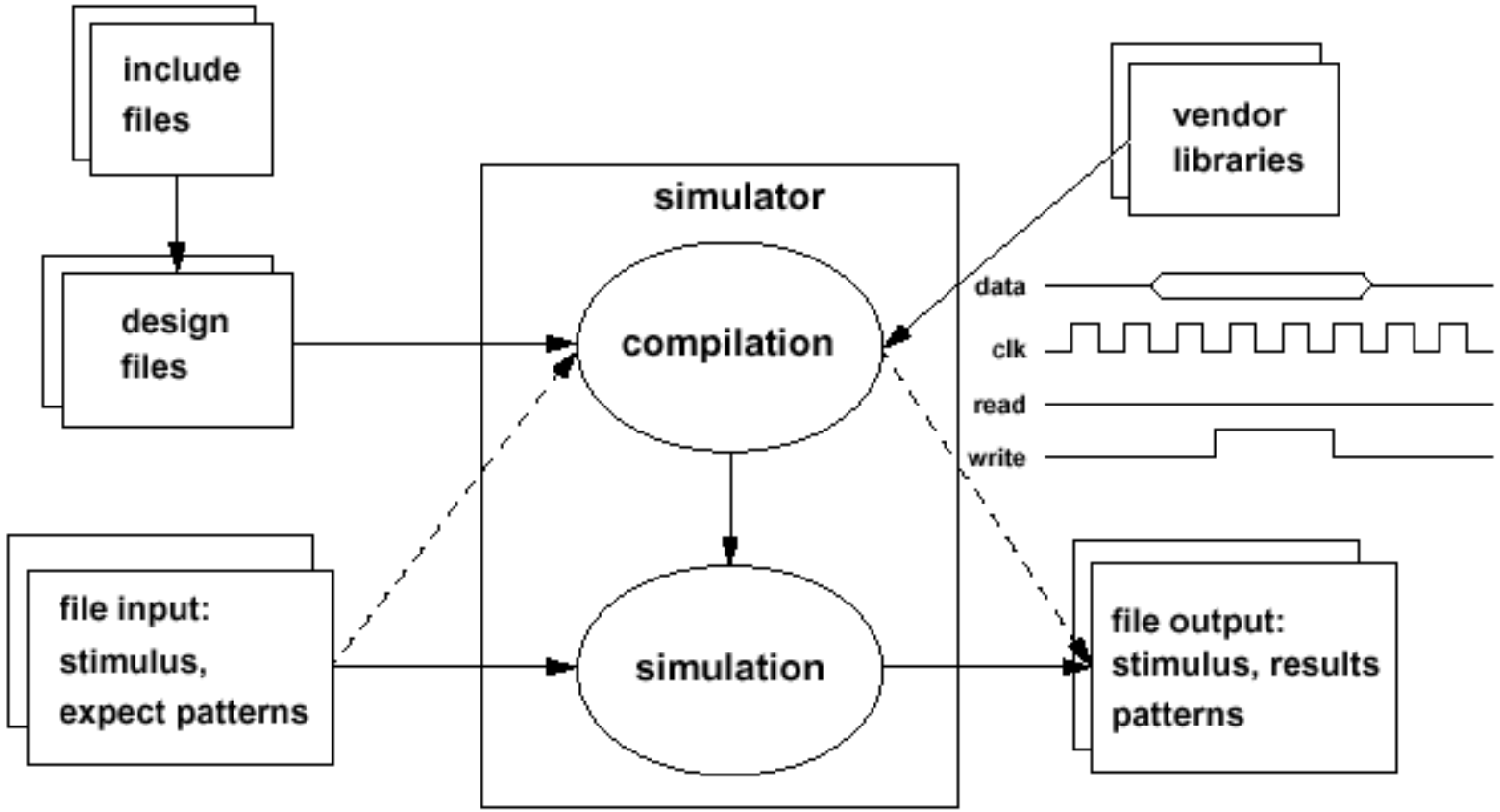
西安交大电信学院微电子学系
程　军
jcheng@mail.xjtu.edu.cn

设计 验证——Verilog TestBench

学习：

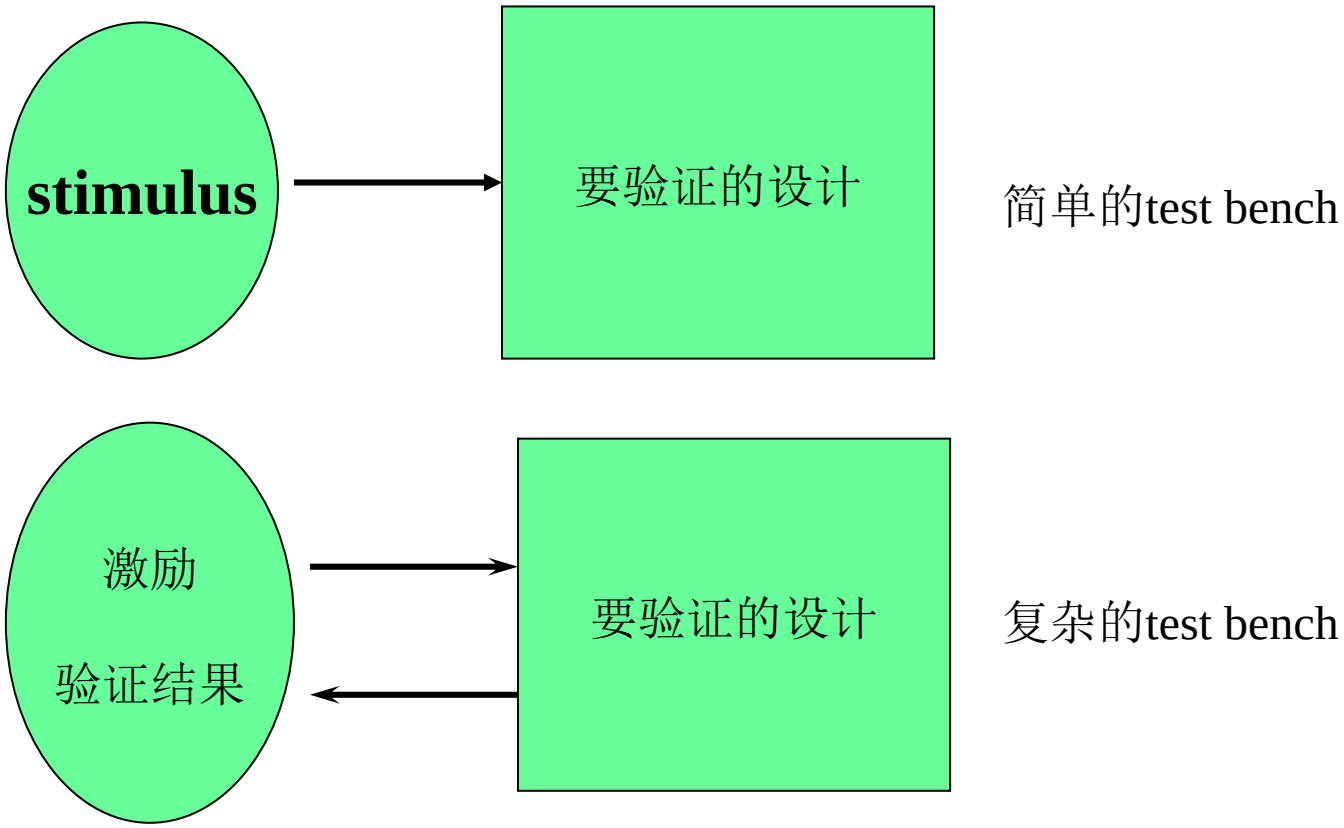
- 用一个复杂的**test bench**复习设计的组织与仿真
- 建立**test bench**通常使用的编码风格及方法

设计组织



虚线表示编译时检测输入文件是否存在及可读并允许生成输出文件。

test bench组织



- 简单的**test bench**向要验证的设计提供向量，人工验证输出。
- 复杂的**test bench**是自检测的，其结果自动验证。

Testbench程序构成

```
module Test_Bench;
    //通常测试验证程序没有输入和输出端口。
    Local_reg_and_net_declarations
    Generate_waveforms_using_initial_&_always_statements
    Instantiate_module_under_test
    Monitor_output_and_compare_with_expected_values
endmodule
```

- 测试平台的作用
  - 产生仿真的激励(波形)；
  - 将激励施加到被测的模块并收集其输出响应；
  - 将输出响应与期望值进行对比；**(option)**

波形产生——并行块

- **fork…join**块在测试文件中很常用。他们的并行特性使用户可以说明绝对时间，并且可以并行的执行复杂的过程结构，如循环或任务。

```
module inline_ tb;
   reg [7: 0] data_ bus;
   // instance of DUT
   initial    fork
              data_bus = 8'b00;
      #10 data_bus = 8'h45;
      #20 repeat (10) #10 data_bus = data_bus + 1;
      #25 repeat (5) #20 data_bus = data_bus << 1;
      #140 data_bus = 8'h0f;
   join
endmodule
```

Time	data_ bus
0	8'b0000_0000
10	8'b0100_0101
30	8'b0100_0110
40	8'b0100_0111
45	8'b1000_1110
50	8'b1000_1111
60	8'b1001_0000
65	8'b0010_0000
70	8'b0010_0001
80	8'b0010_0010
85	8'b0100_0100
90	8'b0100_0101
100	8'b0100_0110
105	8'b1000_1100
110	8'b1000_1101
120	8'b1000_1110
125	8'b0001_1100
140	8'b0000_1111

上面的两个**repeat**循环从不同时间开始，并行执行。象这样的特殊的激励集在单个的**begin…end**块中将很难实现。

包含文件

- 包含文件用于读入代码的重复部分或公共数据。

```
module clk_gen (clk);
output clk; reg clk;
`include "common.txt"
initial begin
   while ($ time < sim_end)
   begin
        clk = initial_clock;
      #(period/2)  clk = !initial_clock;
      #(period/2);
   end
   $finish;
end
endmodule
```

```
// common. txt
// clock and simulator constants
parameter initial_clock = 1;
parameter period = 15;
parameter max_cyc = 100;
parameter sim_end = period * max_cyc
```

在上面的例子中，公共参数在一个独立的文件中定义。此文件在不同的仿真中可被不同的测试文件调用。

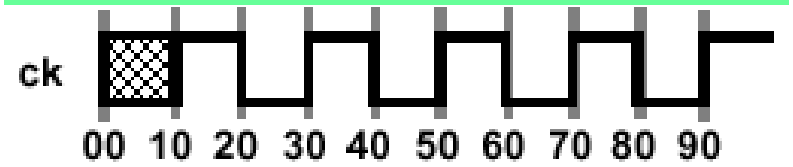
建立时钟

时钟通常在测试基准中建立，是同步电路基本的信号。

下面介绍如何产生不同的时钟波形。同时给出用门级和行为级描述方法

例1：下面是一个简单对称时钟的例子

产生的波形（假定period为20）




```
reg ck;
always begin
    #( period/2) ck = 1;
    #( period/2) ck = 0;
end
```

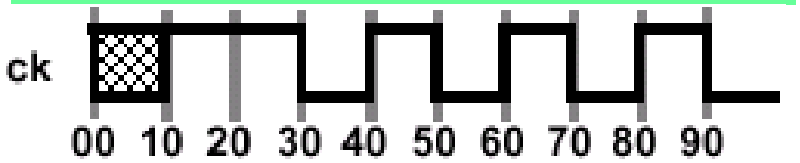
```
reg go; wire ck;
nand #( period/2) u1 (ck, ck, go);
initial begin
                    go = 0;
    #( period/2) go = 1;
end
```

注意：在一些仿真器中，时钟与设计使用相同的抽象级描述时，仿真性能会好一些。

建立时钟(续1)

例2：有启动延时的对称时钟的例子：

产生的波形（假定period为20）




```
reg ck;
initial  begin
        ck = 1;
        #( period)
        forever
            #( period/2) ck = !ck;
end
```

```
reg go;  wire ck;
nand #( period/2) u1 (ck, ck, go);
initial
begin
            go = 0;
            #(period) go = 1;
end
```

注意：在行为描述中，在时间0将CK初始化为0；而在结构描述中，直到period/2才影响CK值。当go信号在时间0初始化时，CK值到period/2才变化。可以使用特殊命令force和release立即影响CK值。

建立时钟(续2)

例3：含不规则启动延时的不对称时钟的例子

产生的波形（假定period为20）


```
reg ck;
initial begin
    #(period + 1) ck = 1;
    #(period/2 – 1);
    forever begin
        #(period/4) ck = 0;
        #(3*period/4) ck = 1;
    end
end
```

```
reg go;  wire ck;
nand #(3*period/4, period/4)   u1(ck, ck, go);
initial begin
    #(period/4 + 1) go = 0;
    #(5*period/4 – 1) go = 1;
end
```

注意：在行为描述中，CK值立刻被影响；而在结构描述中，在传输延时后才输出正确波形。

产生指定波形的激励

产生激励并加到设计有很多 种方法。一些常用的方法有：

- 从一个**initial**块中施加线激励
- 从一个循环或**always**块施加激励
- 从一个向量或整数数组施加激励
- 记录一个仿真过程，然后在另一个仿真中回放施加激励

线性激励

- 线性激励有以下特性：
  - ✓ 只有变量的值改变时才列出
  - ✓ 易于定义复杂的时序关系
  - ✓ 对一个复杂的测试，测试基准(test bench)可能非常大

```
module inline_ tb;
   reg [7: 0] data_ bus, addr;
   wire [7: 0] results;
   DUT u1 (results, data_ bus, addr);
   initial
      fork
            data_bus = 8'h00;
            addr = 8'h3f;
        #10 data_ bus = 8'h45;
        #15 addr = 8'hf0;
        #40 data_ bus = 8'h0f;
        #60 $finish;
      join
endmodule
```

循环激励

- 从循环产生激励有以下特性：
  - ✓ 在每一次循环，修改同一组激励变量
  - ✓ 时序关系规则
  - ✓ 代码紧凑

```
module loop_tb;
    reg clk;
    reg [7:0] stimulus;
    wire [7:0] results;
    integer i;
    DUT u1 (results, stimulus);
    always begin // clock generation
        clk = 1;  #5;  clk = 0;  #5;
    end
    initial begin
        for (i = 0; i < 256; i = i + 1)
       @( negedge clk)  stimulus = i;
       #20 $finish;
   end
endmodule
```

数组激励

- 从数组产生激励有以下特性：
  - ✓ 在每次反复中，修改同一组激励变量
  - ✓ 激励数组可以直接从文件中读取

```
module array_ tb;
   reg [7: 0] data_ bus, stimulus, stim_ array[ 0: 15]; // 数组
   integer i;
   DUT u1 (results, stimulus);
   initial begin
      // 从数组读入数据
      #20 stimulus = stim_array[0];
      #30 stimulus = stim_array[15]; // 线性激励
      #20 stimulus = stim_array[1];
      for (i = 14; i > 1; i = i - 1) // 循环
         #50 stimulus = stim_array[i] ;
      #30 $finish;
   end
endmodule
```

矢量采样

- 在仿真过程中可以对激励和响应矢量进行采样，作为其它仿真的激励和期望结果。

```
module capture_tb;
   parameter period = 20
   reg [7:0] in_vec, out_vec;
   integer RESULTS, STIMULUS;
   DUT u1 (out_ vec, in_ vec);
   initial
      begin
         STIMULUS = $fopen("stimulus. txt") ;
         RESULTS = $fopen("results. txt") ;
         fork
            if (STIMULUS != 0 ) forever #( period/2)
               $fstrobeb (STIMULUS, "%b", in_vec);
            if (RESULTS != 0 ) #( period/2) forever #( period/2)
               $fstrobeb (RESULTS, "%b", out_vec);
         join
   end
endmodule
```

矢量回放

- 保存在文件中的矢量反过来可以作为激励

```
module read_file_tb;
   parameter num_vecs = 256;
   reg [7:0] data_bus;
   reg [7:0] stim [num_vecs-1:0];
   integer i;
   DUT u1 (results, data_bus)
   initial
      begin // Vectors are loaded
         $readmemb ("vec. txt", stim);
         for (i =0; i < num_vecs ; i = i + 1)
         #50 data_bus = stim[i];
   end
endmodule
```

```
// 激励文件vec.txt
00111000
00111001
00111010
00111100
00110000
00101000
00011000
01111000
10111000
.
.
```

- 使用矢量文件输入/输出的优点：
  - 激励修改简单
  - 设计反复验证时直接使用工具比较矢量文件。

错误及警告报告

- 使用文本或文件输出类的系统任务报告错误及警告

```
always @( posedge par_err)
    $display (" error-bus parity errors detected");
always @( posedge cor_err)
    $display("warning-correctable error detected");
```

- 一个更为复杂的**test bench**可以：
  - 不但能报告错误，而能进行一些动作，如取消一个激励块并跳转到下一个激励。
  - 在内部保持错误跟踪，并在每次测试结束时产生一个错误报告。

强制激励——assign/deassign, force/release

- 在过程块中，可以用两种连续赋值语句驱动一个值或表达式到一个信号。
  - 过程连续赋值通常不可综合，所以它们通常用于测试基准描述。
  - 对每一种连续赋值，都有对应的命令停止信号赋值。
  - 不允许在赋值语句内部出现时序控制。
- 对一个寄存器使用**assign**和**deassign**，将覆盖所有其他在该信号上的赋值。这个寄存器可以是**RTL**设计中的一个节点或测试基准中在多个地方赋值的信号等。

```
initial begin
    #10 assign  top.dut.fsm1.state_reg = `init_state ;
    #20 deassign  top.dut.fsm1.state_reg ;
end
```

- 在**register**和**net**上（例如一个门级扫描寄存器的输出）使用**force**和**release**，将覆盖该信号上的所有其他驱动。

```
initial begin
    #10 force top. dut. counter. scan_ reg. q = 0 ;
    #20 release top. dut. counter. scan_ reg. q ;
end
```

强制激励(续1)

在上面两个例子中，在 net或register上所赋的常数值，覆盖所有在时刻10和时刻20之间可能发生在该信号上的其他任何赋值或驱动。如果所赋值是一个表达式，则该表达式将被连续计算。

- force/release可以作用于一个信号的位选择、部分选择或连接，但位的指定不能是一个变量（例如out_vec[i]）
- 不能对寄存器的位选择或部分选择使用assign和deassign
- 对同一个信号，force覆盖assign。
- 后面的assign或force语句覆盖以前相同类型的语句。
- 如果对一个信号先assign然后force，它将保持force值。在对其进行release后，信号为assign值。
- 如果在一个信号上force多个值，然后release该信号，则不出现任何force值。

使用task

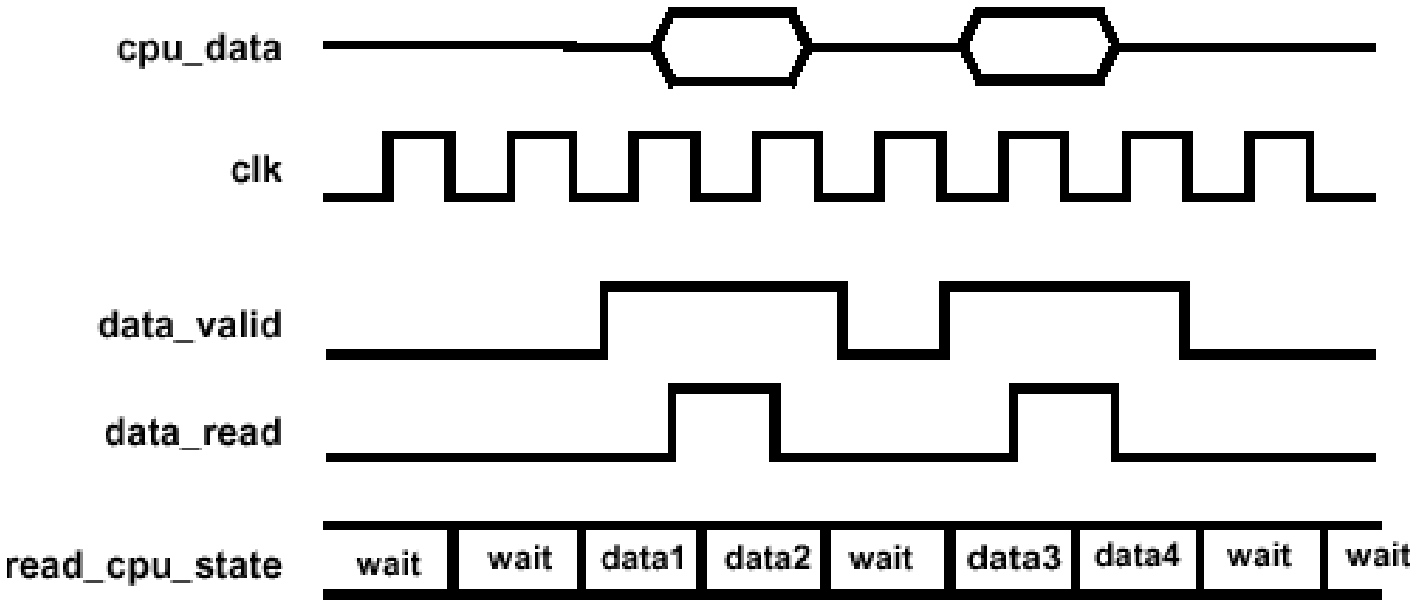
在test bench中使用task可以压缩重复操作，提高代码效率。

```
module bus_ctrl_tb;
    reg [7: 0] cpu_data;
    reg data_valid, data_read;
    cpu u1 (data_valid, cpu_data, data_read);
    initial begin
        cpu_driver (8'b0000_0000);
        cpu_driver (8'b1010_1010);
        cpu_driver (8'b0101_0101);
    end
```

```
task cpu_driver;
input [7:0] data_in;
    begin
        #30 data_valid = 1;
            wait (data_read == 1);
        #20 cpu_data = data_ in;
            wait (data_read == 0);
        #20 cpu_data = 8'hzz;
        #30 data_valid = 0;
    end
 endtask
```

使用task(续1)

产生的波形



设计实例—组合逻辑电路

- 2—4译码器，数据流描述

```
module Decoder2x4(A, B, EN, Z);
    input A, B, EN;
    output [0:3] Z;
    wire Abar, Bbar;
    assign #1 Abar = ~A;
    assign #1 Bbar = ~B;
    assign #2 Z[0] = ~(Abar & Bbar & EN);
    assign #2 Z[1] = ~(Abar & B & EN);
    assign #2 Z[2] = ~(A & Bbar & EN);
    assign #2 Z[3] = ~(A & B & EN);
endmodule
```

设计实例—组合逻辑电路(续)

- 2—4译码器，行为描述

```
module Decoder2x4(A, B, EN, Z);
    input A, B, EN;
    output [0:3] Z; reg [0:3] Z;
    always@(A or B or EN) begin
        if(!EN) Z=4'b1111;
        else
            case({A,B})
              2'b00: Z=4'b1110;
              2'b01: Z=4'b1101;
              2'b10: Z=4'b1011;
              2'b11: Z=4'b0111;
            endcase
     end
endmodule
```

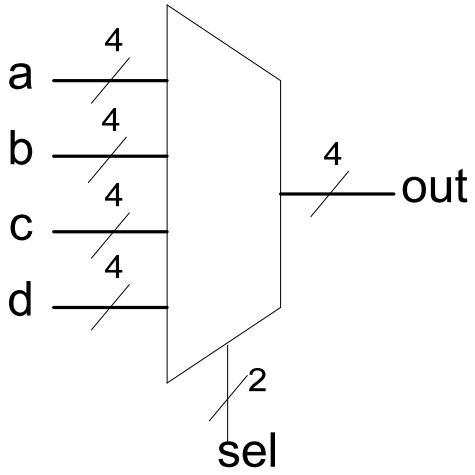
```
always@(A or B or EN)
begin
    Z[3] = ~(~A & ~B & EN);
    Z[2] = ~(~A & B & EN);
    Z[1] = ~(A & ~B & EN);
    Z[0] = ~(A & B & EN);
end
```

设计实例—组合逻辑电路(续)

- 2—4译码器，结构描述

```
module Decoder2x4(A, B, EN, Z);
    input A, B, EN;
    output [0:3] Z;
    wire Abar, Bbar;
    not #(1, 2)
          V0 (Abar, A),
          V1 (Bbar, B);
    nand #(4, 3)
         N0 (Z[3], EN, A, B),
         N1 (Z[0], EN, Abar, Bbar),
         N2 (Z[1], EN, Abar, B),
         N3 (Z[2], EN, A, Bbar);
endmodule
```

设计实例(续)



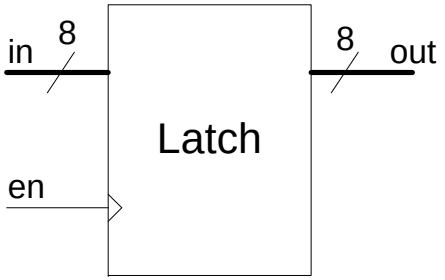
- 多路选择器

```
module mux4(out, a, b, c, d, sel);
  output [3:0] out; reg [3:0] out;
  input [3:0] a, b, c, d;
  input [1:0] sel;
  always@(a or b or c or d or sel)
      case(sel)
         2'b00: out = a;
         2'b01: out = b;
         2'b10: out = c;
         2'b11: out = d;
         default: out = 4'b0;
      endcase
endmoule
```

设计实例(续)

- 锁存器：

```
module dlatch (out, in, en);
  input [7:0] in;
  output [7:0] out;
  input en;
//方法1，连续赋值
  wire [7:0] out;
  assign out = en? in : out;
endmodule
```




```
//方法2：过程赋值
reg [7:0] out;
always@(in or en)
    if(en) out=in;

//方法3：过程连续赋值
reg [7:0] out;
always@(en)
    if(en) assign out=in;
    else deassign out;
```

设计实例—时序电路

- 普通DFF

```
module dff(q, data, clk);
    output q;
    input data, clk;
    reg q;
    always@(posedge clk)
    begin
        q <= data;  //时序电路使用非阻塞赋值
    end
endmodule
```

设计实例—时序电路(续)

- 异步清零的DFF

```
module dffr_asyn(q, data, reset, clk);
    input data, reset, clk;
    output q;  reg q;
    always@(posedge clk or posedge reset)
    begin
        if(reset) q<=0;
        else  q<= data;
    end
endmodule
```

设计实例—时序电路(续)

- 同步清零的DFF

```
module dffr_syn(q, data, reset, clk);
    input data, reset, clk;
    output q;  reg q;
    always@(posedge clk)
    begin
        if(reset) q<=0;
        else  q<= data;
    end
endmodule
```

设计实例—时序电路(续)

- 移位寄存器

```
module shift(dout, din, clr, clk);
    input din, clr, clk;
    output [7:0] dout;
    reg [7:0] dout;
    always@(posedge clk) begin
        if(!clr) dout <= 8'b0;
        else begin
            dout <= {dout[6:0], din};
        end
    end
endmodule
```

练习：修改设计，使din为并行数据输入，当load信号有效时，加载din，load无效时，每个clk上升沿，LR位1时左移，LR为0时右移。

移位寄存器

```
module shiftlr(dout, din, load, clr, lr, clk);
  input [7:0] din;
  input clr, lr, clk;
  output [7:0] dout;
  reg [7:0] dout;

  always@(posedge clk) begin
    if(!clr) dout<=0;
    else if(load) dout<=din;
    else if(lr) dout<=dout<<1;
    else dout<=dout>>1;
  end
endmodule
```

设计实例—时序电路(续)

- 二分频电路

```
module freq_div2(rst, clk, clk_div2);
input rst, clk;
output  reg clk_div2;
  always@(posedge clk, negedge rst) begin
     if(~rst ) clk_div2 <= 1'b0;
     else clk_div2 <= ~clk_div2;
  end
endmodule
```

设计实例—时序电路(续)

- 四分频电路(同步逻辑)

```
module freq_div4(rst, clk, clk_div4);
input rst, clk;
output  clk_div4;
reg t1, clk_div4;
  always@(posedge clk, negedge rst) begin
     if(~rst ) begin clk_div4 <= 1'b0;
        t1<=1'b0;
     end
     else begin clk_div4 <= t1;
         t1<= ~clk_div4;
     end
  end
endmodule
```

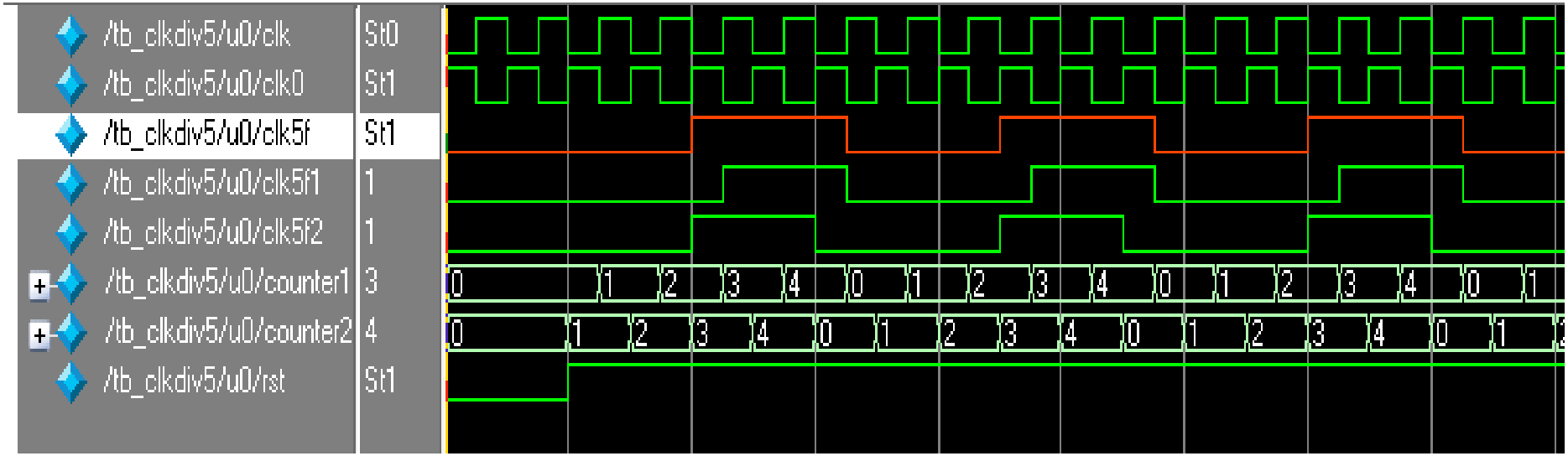
设计实例—时序电路(续)

- N分频电路???
- 思考：5分频电路和16分频电路的Verilog代码

5分频电路代码

```
module d5f(clk,rst,clk5f);
  input clk, rst;
  output clk5f;
  wire clk0;
   wire clk5f;
  reg [2:0] counter1, counter2;
  reg clk5f1, clk5f2;
  assign clk0 = ~clk;
 always@(posedge clk or negedge rst)
  if(!rst) counter1 <=0;
  else if(counter1==3'b100)
      counter1<=0;
  else counter1 <= counter1 + 1;
always@(posedge clk or negedge rst)
begin
   if(!rst) clk5f1 <=0;
   else if((counter1==3'b100)||
            (counter1==3'b010))
      clk5f1 <= ~clk5f1;
end
always@(posedge clk0 or negedge rst)
  if(~rst) counter2<=0;
  else if(counter2== 3'b100)
     counter2<=0;
  else counter2<=counter2 + 1;
always@(posedge clk0 or negedge rst)
   if(!rst) clk5f2 <= 0;
   else if((counter2==3'b100)||
             (counter2==3'b010))
     clk5f2 <= ~clk5f2;
assign clk5f = clk5f1 | clk5f2;
endmoudule
```

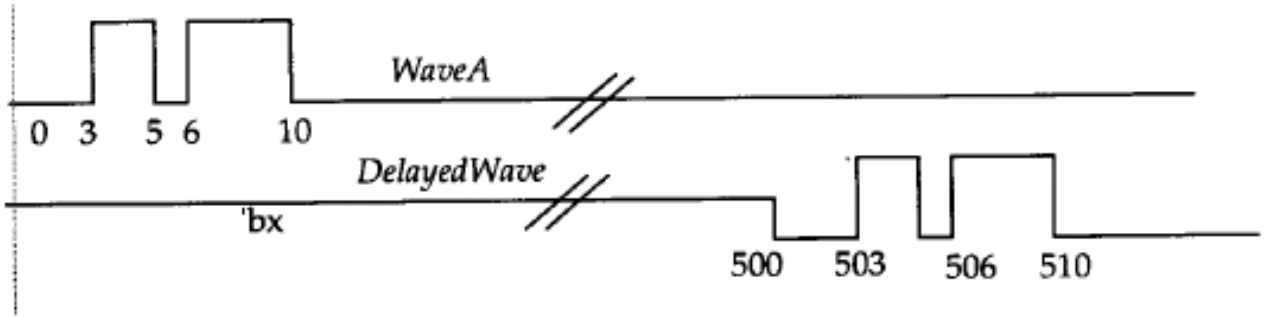
5分频电路仿真结果


/tb_clkdiv5/u0/clk
/tb_clkdiv5/u0/clk0
/tb_clkdiv5/u0/clk5f
/tb_clkdiv5/u0/clk5f1
/tb_clkdiv5/u0/clk5f2
/tb_clkdiv5/u0/counter1
/tb_clkdiv5/u0/counter2
/tb_clkdiv5/u0/rst
St0
St1
St1
1
1
3
4
St1

传输延时建模

- 惯性延时
  - 连续信号赋值语句和门原语中的延时
- 带语句内延时的非阻塞赋值

```
module Transport (WaveA, DelayedWave);
    parameter T_DELAY = 500;
    input WaveA;    output DelayedWave;
    reg DelayedWave;
    always
      @(WaveA) DelayedWave <= #T_DELAY WaveA;
endmodule
```



可综合的代码中阻塞赋值与非阻塞赋值的使用准则

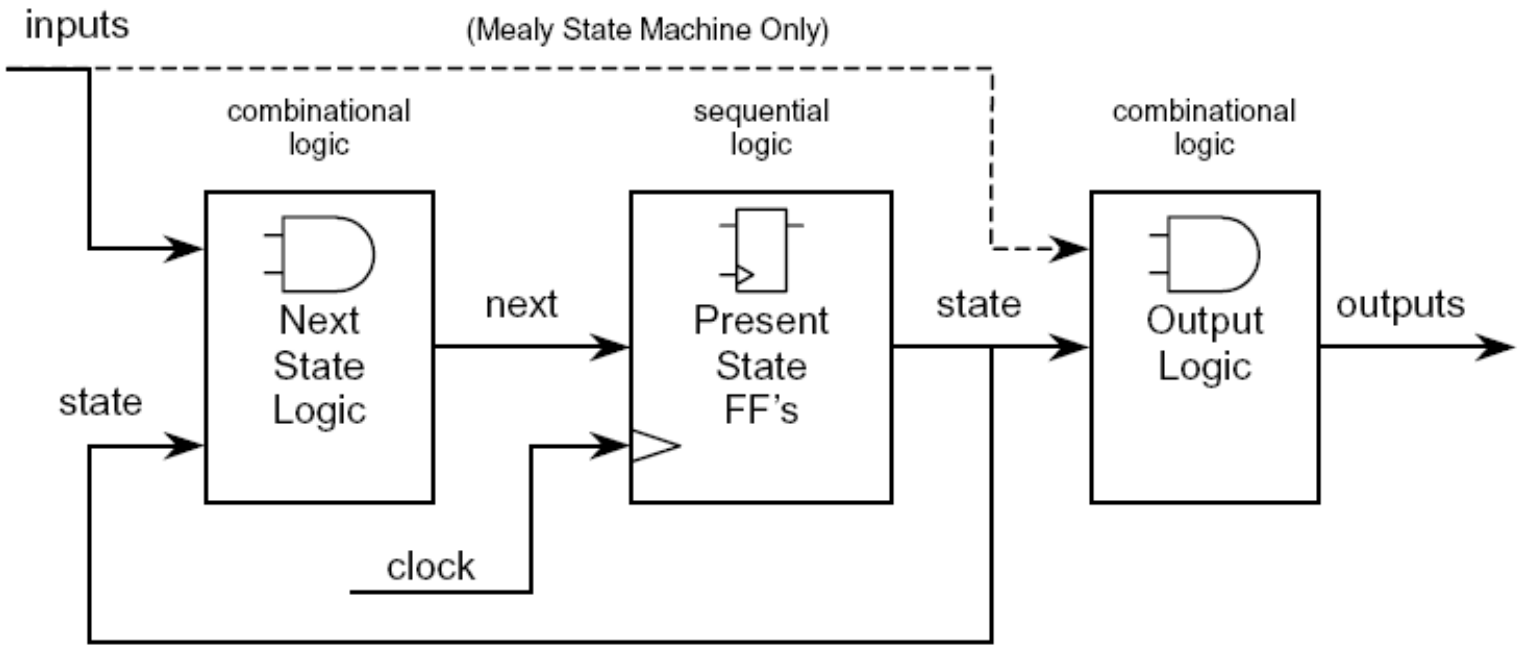
1. 当为时序逻辑建模，使用“非阻塞赋值”。
2. 当为锁存器（latch）建模，使用“非阻塞赋值”。
3. 当用always块为组合逻辑建模，使用“阻塞赋值”。
4. 当在同一个always块里面既为组合逻辑又为时序逻辑建模，使用“非阻塞赋值”。
5. 不要在同一个always块里面混合使用“阻塞赋值”和“非阻塞赋值”。
6. 不要在两个或两个以上always块里面对同一个变量进行赋值。
7. 使用$strobe显示已被“非阻塞赋值”的值。
8. 不要使用＃0延迟的赋值。

有限状态机(FSM)设计

- moore FSM和Mealy FSM

Moore FSM的输出只与状态有关，与输入无关

Mealy FSM的输出与输入和状态同时有关。

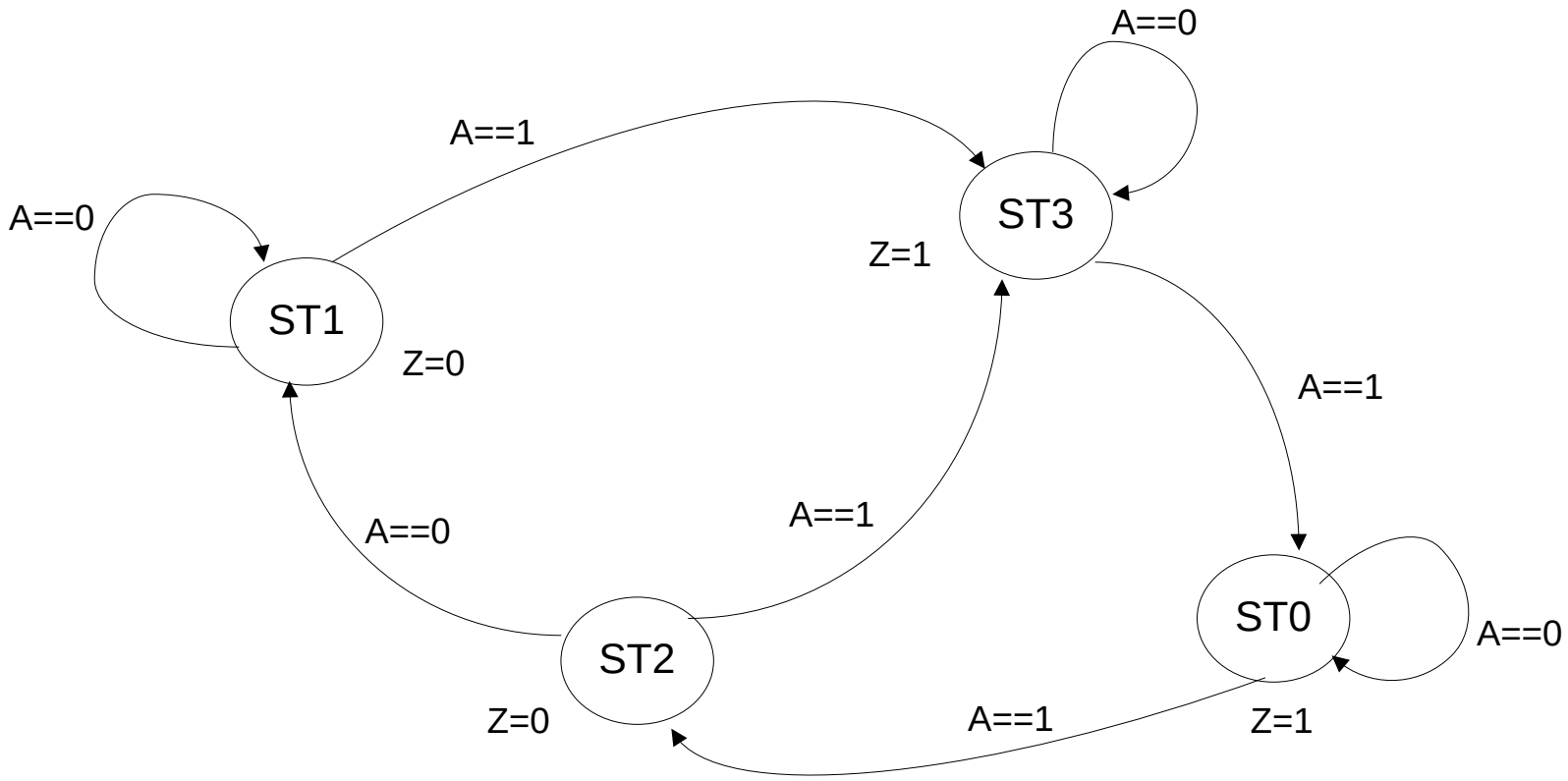


有限状态机(FSM)设计

- 编码方式
  - 二进制码；
  - 格雷码(gray-code)；
  - 独热码(one-hot，one-hot with zero idle)；
- 代码风格
  - 两个always语句；
  - 一个always语句；
  - 非阻塞赋值描述时序逻辑，阻塞赋值描述组合逻辑；
  - 组合逻辑的默认值：x、0、IDLE

有限状态机(FSM)设计

- 示例：moore状态机

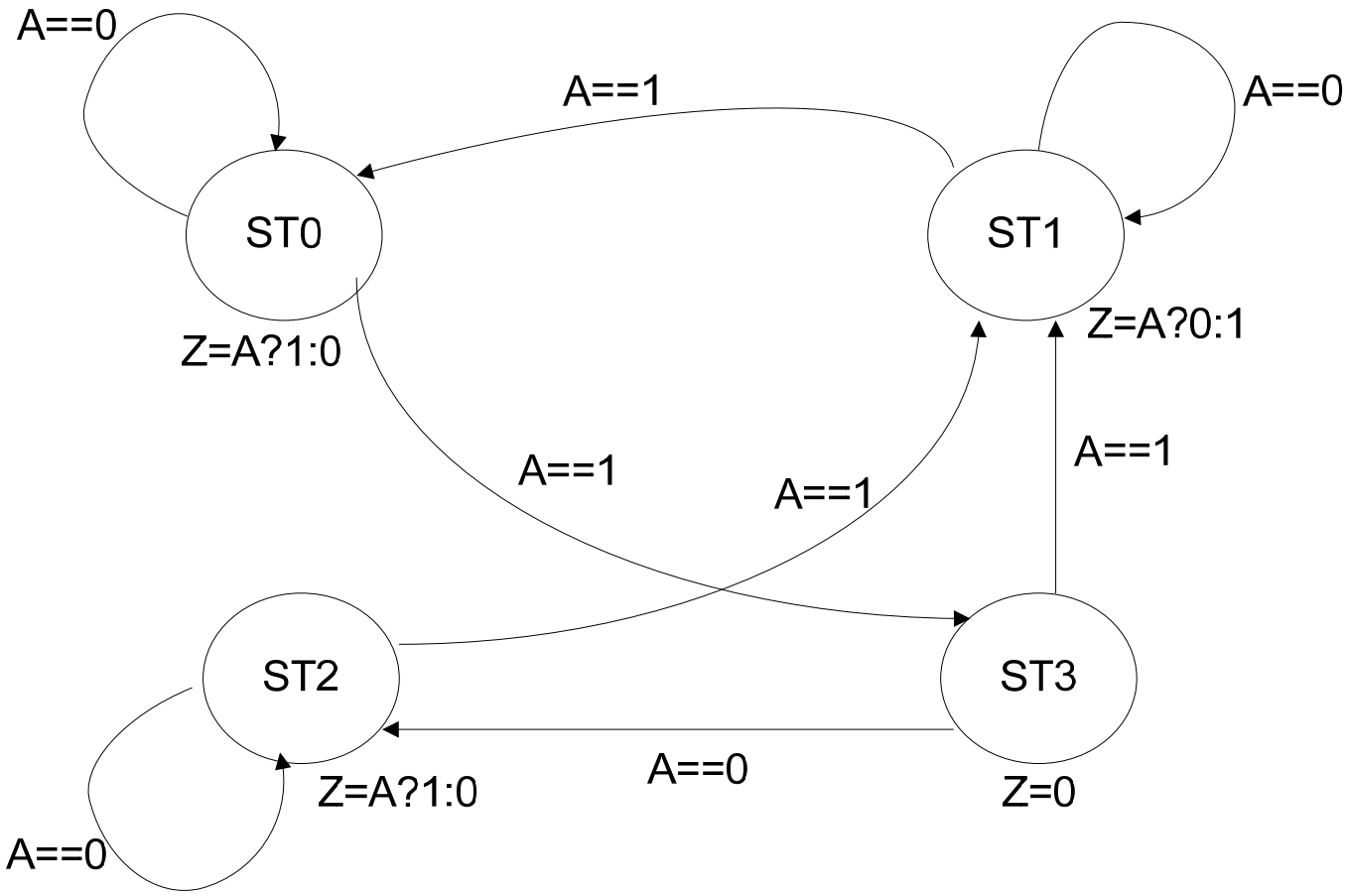


有限状态机(FSM)设计(续)

```
module Moore_FSM(A, clock, Z);
    input A, clock;
    output Z; reg Z;
    parameter ST0=0, ST1=1, ST2=2, ST3=3;
    reg [0:1] Moore_State;
    always@(posedge clock)
        case(Moore_State)
          ST0: begin Z<=1; if(A) Moore_State<=ST2; end
          ST1: begin Z<=0; if(A) Moore_State<=ST3; end
          ST2: begin Z<=0;
                       if(~A) Moore_State<=ST1;
                       else Moore_State<=ST3;
                end
          ST3: begin Z<=1; if(A) Moore_State<=ST0; end
        endcase
endmodule
```

有限状态机(FSM)设计(续)

- Mealy FSM

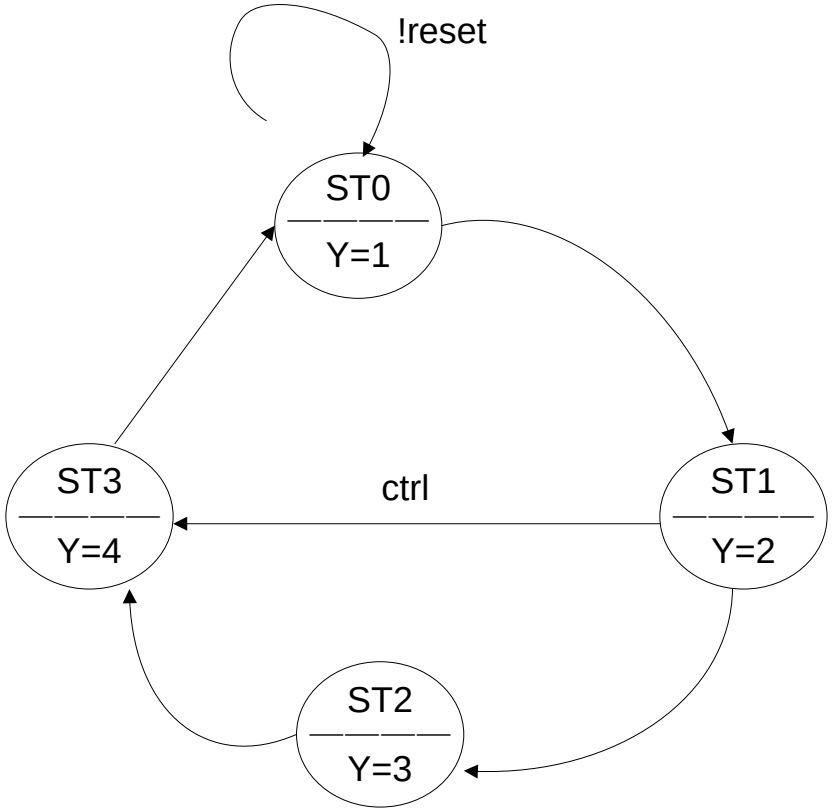


有限状态机(FSM)设计(续)

```
module Mealy_FSM(A, clock, Z);
   input A, clock;    output Z; reg Z;
   parameter ST0=0, ST1=1, ST2=2, ST3=3;
   reg [1:0] C_State, N_State;
   always@(negedge clock) C_State<=N_State;
   always@(C_State or A) begin
      case (C_State)
        ST0: if(A) begin Z=1; N_State=ST3; end
                else Z=0;
        ST1: if(A) begin Z=0; N_State=ST0; end
               else Z=1;
        ST2: if(~A) Z=0;
               else begin Z=1; N_State=ST1;end
        ST3: begin Z=0;
                        if(~A) N_State=ST2; else N_State=ST1;
               end
      endcase
   end
endmodule
```

有限状态机(FSM)设计(练习)

使用3个不同的always语句分别处理当前状态(CS)、下一状态(NS)和输出逻辑(OL)。




```
module fsm1(clk, reset, ctrl, y);
  input clk, reset, ctrl;
  output y;
  integer y;
  ......
  ......
endmodule
```

有限状态机(FSM)设计(练习)

```
module fsm1(clk, reset, ctrl, y);
  input clk, reset, ctrl;
  output y; integer y;
  reg [1:0] CS, NS;
  parameter ST0=0, ST1=1, ST2=2, ST3=3;
  always@(posedge clk or negedge reset)
  begin
    if(!reset) CS<=ST0;
    else CS<=NS;
  end
  always@(ctrl or CS) begin
    case(CS)
        ST0: NS=ST1;
        ST1: if(ctrl) NS=ST3;
                 else NS=ST2;
        ST2: NS=ST3;
        ST3: NS=ST0;
    endcase
  always@(CS)
    case(CS)
        ST0: y=1;
        ST1: y=2;
        ST2: y=3;
        ST3: y=4;
    endcase
endmodule
```

练习：

- 设计一个加3减5计数器，8位输出，带奇偶校验输出，带进位/借位标志。

```
module countdn_up(rst, clk, up, dn, din, dout, par, carry, borrow);
  input rst, clk, up, dn; input [7:0] din;
  output [7:0] dout;
  output par, carry, borrow;
  ......
endmodule
```

- 当rst为低电平时，dout, par, carry, borrow复位
- up, dn为00时，dout从din加载数据；
- up, dn为01时，dout作减5计数；
- up, dn为10时，dout作加3计数；
- up, dn为11时，dout保持不变；
- par在clk的上升沿输出dout的奇偶校验，carry为进位，borrow为借位，同样在clk上升沿同步输出。

```
module countu3d5(rst, clk, up, dn, din, dout, par, carry, borrow);
 input rst, clk, up, dn;
 input [7:0] din;
 output [7:0] dout;
 output par, carry, borrow;
 reg [7:0] dout;
 reg par, carry, borrow;
 wire [8:0] cnt_up, cnt_dn;
 reg [7:0] cnt_nxt;
     assign  cnt_dn = dout - 3'b101;
     assign  cnt_up = dout + 2'b11;
     always@(up or dn or din or cnt_dn or cnt_up)
       case ({up,dn})
            2'b 00 : cnt_nxt = din;
            2'b 01 : cnt_nxt = cnt_dn;
            2'b 10 : cnt_nxt = cnt_up;
            2'b 11 : cnt_nxt = cnt_nxt;
       endcase
```

```
always @(posedge clk or negedge rst)
    begin
     if(!rst) begin
        dout<=0;
        par<=0;
        carry<=0;
        borrow<=0;
     end
     else begin
        par <= ^cnt_nxt;
        carry <= up & cnt_up[8];
        borrow <= dn & cnt_dn[8];
        dout <= cnt_nxt;
     end
    end
endmodule
```

存储器(ROM/RAM)建模

学习内容：

- 如何描述存储器
- 如何描述双向端口

存储器件建模

描述存储器必须做两件事：

- 说明一个适当容量的存储器。
- 提供内容访问的控制，例如：
  - ✓只读
  - ✓读和写
  - ✓写同时读
  - ✓多个读操作，同时进行单个写操作
  - ✓同时有多个读和多个写操作，有保证一致性的方法

简单ROM描述

下面的ROM描述中使用二维寄存器组定义了一个存储器mem。ROM的数据单独保存在文件my_rom_data中，如右边所示。通常用这种方法使ROM数据独立于ROM描述。

```
`timescale 1ns/10ps
module myrom (read_data, addr, read_en_ );
   input read_en_;
   input [3:0] addr;
   output [3:0] read_data;
   reg [3:0] read_data;
   reg [3:0] mem [0:15];
   initial
      $readmemb ("my_rom_data", mem);
   always @( addr or read_en_)
      if (! read_en_)
         read_data = mem[addr];
endmodule
```

```
my_rom_data
0000
0101
1100
0011
1101
0010
0011
1111
1000
1001
1000
0001
1101
1010
0001
1101
```

简单的RAM描述

RAM描述比ROM略微复杂，因为必须既有读功能又有写功能，而读写通常使用同一数据总线。这要求使用新的处理双向数据线的建模技术。在下面的例子中，若读端口未使能，则模型不驱动数据总线；此时若数据总线没有写数据驱动，则总线为高阻态Z。这避免了RAM写入时的冲突。

```
`timescale 1ns /1ns
module mymem (data, addr, read, write);
    inout [3:0] data;
    input [3:0] addr;
    input read, write;
    reg [3:0] memory [0:15]; // 16*4
// 读
    assign data = read ? memory[addr] : 4'bz;
// 写
    always@(write or data or addr)
        if(write) memory[addr] = data;
endmodule
```

参数化存储器描述

在下面的例子中，给出如何定义一个字长和地址均参数化的只读存储器件。

```
module scalable_ROM (mem_word, address);
    parameter addr_bits = 8;  // 地址总线宽度
    parameter wordsize = 8;   // 字宽
    parameter words = (1 << addr_bits); // mem容量
    output [wordsize:1] mem_word; // 存储器字
    input [addr_bits:1] address; // 地址总线
    reg [wordsize:1] mem [0 : words-1]; // mem声明
// 输出存储器的一个字
    wire [wordsize:1] mem_word = mem[address];
endmodule
```

例中存储器字范围从0而不是1开始，因为存储器直接用地址线确定地址。也可以用下面的方式声明存储器并寻址。

```
reg [wordsize:1] mem [1:words]; // 从地址1开始的存储器
// 存储器寻址时地址必须加1
wire [wordsize:1] mem_word = mem[ address + 1];
```

存储器数据装入

可以使用循环或系统任务给存储器装入初始化数据

- 用循环给存储器的每个字赋值

```
for (i= 0; i < memsize; i = i+ 1) // initialize memory
    mema[ i] = {wordsize{ 1'b1}};
```

- 调用系统任务$readmem

```
$readmemb("mem_file. txt", mema);
```

可以用 系统任务$readmem给一个ROM或RAM加载数据。对于ROM，开始时写入的数据就是其实际内容。对于RAM，可以通过初始化，而不是用不同的写周期给每个字装入数据以减少仿真时间。

使用双向端口

用关键词**inout**声明一个双向端口

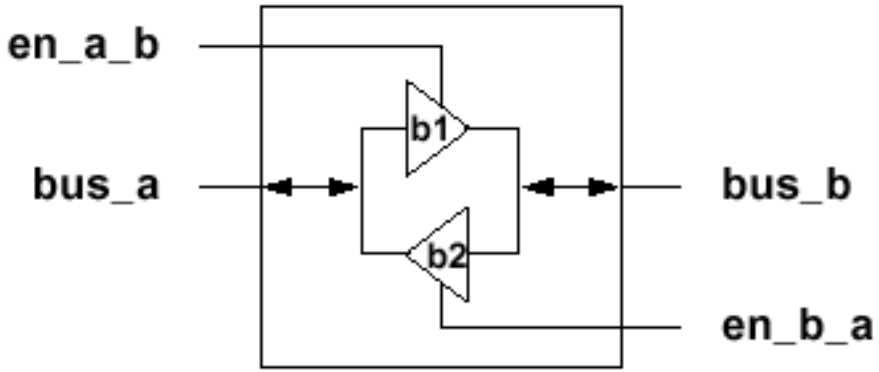
inout [7:0] databus;

双向端口声明遵循下列规则：

- **inout端口不能声明为寄存器类型，只能是net类型。**
  - ✓ 这样仿真器若有多个驱动时可以确定结果值。
  - ✓ 对**inout**端口可以从任意一个方向驱动数据。端口数据类型缺省为**net**类型。不能对**net**进行过程赋值，只能在过程块外部连续赋值，或将它连接到基本单元。
- 在同一时间应只从一个方向驱动**inout**端口。
  - ✓ 例如：在**RAM**模型中，如果使用双向数据总线读取**RAM**数据，同时在数据总线上驱动写数据，则会产生逻辑冲突，使数据总线变为未知。
  - ✓ 必须设计与**inout**端口相关的逻辑以确保正确操作。当把该端口作为输入使用时，必须禁止输出逻辑。

双向端口建模 — 使用基本单元建模

信号en_a_b和en_b_a控制使能



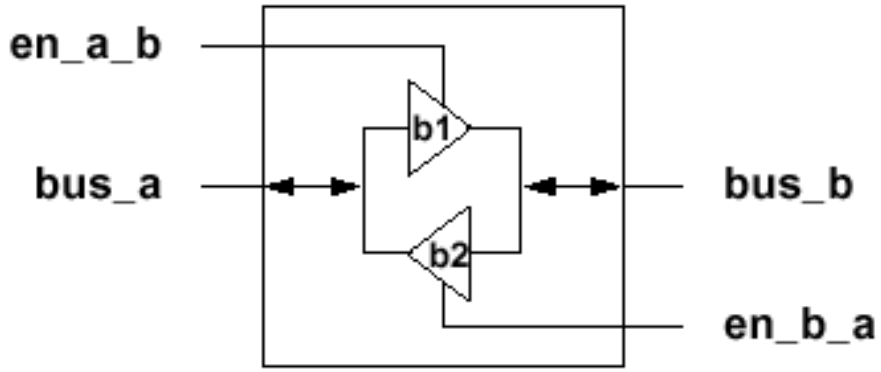

```
module bus_xcvr( bus_a, bus_b, en_a_b, en_b_a);
   inout bus_a, bus_b;
   input en_a_b, en_b_a;
   bufif1 b1 (bus_b, bus_a, en_a_b);
   bufif1 b2 (bus_a, bus_b, en_b_a);
// Structural module logic
endmodule
```

若en_a_b=1，基本单元b1使能，bus_a数据传送到bus_b

若en_b_a=1，基本单元b2使能，bus_b数据传送到bus_a

双向端口建模 — 使用连续赋值建模

信号en_a_b和en_b_a控制使能

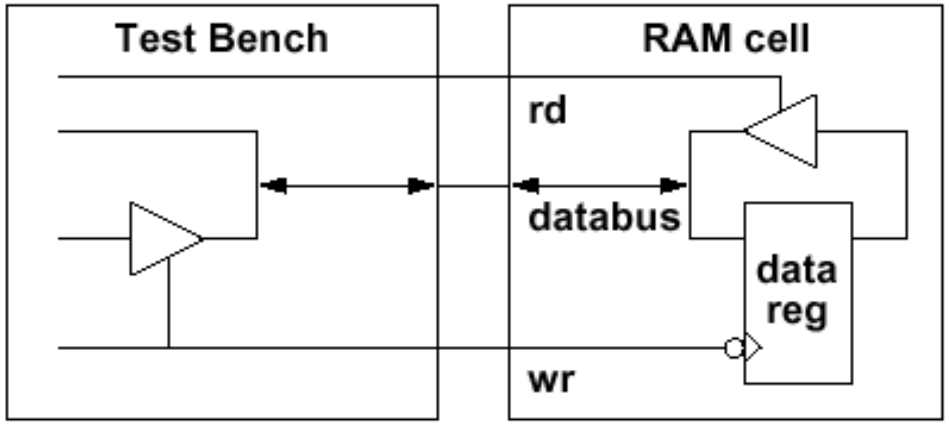



```
module bus_xcvr( bus_a, bus_b, en_a_b,
en_b_a);
    inout bus_a, bus_b;
    input en_a_b, en_b_a;
    assign bus_b = en_a_b ? bus_a : 'bz;
    assign bus_a = en_b_a ? bus_b : 'bz;
endmodule
```

若en_a_b=1，赋值语句驱动bus_a数据到bus_b

若en_b_a=1，赋值语句驱动bus_b值到bus_a

双向端口建模 — 存储器端口建模




```
module ram_cell( databus, rd, wr);
    inout databus;
    input rd, wr;
    reg datareg;
    assign databus = rd ? datareg : 'bz;
    always @( negedge wr)
        datareg <= databus;
endmodule
```

当rd=1时，datareg的值赋值databus

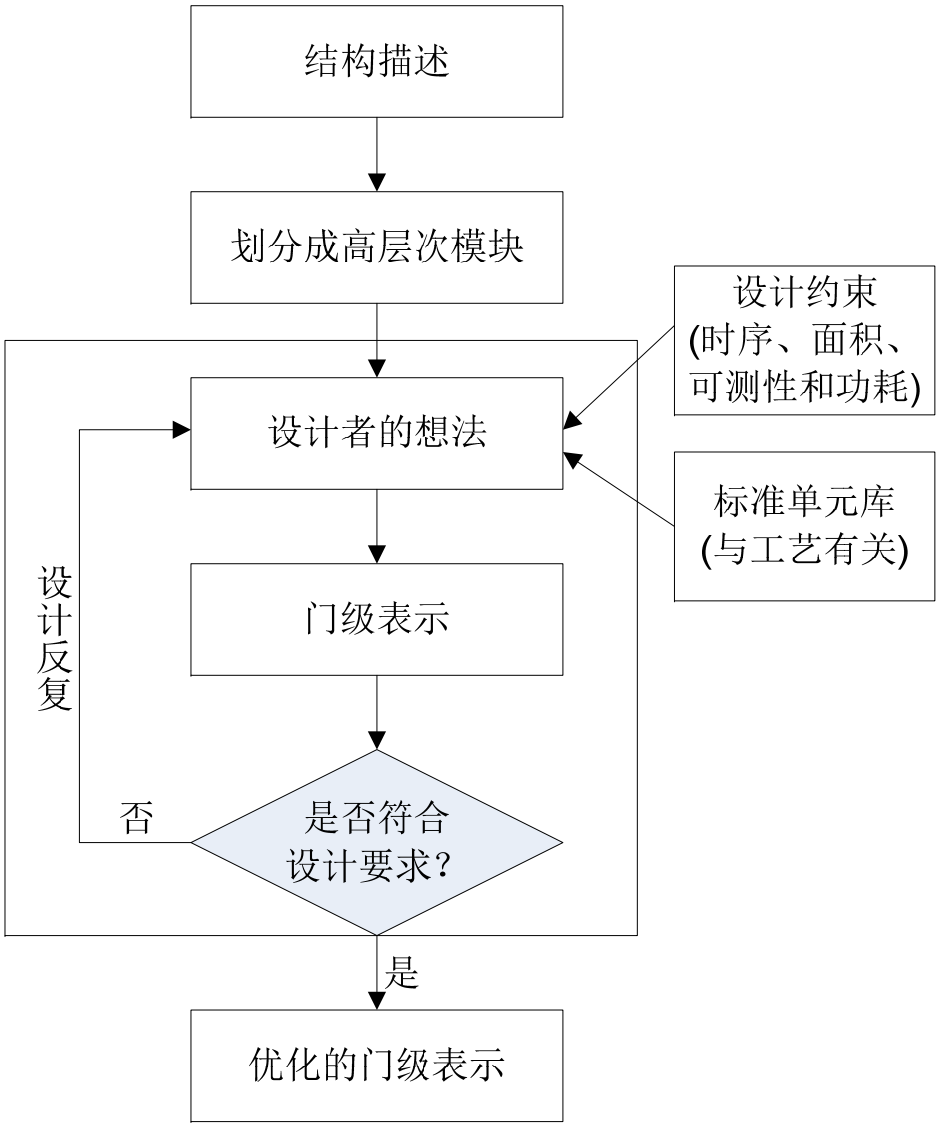
在wr下降沿，databus数据写入datareg

Verilog逻辑综合

- 什么是逻辑综合？
  - 逻辑综合是在标准单元库和特定的设计约束的基础上，把设计的高层次描述转换成优化的门级网表的过程。
  - 标准单元库可以包含简单的单元，如：与非门、或非门、与门和或门等基本逻辑门，也可以包含宏单元，如加法器、多路选择器和触发器等。

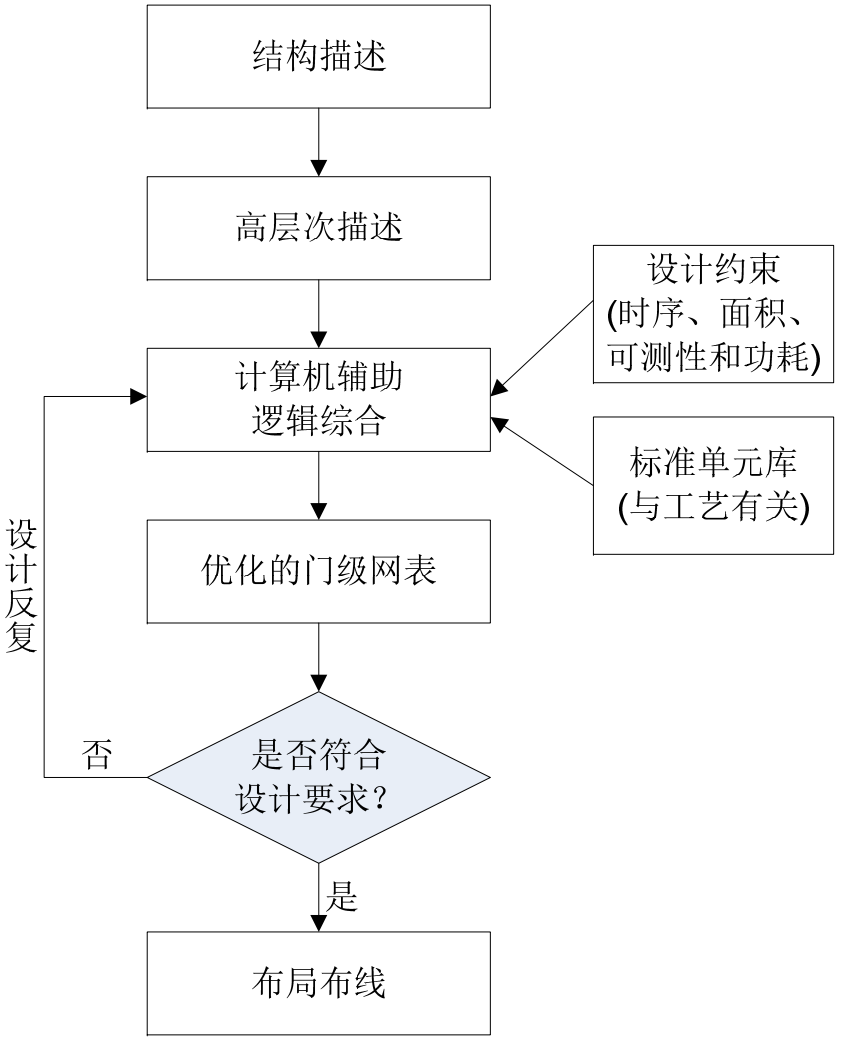
什么是逻辑综合？

- 人工完成的逻辑综合过程。
  - 绘制原理图的过程实际就是在大脑中完成逻辑综合的过程。
  - 设计过程要经过多次反复，耗费大量时间。



什么是逻辑综合？

- 计算机辅助逻辑综合工具可以把高层次描述向逻辑门的转换过程自动化。大大减少了转换时间。
- 设计者不需要在大脑里实现逻辑综合，而可以集中精力于体系结构的方案、设计的高层次描述、精确的设计约束和标准单元库中的单元优化。这些都作为综合工具的输入。
- 设计者使用HDL描述高层次设计、而不必在屏幕上画高层次描述。



逻辑综合对数字IC设计业的影响

- 手工设计的限制：
  - 容易带来错误。一个很小的逻辑门遗漏都有可能造成整个模块的重新设计。
  - 在完成门级设计并进行测试之前，设计者一直都不能确信设计约束是否能得到满足。
  - 把高层次设计转换成逻辑门占去了整个设计周期的大部分时间。
  - 如果门级设计不满足要求，模块的重新设计时间非常长。
  - 推测难以验证。例如：设计者设计了一个时钟周期为20ns的门级模块。如果设计者想分析该电路是否能优化到15ns的时钟，整个模块要重新设计。

逻辑综合对数字IC设计业的影响

- 每个设计者以不同的方式实现模块设计，设计风格缺乏一致性，对大规模设计来说，其中的各个小模块可能是最优的，但整个设计却不是最优化的。
- 如果最终的门级设计中发现了一个错误，可能需要重新设计数以千计的逻辑门。
- 标准单元库的时序、门级、功耗都是工艺相关的，所以门级设计完成后，如果改变IC制造商，就意味着要重新设计整个电路，还可能要改变设计方法。
- 设计不可能重用，因为设计与工艺有关，难以改变，也难以重用。

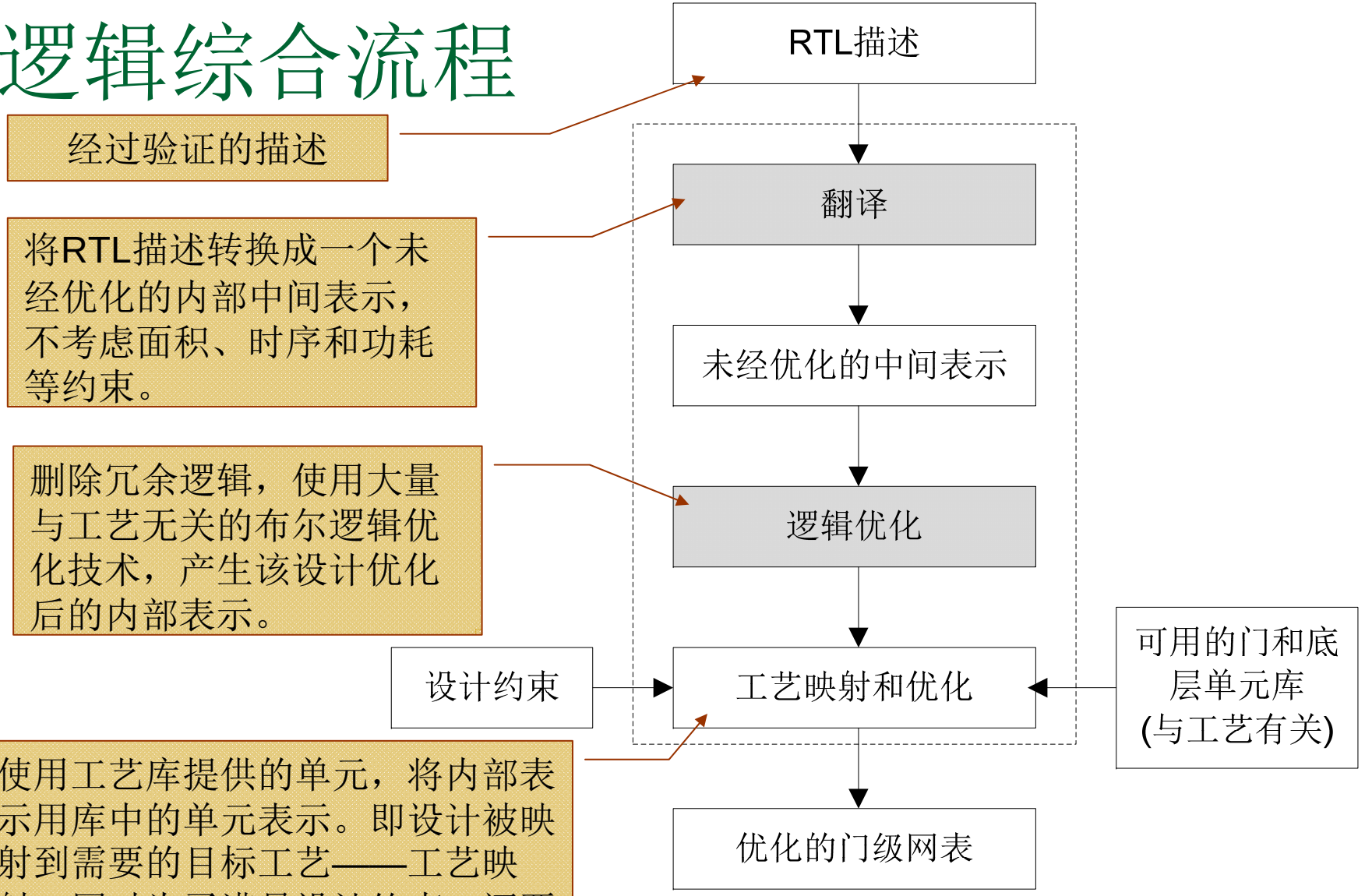
逻辑综合对数字IC设计业的影响

- 自动逻辑综合工具解决的问题：
  - 采用高层次设计方法，人为错误会更少，因为设计是在更高的抽象层次描述的。
  - 高层次设计无需过多关注设计约束。逻辑综合工具将把高层次设计转换成门级网表，并确保满足所有的约束。如果不能满足，设计者就回去修改高层次设计，重复这个过程，直到得到满足时序、面积和功耗约束的门级网表为止。
  - 从高层次设计到逻辑门的转换非常迅速。有了这方面的提高，设计周期可以大大缩短。原来耗费数月的设计现在可能只要数小时或数天即可完成。
  - 模块重新设计所需的反复时间更短，因为改变仅需要在RTL完成；然后设计只需简单地重新综合就可以获得门级网表。

逻辑综合对数字IC设计业的影响

- 推测容易验证。高层次描述不需要改变，设计者只需要把时序约束从20ns改变到15ns，并重新综合，看看是否能获得时钟周期优化为15ns的新门级网表。
- 逻辑综合工具在整体上优化了设计，这样就消除了由于不同模块之间和局部优化的各个设计之间的设计风格不同所带来的问题。
- 如果发现门级设计中有错误，设计者可以回头修改高层次描述消除错误，然后再次综合就可以生成新的门级描述。
- 逻辑综合工具允许进行与工艺无关的设计。可以在不考虑IC制造工艺的情况下编写高层次描述。改变制造工艺，设计者只需在新工艺的标准单元库基础上进行逻辑综合，就可以得到新的门级网表。
- 由于设计描述与工艺无关，设计重用成为可能。

逻辑综合流程

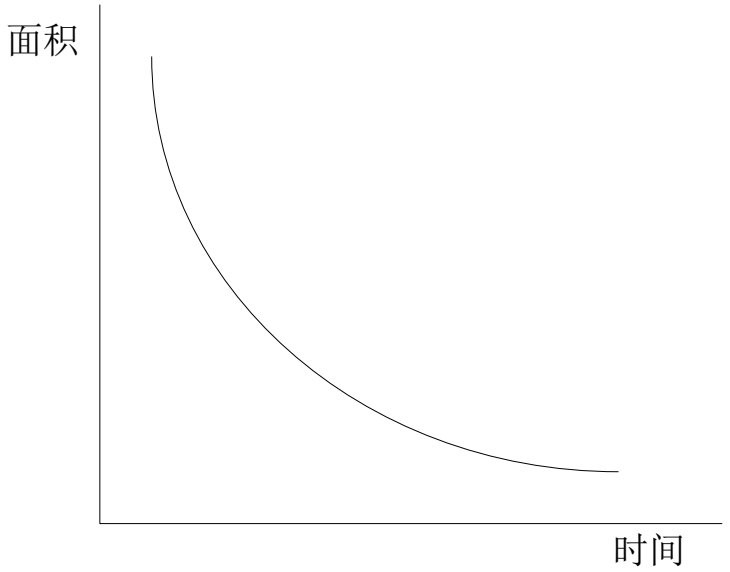

RTL描述
经过验证的描述
翻译
将RTL描述转换成一个未经优化的内部中间表示，不考虑面积、时序和功耗等约束。
未经优化的中间表示
删除冗余逻辑，使用大量与工艺无关的布尔逻辑优化技术，产生该设计优化后的内部表示。
逻辑优化
设计约束
工艺映射和优化
可用的门和底层单元库(与工艺有关)
使用工艺库提供的单元，将内部表示用库中的单元表示。即设计被映射到需要的目标工艺——工艺映射，同时为了满足设计约束，还要进行工艺相关的优化。
优化的门级网表

逻辑综合流程(续)

- 工艺库，或者也叫标准单元库
- 工艺库包括：基本逻辑门，宏单元，加法器、ALU、多路选择器、触发器等；
- 每个单元信息包括：
  - 单元功能；
  - 单元版图面积；
  - 单元时序信息；
  - 单元功耗信息；
- 综合工具产生的结果的质量通常由工艺库中可以使用的单元决定，如果工艺库中可选择的单元有限，综合工具就不能在时序、面积和功耗等方面做充分优化。

逻辑综合流程(续)

- 设计约束：
  - 时序：电路必须满足一定的时序要求，一个内部的静态时序分析器会检查时序。
  - 面积：最终的版图面积不能超过一定的限制。
  - 功耗：电路功耗不能超过一定的限制。
- 一般面积和时序约束之间是矛盾的。对于特定的工艺库，为了优化时序(获得更高的速度)，设计不得不做并行处理，这往往意味着生成更大的电路。



可综合风格的Verilog描述

如果逻辑输出在任何时候都直接由当前输入组合决定，则为组合逻辑。
如果逻辑暗示存储则为时序逻辑。如果输出在任何给定时刻不完全由输入的状态决定，则暗示存储。

通常综合输出不会只是一个纯组合或纯时序逻辑。

一定要清楚所写的源代码会产生什么类型输出，并能够反过来确定为什么所用的综合工具产生这个输出，这是非常重要的。

- 组合逻辑使用阻塞赋值语句描述；
- 时序逻辑使用非阻塞赋值语句描述。

如何为综合工具书写Verilog代码

- 可综合的**HDL**的语法只是它们自己语言的一个子集；
- 可综合子集的国际标准还没有形成，各综合器所支持的可综合**HDL**子集也略有所不同；
- 目的
  - 减少综合的次数或运行时间，最快得到良好的综合结果
  - 采用比较简单的综合约束就可以得到时序和面积的要求
  - 提高综合结果网表的性能，简化静态时序分析的过程

可综合的VerilogHDL

- 逻辑综合工具并不能处理随意编写的Verilog描述。
- 可进行逻辑综合的VerilogHDL结构
  - 端口：input, inout, output
  - 参数：parameter
  - 模块定义：module
  - 信号变量：wire, reg, tri；允许使用向量表示
  - 调用(实例引用)：module instance, gate instance
  - 函数和任务：function, task；不考虑时序结构
  - 过程：always, if, else, case, casex, casez；
  - 过程块：begin...end，命名块，disable
  - 数据流：assign；不考虑延时
  - 循环：for, while, forever；while和forever必须包含@(posedge clock)或@(negedge clock)

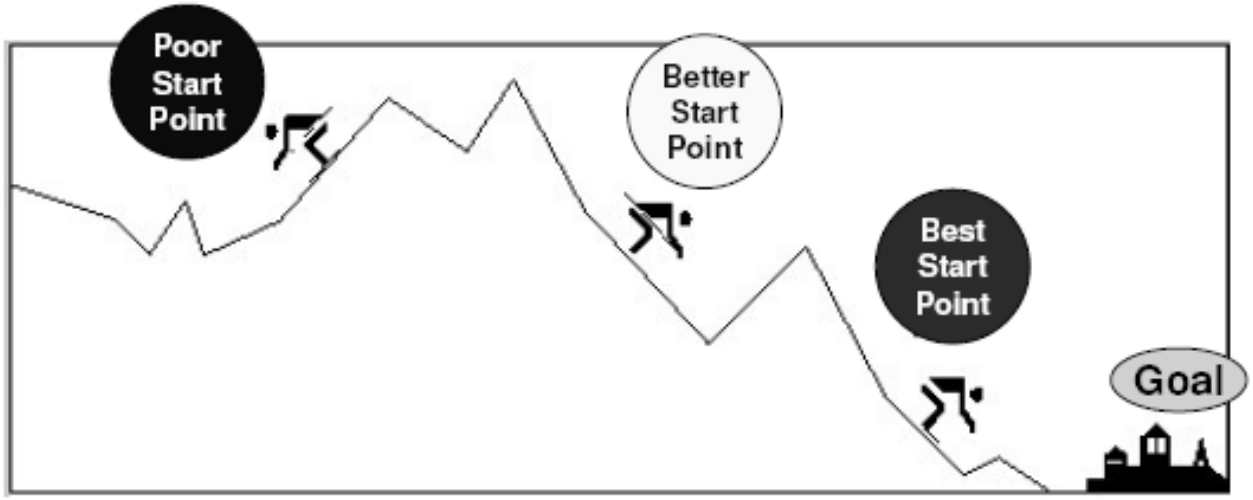
可综合的VerilogHDL

- 可综合的操作符
  - 几乎所有的操作符都可以被综合，除了===，!==两个操作符。
  - 算术操作符： *, /, +, -, %, +(单目), -(单目)
  - 逻辑操作符：!, &&, ||
  - 关系操作符：>, <, >=, <=
  - 等于：==, !=
  - 按位逻辑：~, &, |, ^, ^~或~^
  - 归约：&, ~&, |, ~|, ^, ^~或~^
  - 移位：>>, <<, >>>, <<<
  - 连接：{ }
  - 条件：？:

不可综合的VerilogHDL

- 不可综合的Verilog关键字
  - time、defparam(不一定支持)、fork、join、initial、wait
  - 各种延时
  - UDP
- 不支持的线网类型：triand, trior, trireg, tri0, tri1
- 不支持的类型说明：
  - time，real
  - 所有的MOS开关：nmos, pmos, rnmos, rpmos, cmos, rcmos
  - 双向开关：tranif1, tranif0, rtran, rtranif1, rtranif0
  - pullup、pulldown

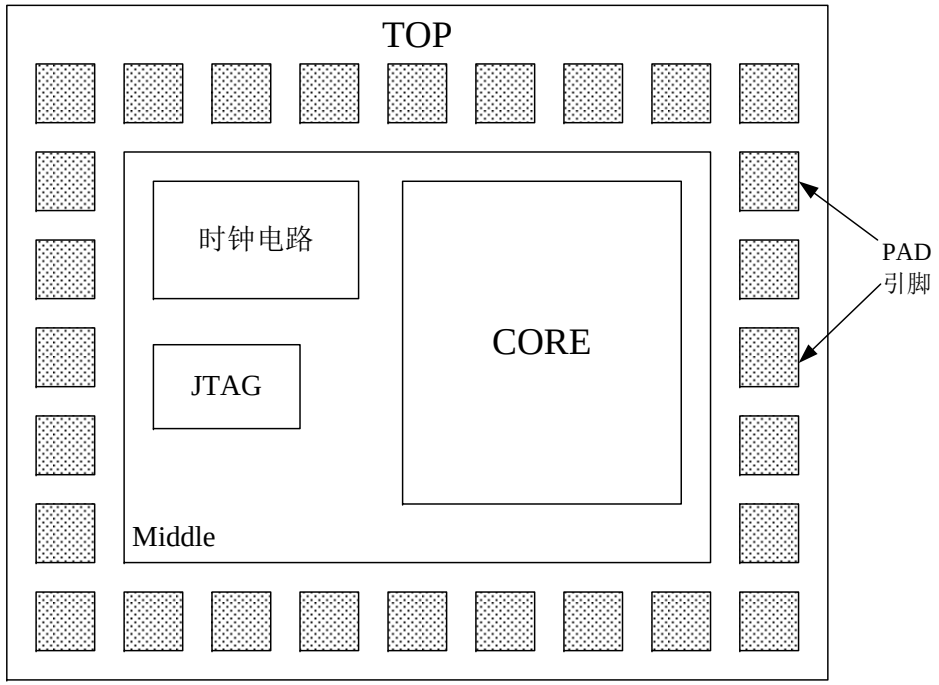
The Importance of quarlity Source



- Codes that are functionally equivalent, but coded differently, will give different synthesis results
- You cannot rely solely on Design Compiler to "fix" a poorly coded design!
- Try to understand the "hardware" you are describing, to give DC the best possible starting point

针对综合的模块划分规则(1)

- 良好的模块划分
  - 只有顶层模块包含I/O；
  - 核心单元门口、边界扫描单元、时钟电路等。



针对综合的模块划分规则(2)

- 核心(CORE)逻辑进行模块划分，要避免子模块之间出现连接用的胶合逻辑
- 减少顶层模块综合时CPU运行时间
- 简化顶层模块综合批处理文件

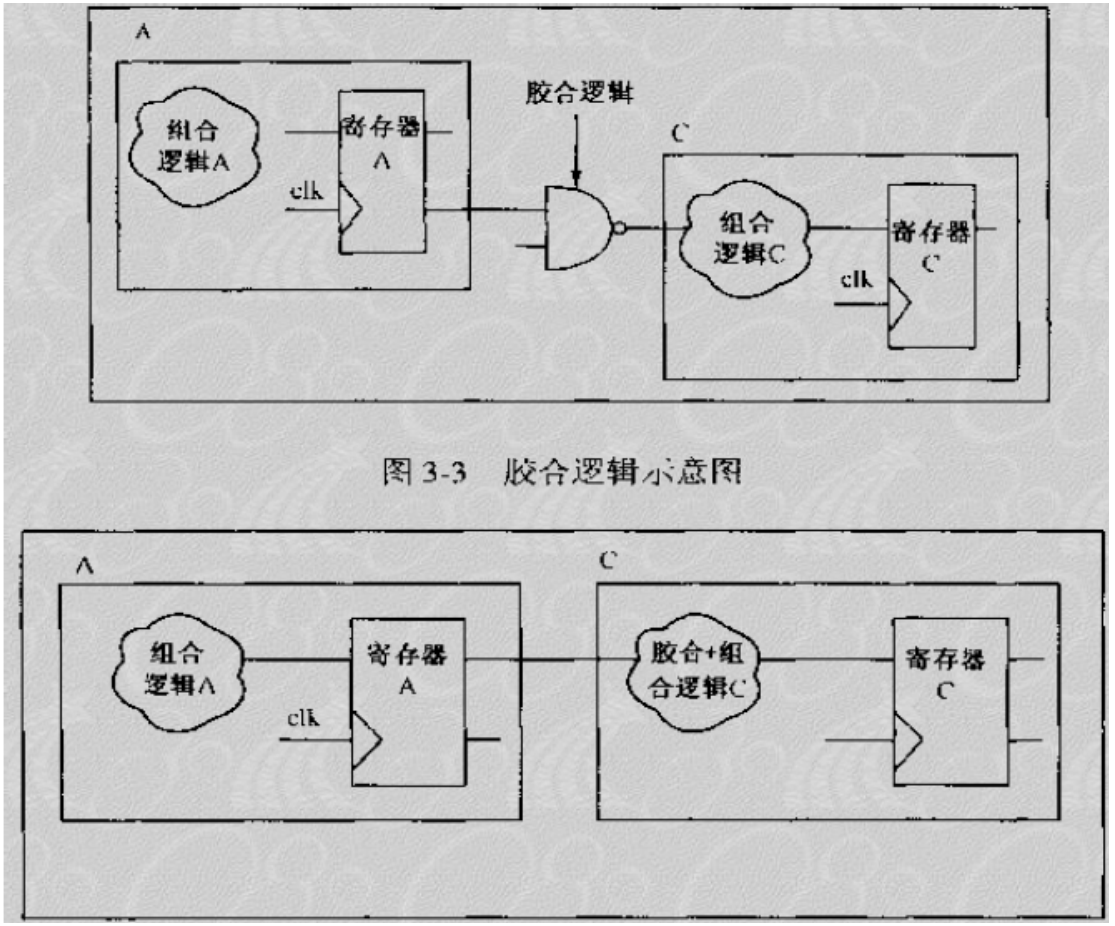
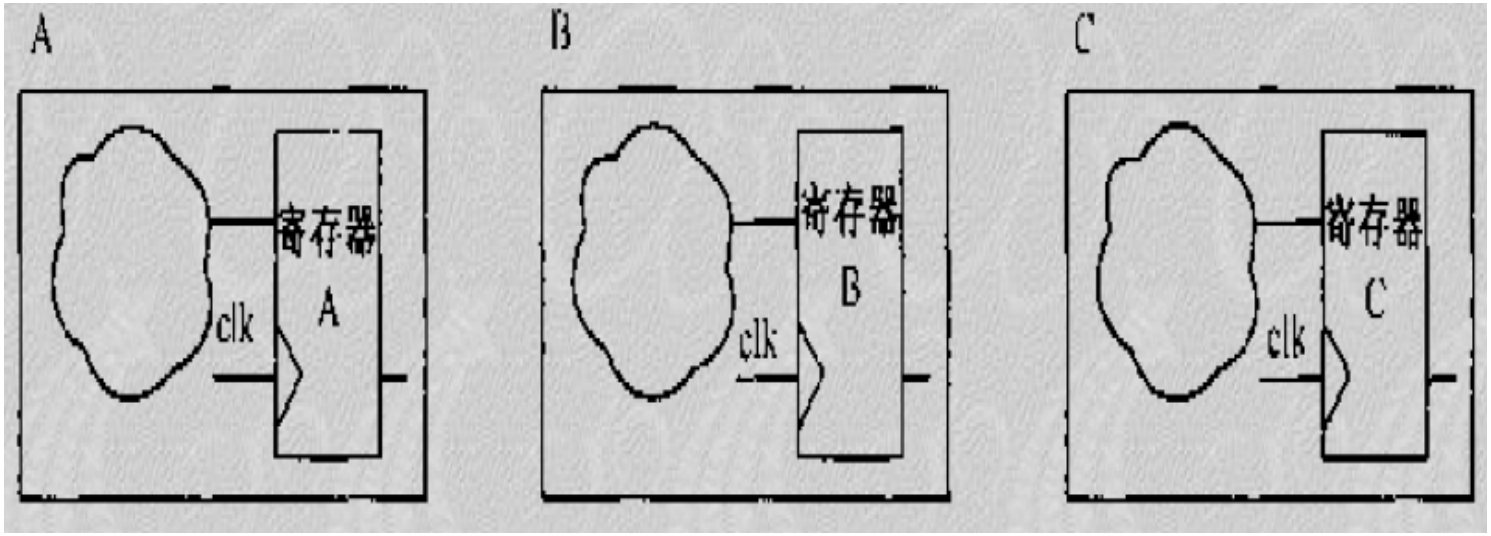


图 3-3　胶合逻辑示意图

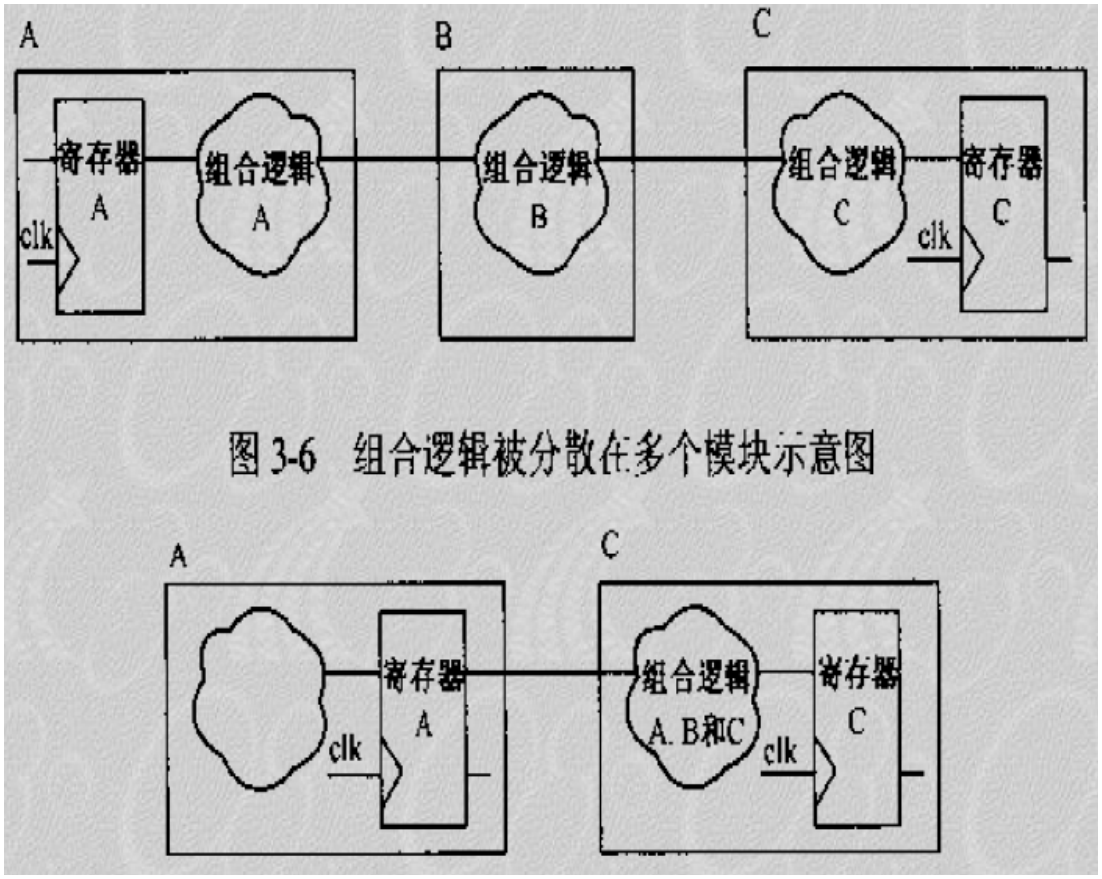
针对综合的模块划分规则(3)

- 每一模块采用寄存器锁存输出：
  - 保证子模块的所有输入信号延时基本一致，只与模块间的布线延时有关；
  - 避免异步路径，容易加综合约束条件
  - 输出信号的驱动强度等于寄存器的平均驱动能力



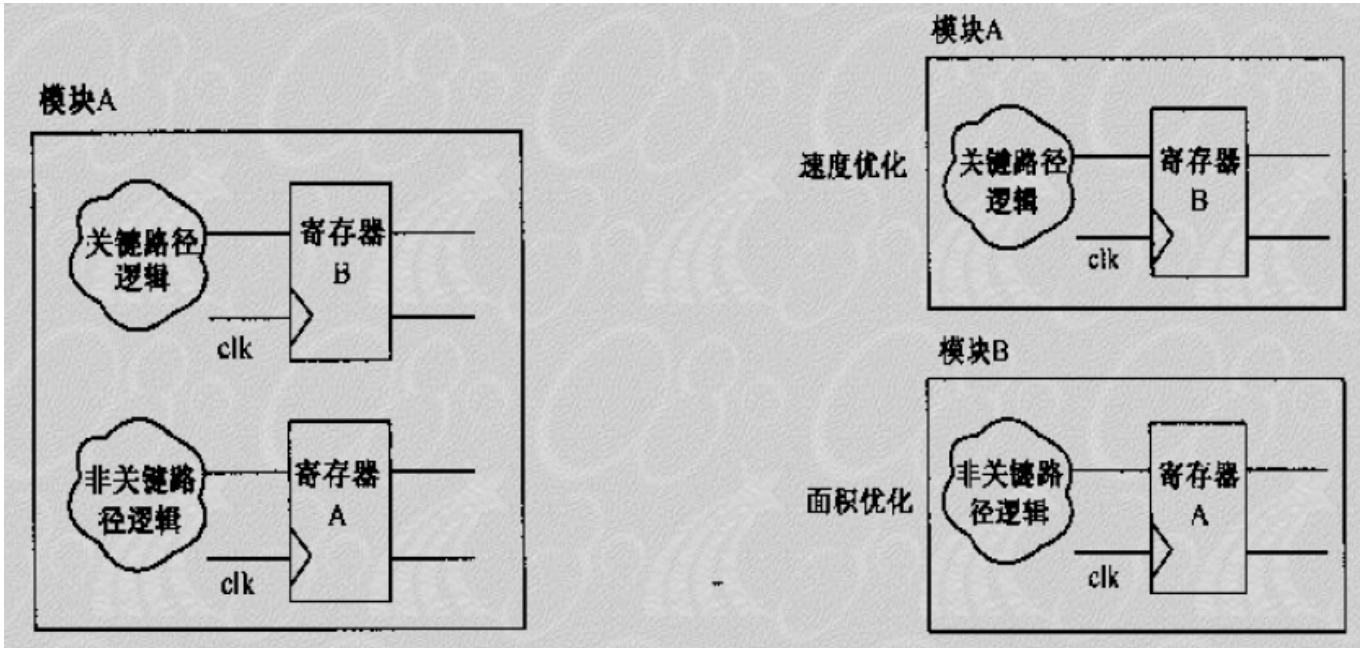
针对综合的模块划分规则(4)

- **DC**不能跨越模块边界对相关的组合逻辑做归并优化
- 尽可能把相关的组合逻辑集中到一个模块做归并优化
- 有利于**DC**完成时序优化
- 时间预算变得容易
- 缩短仿真时间



针对综合的模块划分规则(5)

- **DC**不能对一个模块内不同组的逻辑采用不同的约束条件，**ungroup**又会破坏原有设计层次
- 建议不同设计目标的电路分成不同的模块：向关键路径模块要时间，向非关键路径模块要面积



关键路径逻辑和非关键路径逻辑混淆　　关键路径和非关键路径的逻辑被分开

采用与工艺无关的寄存器实现时序逻辑

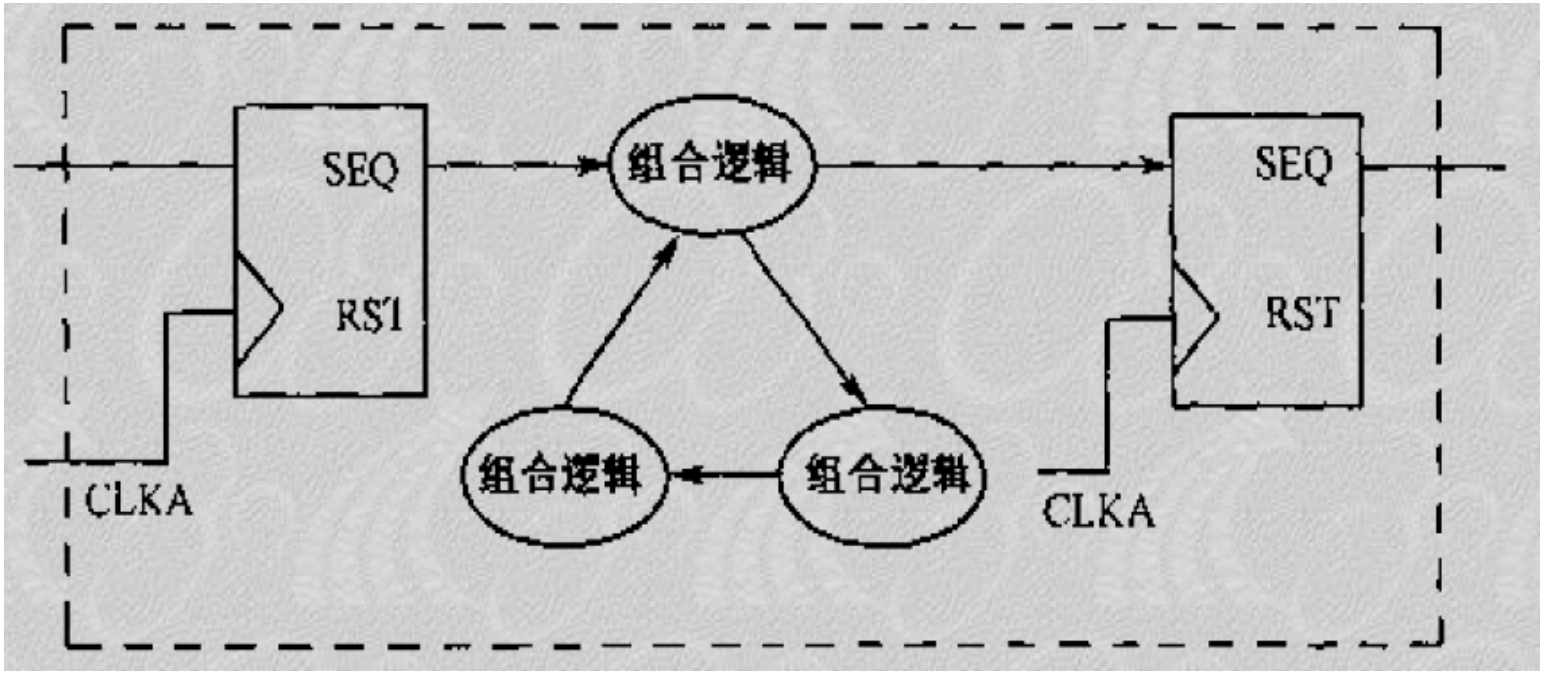
- 为了防止综合前后仿真结果的不一致，建议采用独立于工艺的寄存器实现时序逻辑，并用复位信号来初始化每一个寄存器。
- 综合工具不支持**initial**语句，在**verilog**电路描述中不要使用**initial**语句对信号赋值。

always语句

- 在生成组合逻辑的**always**块中，赋值表达式右端参与赋值的信号必须在敏感列表中列出。
  - 仿真工具的仿真结果与敏感列表是否完全关系密切
  - 敏感表不完备导致综合前后仿真结果不一致
- 如果敏感列表中的信号在**always**模块中没有出现，不会出现错误，只会使综合前仿真速度变慢；
- 带有**posedge**或**negedge**关键字的事件表达式表示沿触发时序逻辑，没有**posedge**或**negedge**表示组合逻辑或电平敏感的锁存器；
- 对于时序模块，敏感列表必须包含触发时钟信号，而且只能由一个时钟跳变沿触发，置位和复位最好也由该时钟跳变沿触发(同步复位/置位)。异步置位/复位要将复位/置位信号放在敏感量表中(边沿触发)。

避免组合逻辑反馈

- 在设计中要避免组合反馈环，组合反馈环的电路不利于**DFT**测试和静态时序分析；



避免latch——不完全条件语句

若 a 变为 0，
e 为何值

```
module inccase (a, b, c, d, e);
   input a, b, c, d;
   output e;
   reg e;
   always @( a or b or c or d)
      case ({ a, b})
         2'b11: e = d;
         2'b10: e = ~c;
      endcase
endmodule
```

```
module incpif (a, b, c, d, e);
   input a, b, c, d;
   output e;
   reg e;
   always @( a or b or c or d)
      if (a & b)
         e = d;
      else if (a & ~b)
         e = ~c;
endmodule
```

在上面的例子中，当a变为零时，不对e赋新值。因此e保存其值直到a变为1。这是锁存器的特性。

default完全条件语句

```
module comcase (a, b, c, d, e);
   input a, b, c, d;
   output e;
   reg e;
   always @( a or b or c or d)
      case ({ a, b})
         2'b11:   e = d;
         2'b10:   e = ~c;
         default: e = 'bx;
      endcase
endmodule
```

```
module compif (a, b, c, d, e);
   input a, b, c, d;
   output e;
   reg e;
   always @( a or b or c or d)
      if (a & b)
         e = d;
      else if (a & ~b)
         e = ~c;
      else
         e = 'bx;
endmodule
```

综合工具将 ‘bx作为无关值，因此if语句类似于“ full case”，可以进行更好的优化。

例中没有定义所有选项，但对没有定义的项给出了缺省行为。同样，其综合结果为纯组合逻辑——没有不期望的锁存器产生。

避免引入不必要的latch

- 避免在综合时引入锁存器的方法：
  - 组合函数的输出必须在每个可能的控制路径中被赋值
  - 每次执行**always**块时，在生成组合逻辑的**always**块中赋值的所有信号都必须有明确的值
  - 组合电路的每一个**if**描述语句都对应一个**else**语句
  - 每一个**case**语句都对应一个**default**语句（在没有优先级的情况下优先使用，设计路径延时要小于**if**－**else**）

case语句

- **//synopsys full case**
  - case语句通过对每个可能的控制路径的输出进行说明来帮助避免产生推断的锁存器。
  - 一些综合工具提供编译指令，允许在所有case项没有都说明的情况下也认为case是完备的。此时，不会推断出锁存器，并且出于逻辑优化的考虑，未进行说明的case项的输出值被认为是未定义的。
  - 综合器对于未声明的分支，其输出为“don’t care”。
  - 可能导致仿真与综合的不匹配
  - 有时候使综合出的电路更大、更慢

casex

- 使用casex可能引起问题
  - casex语句将‘x’和‘z’处理为“don’t cares”，不必考虑。无论它是出现在case表达式和分支项中。
  - 当casex表达式初始化到一个未知状态时，则发生错误。(综合前仿真将未知输入当作don’t care，综合后仿真将传播‘x’)
- 尽量不用，或用casez代替

casex

```
module synUsingDC(f, a, b);
  output f; input a, b;
  reg f;
  always@(a or b)
    casex({a, b})
      2'b0?: f=1;
      2'b10: f=0;
      2'b11: f=1;
    endcase
endmodule
```

第一个case项说明如果a为0，那么f为1。语句中？的使用表示在这种情况下b的值无关紧要，也可以用x或者z。

casex

```
module code(memce0, memce1, cs, en, addr);
  output memce0, memce1, cs; input en;
  input [31:30] addr;
  reg memce0, memce1, cs;
  always@(addr or en) begin
    {memce0, memce1, cs}=3'b0;
    casex ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs = 1'b1;
    endcase
  end
endmodule
```

casez

- 与casex语句类似，但casez仅匹配高阻信号，因此引起错误的概率较低。
- 在地址译码或priority decoder建模时有用
- 需要谨慎使用
- 为了减少仿真和综合之间的差别，可综合的描述不与x或z作比较。此外， x仅赋值给组合电路的输出

casez

```
module code(memce0, memce1, cs, en, addr);
  output memce0, memce1, cs; input en;
  input [31:30] addr;
  reg memce0, memce1, cs;
  always@(addr, or en) begin
    {memce0, memce1, en}=3'b0;
    casez ({addr, en})
      3'b101: memce0 = 1'b1;
      3'b111: memce1 = 1'b1;
      3'b0?1: cs = 1'b1;
    endcase
  end
endmodule
```

casez

```
module mux_z(out, a, b, c, d, select);
  output out; input a,b,c,d;
  input [3:0] select;
  reg out;
  always@(select or a or b or c or d) begin
    casez(select)
      4'b???1: out = a;
      4'b??1?: out = b;
      4'b?1??: out = c;
      4'b1???: out = d;    // 后面不再需要default语句了
    endcase
  end
endmodule
```

x使用的限制

- 对于可能用到的运算符和未知量（x）的方式有一定限制。在可综合的描述中可以使用未知量，但仅在某些情况下可以用。
- assign y=(a==1'bx)?c:1;
  - 上面语句不可综合，它用来确定a是否变成未知量了，而实际未知量硬件上不存在的。因此不可综合。
- assign y=(a==b)?1'bx:c;
  - 这是可综合的，当a等于b时，并不关心给y赋什么值。如果不相等，y被赋值成c。这个例子中，a等于b时，y可以是1，又可以是0，所以最佳的实现就是y=c。

初始化

- 赋值以‘x’可能引起仿真和综合的不一致。
- 在仿真中，x是四个已定义的逻辑值中的一个，与x进行比较在仿真器中是有意义的，而在综合中无意义。
- 仿真器将‘x’解释为unknown.
- 把某一信号赋值‘x’，综合器就把它解释为don’t care，无关状态，把它作为一个逻辑无关项说明，可以是任意值。因此,综合器为其生成的硬件电路最简单

带有无关项模块

```
module syncase(f, a, b, c);
  output f; input a, b, c; reg f;
  always@(a or b or c)
    case ({a, b, c})
      3'b001: f = 1'b1;
      3'b010: f = 1'b1;
      3'b011: f = 1'b1;
      3'b100: f = 1'b1;
      3'b110: f = 1'b0;
      3'b111: f = 1'b1;
      default: f = 1'bx;
    endcase
endmodule
```

无关项

```
module cade1(y, a, b, c, s);
  output y; input a,b,c;
  input [1:0] s; reg y;
  always@(a or b or c or s)
  begin
    y=1'bx;
    case(s)
      2'b00: y=a;
      2'b01: y=b;
      2'b10: y=c;
    endcase
  end
endmodule
```

```
module cade2(y, a, b, c, s);
  output y; input a,b,c;
  input [1:0] s; reg y;
  always@(a or b or c or s)
    case(s)
      2'b00: y=a;
      2'b01: y=b;
      2'b10, 2'b11: y=c;
    endcase
  end
endmodule
```

速度更快，面积更小

时序电路综合

- always块的敏感表仅包括时钟沿、复位或置位条件。
- always块内，首先规定复位和置位条件。如**if**（**~reset**）, **if**（**reset**）
- **begin…end**块内没有时钟的条件。最后面的**else**中的赋值语句是由综合工具来假定下一个状态。
- 在顺序**always**块中进行赋值的任何寄存器将采用在综合结果电路中的触发器来实现。
- 当规定边沿敏感的电路行为时，**“<=”**表示规定在敏感表中的边沿上出现的整个系统的所有转换将并发地出现。

组合逻辑电路综合

- 检查所有组合函数的所有输入是否都包含在控制事件的敏感表内，如果其中的一个输入改变了，那么重新计算输出；
- always块中的赋值采用阻塞赋值语句；
- 在执行每个**begin ... end**循环中输出至少被赋值一次，否则，电路需要记住以前的值，综合需要锁存器来实现。

时钟和复位

- 良好的时钟设计十分关键
- 原则一:采用简单的时钟结构
- 原则二:采用单一的全局时钟信号
- 原则三:同一模块中所有的寄存器都在时钟的上升沿触发

避免同时使用上升沿和下降沿

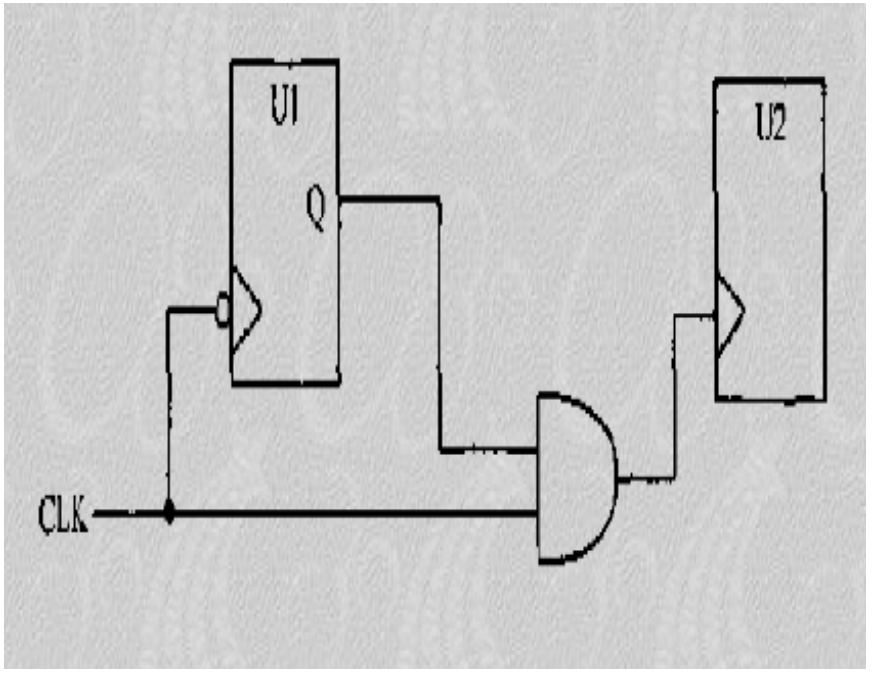
- 混合时钟沿的设计在时序要求苛刻的关键路径上有时可能是必要的，甚至要求两个时钟沿都能捕获到数据
  - 需要考虑时钟占空比
  - 多数基于扫描链的测试方法要求时钟上升沿和下降沿触发的寄存器分开处理
- 确保在综合和时序分析最坏情况下的时种占空比不会导致电路失败
- 将上升沿和下降沿触发的寄存器分别放在不同的模块中实现，有利于**DFT**工具产生扫描链

时钟优化

- 时钟**Buffer**通常是在综合完成后在物理设计时插入的
- 在布局布线阶段，时种树插入工具会尽可能采用合适的时钟树结构产生接近理想的、平衡的分布网络
- 在综合阶段，时种网络通常被认为是没有延时的理想网络。

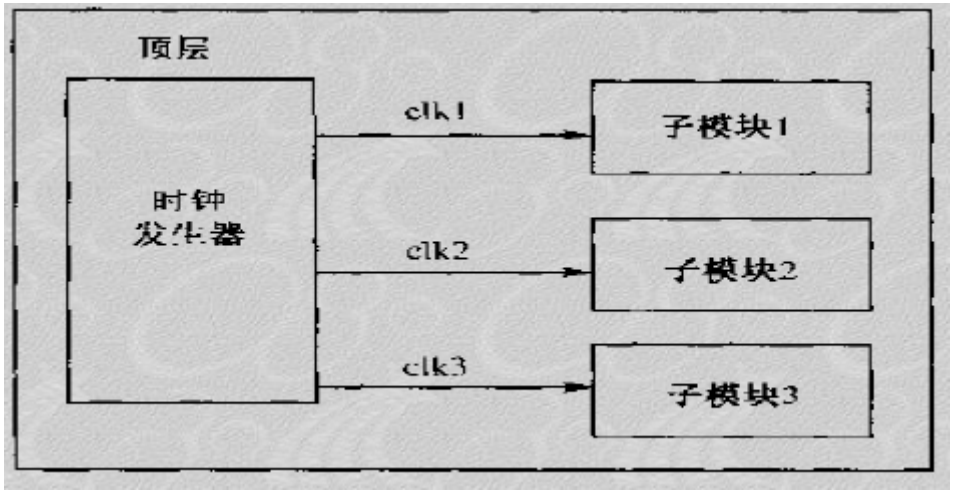
避免使用组合逻辑时钟和门控时钟

- 内部时钟信号很难用扫描链技术达到测试的目的
- 综合的约束文件变得更加复杂和困难
- 门控时钟其时序往往依赖于具体的实现工艺，时序紧张的门控时钟电路会引发电路的操作错误
- 多个内部时种信号的扭曲量不同会导致相关寄存器在某些条件下保持时间要求不满足



避免使用组合逻辑时钟和门控时钟

- 低功耗设计需要门控时种不应该在**RTL**级代码编写时插入，应该用**EDA**工具去自动完成。
- 必须采用组合逻辑时钟或门控时钟，最好在芯片的顶层实现。将时种作为一个单独的模块（仅对时钟模块开发特定的测试电路；）



避免使用组合逻辑时钟和门控时钟

- 模块级设计需要使用门控时钟的地方，一般又可以把门控信号作为赋值使能信号同步插入

```
assign clk_p1 = clk & p1_gate;
always@(posedge clk_p1) begin
….
end
```

可替换为：

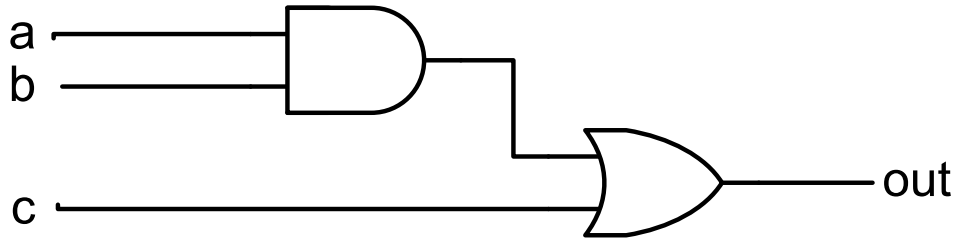
```
always@(posedge clk)
begin
  if (p1_gate = 1’b1) begin
   ……
  end
end
```

避免使用组合逻辑时钟和门控时钟

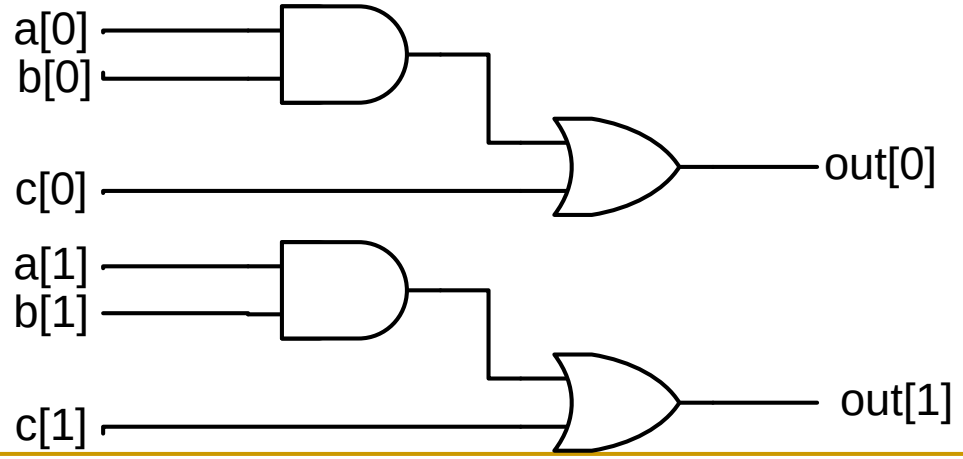
- 复位信号的设计，要采用单一的全局复位信号；
- 避免使用模块内部产生的条件复位信号，模块内部产生的条件复位信号可以转换为同步输入的使能信号处理；
- 芯片内部信号、软件写寄存器提供的全局复位信号、针对某些功能的局部模块复位信号都应该采用同步复位策略；
- 所有的时钟信号和复位信号在芯片的最顶层都必须是可控制和可观测的；

Verilog算法与硬件对应关系

- 赋值语句，用于描述组合逻辑的基本结构。
- 例如：assign out = (a & b) | c;

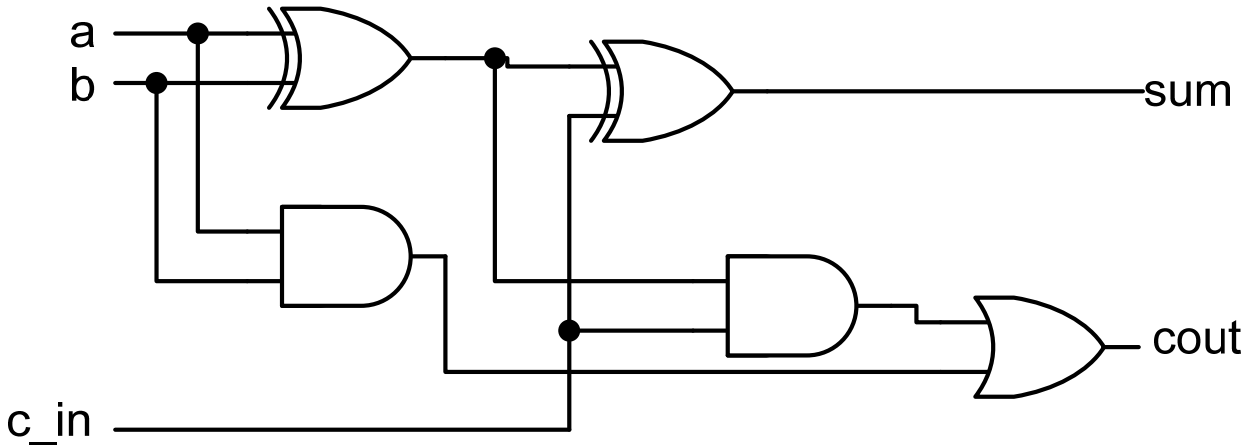


- 如果a, b, c和out都试两位的矢量[1:0]，上面的赋值会被综合成两个完全相同的电路。



Verilog算法与硬件对应关系

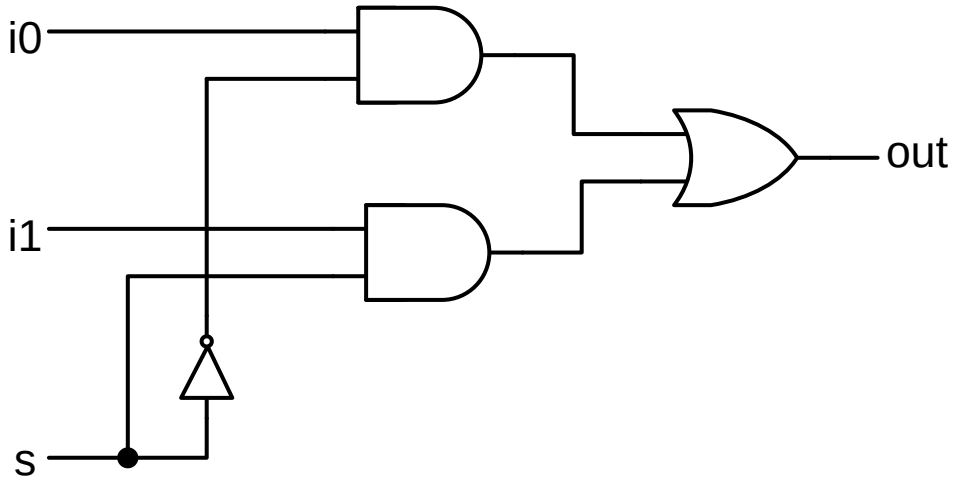
- 如果表达式中用到算术操作符，每个算术操作符由逻辑综合工具中可用的算术运算硬件模块实现。
- 如：assign {cout, sum}=a+b+cin;



- 如果是多位加法器，综合工具会进行优化，结果可能与上图不同。行波进位，进位旁路，进位选择，进位保留和超前进位等各种加法器，取决于单元库中加法器的类型。

Verilog算法与硬件对应关系

- 条件操作符会被推断成多路选择器。
- 如：assign out = s ? i1: i0;

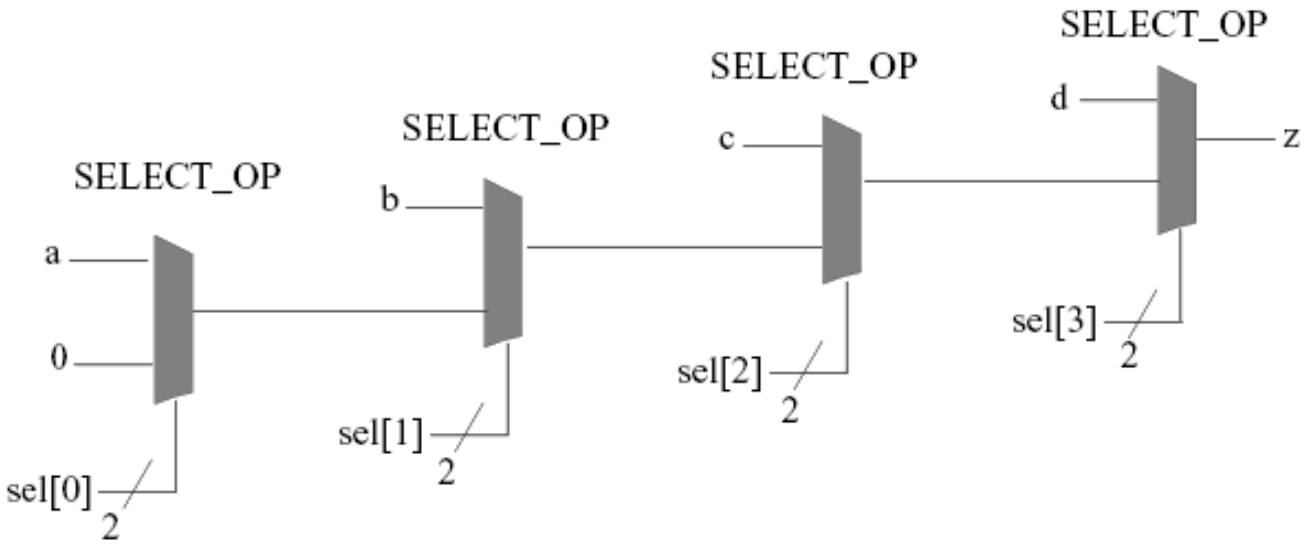


- if-else也被转换成电路选择器，但要注意如果不写else会推断出latch。

Verilog算法与硬件对应关系

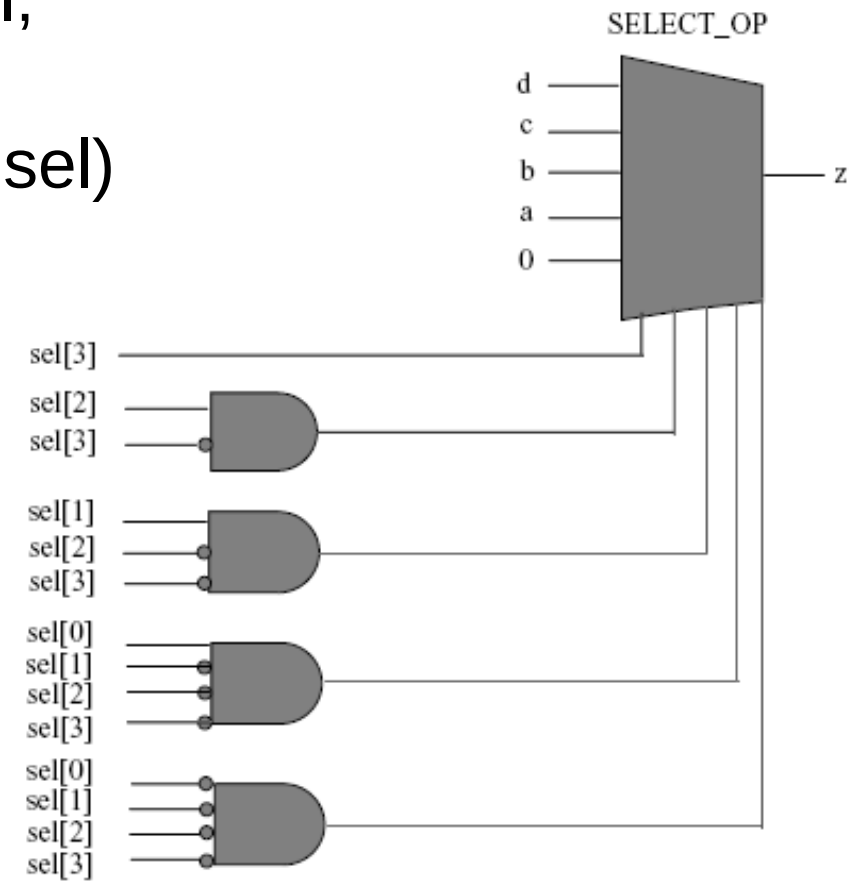
- 多级if-else-if被综合成带优先级的译码电路。

```
module mult_if(a, b, c, d, sel, z);
  input a, b, c, d; input [3:0] sel;
  output z; reg z;
  always @(a or b or c or d or sel)
  begin
    z = 0;
    if (sel[0]) z = a;
    if (sel[1]) z = b;
    if (sel[2]) z = c;
    if (sel[3]) z = d;
  end
endmodule
```



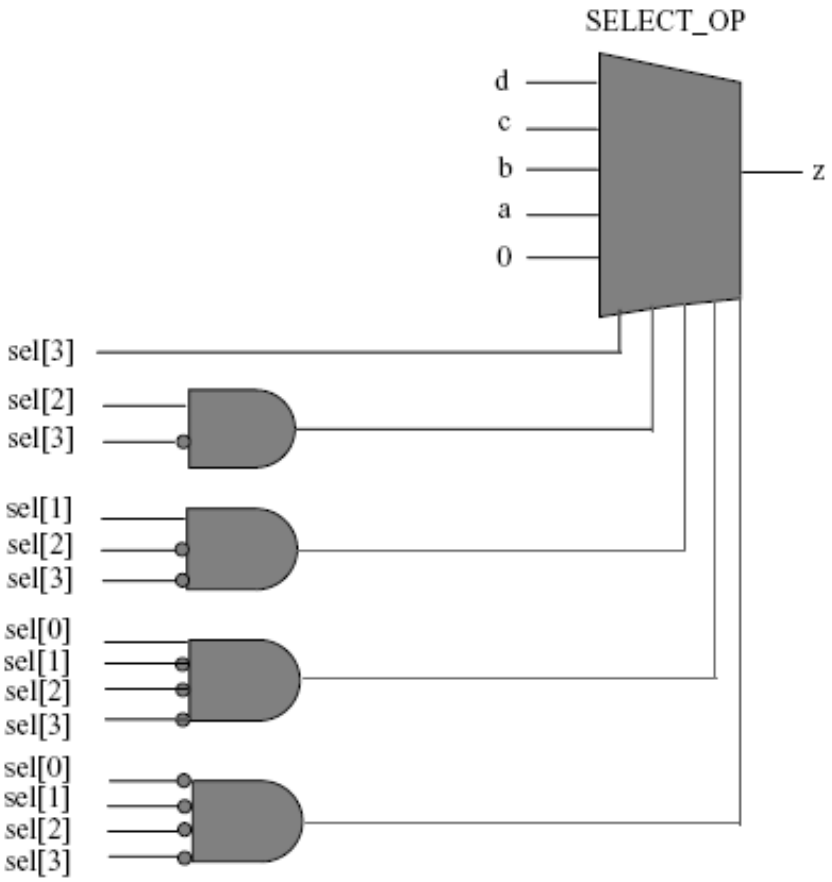
Verilog算法与硬件对应关系

```
module single_if(a, b, c, d, sel, z);
input a, b, c, d; input [3:0] sel;
output z; reg z;
always @(a or b or c or d or sel)
begin
  z = 0;
  if (sel[3])  z = d;
  else if (sel[2])  z = c;
  else if (sel[1])  z = b;
  else if (sel[0])  z = a;
end
endmodule
```



Verilog算法与硬件对应关系

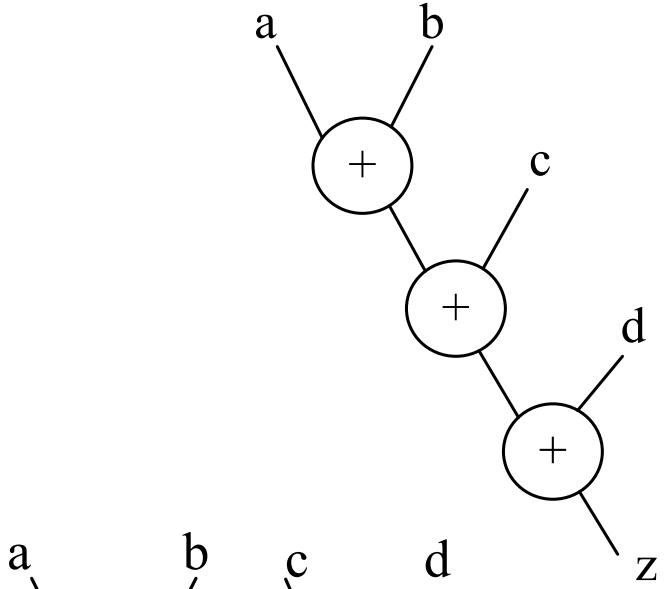
```
module case1(a, b, c, d, sel, z);
  input a, b, c, d;    input [3:0] sel;
  output z; reg z;
  always @(a or b or c or d or sel)
  begin
    casex (sel)
      4'b1xxx: z = d;
      4'bx1xx: z = c;
      4'bxx1x: z = b;
      4'bxxx1: z = a;
      default: z = 1'b0;
    endcase
  end
endmodule
```



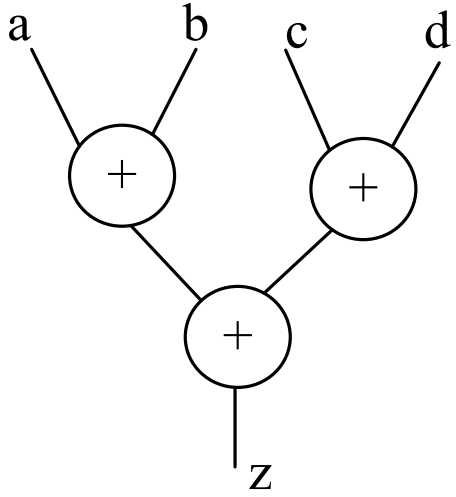
算法与硬件的关系

- 多输入加法，括号的作用

直接描述：z=a+b+c+d;



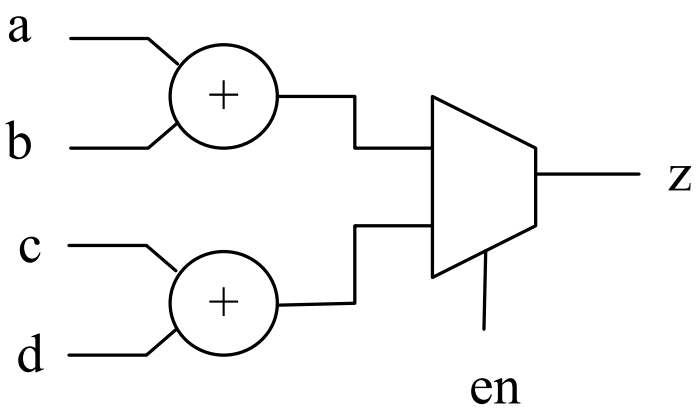
改进描述：z=(a+b)+(c+d);



算法与硬件的关系(续)

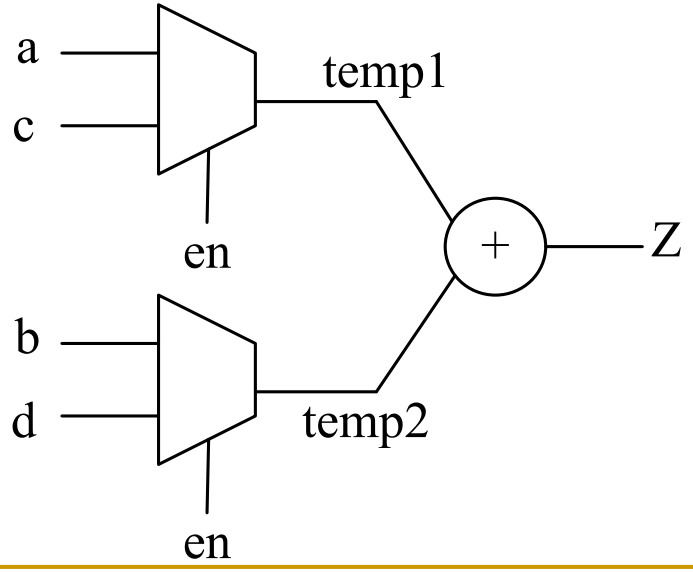
- 硬件共享

```
if(en) z=a+b;
else z=c+d;
```



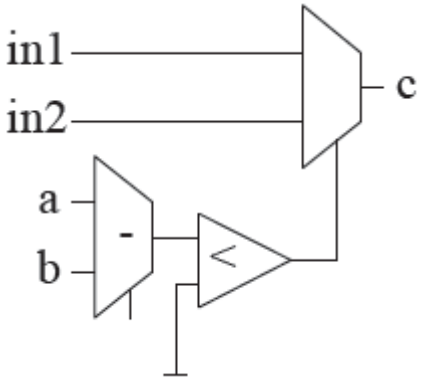
- 改进描述：

```
if(en) begin
   temp1=a;   temp2=b;
end
else begin
 temp1=c; temp2=d;
end
z=temp1+temp2;
```



算法与硬件的关系(续)

```
always@(a or b or in1 or in2)
begin
    if ((a - b) > 0)
        c = in1;
    else
        c = in2;
end
```

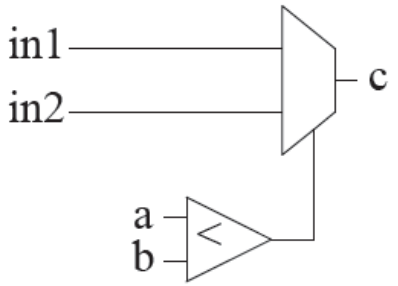


综合的结果？

需要工具根据常数优化逻辑

算法与硬件的关系(续)

```
always@(a or b or in1 or in2)
begin
    if (a > b)
        c = in1;
    else
        c = in2;
end
```



- 结果更好
- 不需要工具做出优化
- 好的代码风格能够能够加速综合器的综合和映射速度

良好的Verilog可综合代码风格

- 采用异步复位，控制电路中所有的FF和latch的异步置位或者清零；
- 只使用时钟的上升沿/或下降沿；
- Reset, set, Clock采用clean信号(或者外部输入信号)；采用gated clock时要保证信号没有glitch；
- vector range的定义采用descending order；
- 模块调用采用显式端口连接，并保持与模块定义中端口顺序一致；
- 模块端口定义按照input、output顺序；
- 芯片内部模块避免使用inout端口；
- reg变量禁止在多个always块中赋值(除了三态总线，最好三态总线也避免使用)；

良好的Verilog可综合代码风格(续)

- FSM至少采用两个进程描述，分别产生组合逻辑和时序逻辑，状态编码使用parameter语句；
- always语句中的敏感量表要完整，但也不要多余；
- verilog组合逻辑描述不要引起多余的或者不期望的Latch；
- 除了仿真调试用的语句外，要采用可综合的描述。仿真调试用的部分可以加注编译指令“synopsys translate_off”；
- 组合逻辑采用阻塞赋值(=)，时序逻辑采用非阻塞赋值(<=)；
- 一个block中不要混合使用阻塞和非阻塞赋值；
- 复杂表达式要采用括号分割；
- 每行只写一个verilog语句；

良好的Verilog可综合代码风格(续)

- 每个子模块的输出信号最好是DFF的寄存输出；
- 尽可能不要对信号赋值x，不要使用casex语句；
- 注释：采用简洁的注释，包括文件头注释，语句模块注释；文件头注释应包括：Filename，Author、VersionHistory，Function Description，parameter等；
- 建议参考synopsys的文档：
  - “Guide to HDL Coding Styles for Synthesis”
  - “HDL Compiler for Verilog Reference Manual”