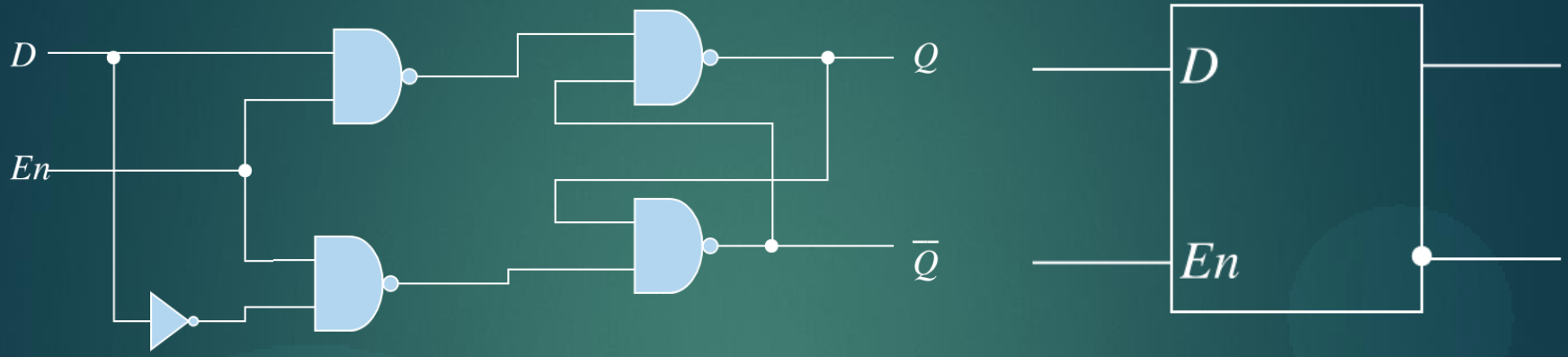
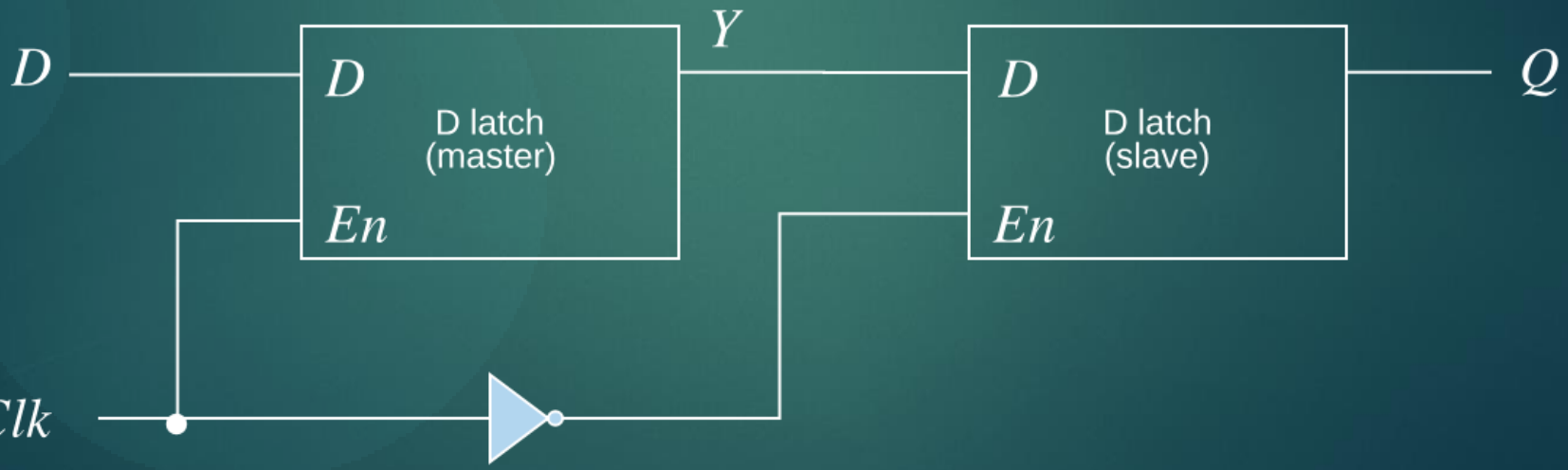
《数字逻辑》
Digital Logic

Verilog介绍（2）

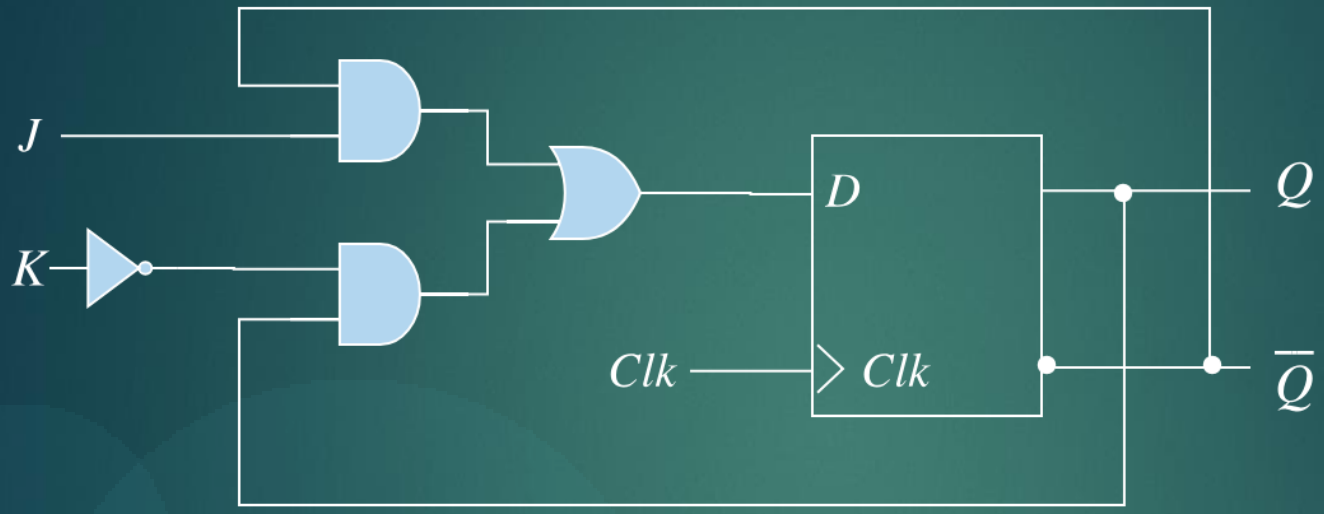
北京工业大学软件学院
王晓懿

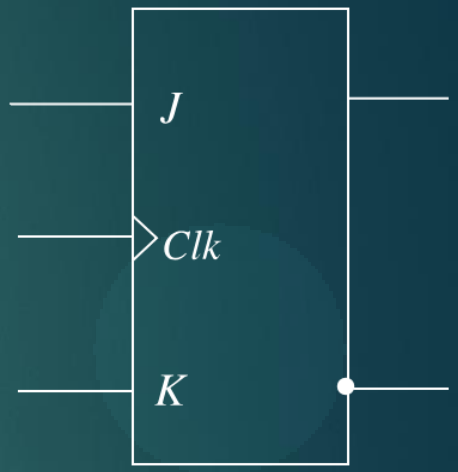
触发器回顾I

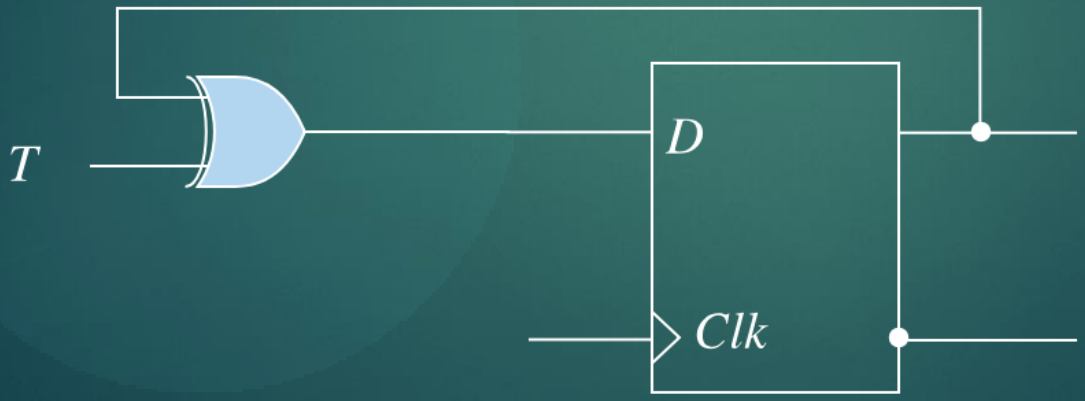

D
En
Q
$\overline{Q}$
D
En

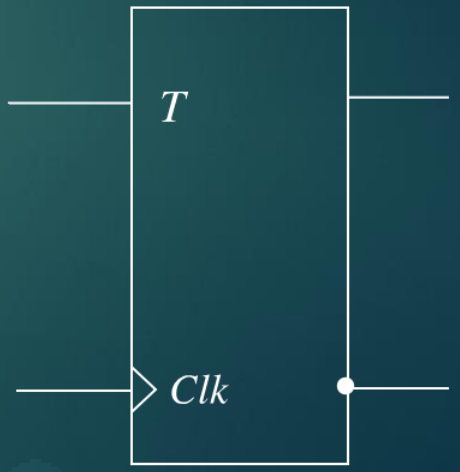

D
D
D latch
(master)
En
Y
D
D latch
(slave)
En
Q
Clk

触发器回顾II









Verilog描述触发器

```
module D_latch (output reg Q,
                input D, enable);
  always @ (enable or D)
    if (enable) Q <= D;
endmodule
```

```
module D_FF (output reg Q,
             input D, Clk);
  always @ ( posedge Clk)
    Q <= D;
endmodule
```

```
module JKFF ( output reg Q,
      input J, K, Clk, rst);
  wire JK;
  assign JK = (J & ~Q) | (~K & Q);
  DFF JK1 (Q, JK, Clk, rst);
endmodule
```

```
module JK_FF (output reg Q, output Q_b,
                        input J, K, Clk);
  assign Q_b = ~ Q ;
  always @ ( posedge Clk)
    case ({J,K})
      2'b00: Q <= Q;
      2'b01: Q <= 1'b0;
      2'b10: Q <= 1'b1;
      2'b11: Q <= !Q;
    endcase
endmodule
```

Verilog描述时序逻辑

- Always语句用来描述状态变量的变化。逻辑综合工具会自动将其综合为时序元件

同步D触发器

```
module D_FF (output reg Q,
    input D, clk, set, rst);
  always @ ( posedge clk)
    if (rst)
      Q <= 1'b0;
    else if (set)
      Q <= 1'b1;
    else
      Q <= D;
endmodule
```

关键字：上升沿

always @(posedge clk)是认定为时序逻辑的关键

可直接映射到FPGA的触发器

“always”语句回顾

- always 语句
  - 总是等待触发信号的变化
  - 触发信号变化即开始执行

```
module and_gate (output reg out,
                 input in1, in2);
  reg result;
  always @(in1, in2) begin
      result = in1 & in2;
      out = result;
  end
endmodule
```

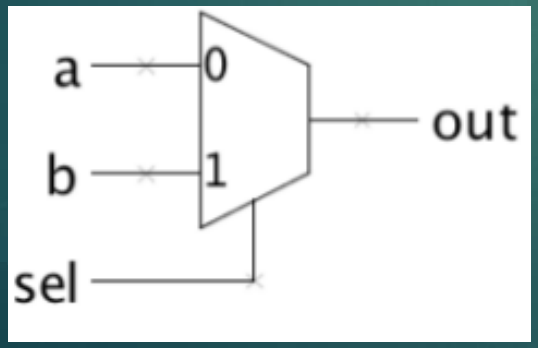
always块中左端项必须为reg类型，但是并不一定就是寄存器！

声明了触发信号，即语句执行的时机

always组合逻辑与时序逻辑

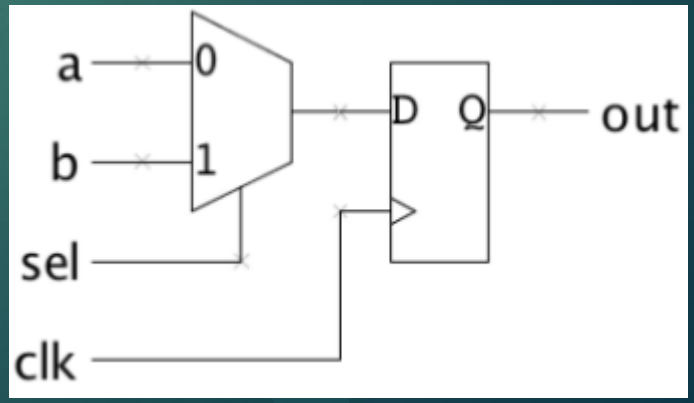
组合逻辑

```
module comb(output reg out,
            input a, b, sel);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



时序逻辑

```
module seq(output reg out,
     input a, b, sel, clk);
  always @(posedge clk) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```



always触发事件（敏感列表）

- 边沿触发posedge, negedge表明always语句描述的是时序逻辑；
- 与组合逻辑不同，时序逻辑中敏感列表会影响电路综合结果；
- 一个变量只在一个always块中设置，避免竞争冒险；

同步重置的D触发器

```
module D_FF (output reg Q,
    input D, clk, set, rst);
  always @ ( posedge clk)
    if (rst)
      Q <= 1'b0;
    else if (set)
      Q <= 1'b1;
    else
      Q <= D;
endmodule
```

异步重置的D触发器

```
module D_FF (output reg Q,
    input D, clk, set, rst);
  always @ ( posedge clk, posedge set,
    negedge rst)
    if (!rst)
      Q <= 1'b0;
    else if (set)
      Q <= 1'b1;
    else
      Q <= D;
endmodule
```

隐藏的锁存器

- 不完整的敏感列表会导致在逻辑综合时为遗漏的触发信号生成额外的锁存器

```
module and_gate (output reg out,
                 input in1, in2);
  always @(in1) begin
    out = in1 & in2;
  end
endmodule
```

in2遗漏了！

会为生成一个锁存器保存in2的值，这样in2的值变化就不会影响本always块

一个技巧：在描述组合逻辑的always块中总是使用@(*)

隐藏的锁存器 II

- 不完整的case语句也会导致生成额外的锁存器
- 要保证每个输出都被赋值，否则会生成锁存器保存上次的输出

缺了sel为2'd2的情况

sel为2'd2时out无赋值

逻辑综合时会增加锁存器保存上次out，当sel为2'd2时输出此值

使用if的实现也有同样现象

```
module mux4to1 (output reg out,
                input a, b, c, d
                input [1:0] sel);
always @(*) begin
   case (sel)
      2'd0: out = a;
      2'd1: out = b;
      2'd3: out = d;
   endcase
end
endmodule
```

隐藏的锁存器

- case语句中列出所有的情况，就可以避免生成不必要的锁存器
- 或者，直接在case中使用default语句，给出默认值

```
default:  out = in;
```

- 如果不关心某些情况的取值，可以使用x作为默认值

```
default:  out =1'bx;
```

注意，当使用不管项x的时候，逻辑综合得到的电路可能不同！

reg类型

- always块中赋值的左端必须是reg类型
- always块中赋值的输出变量也必须是reg类型
- 在always触发之前reg类型数据保持不变

```
module and_gate (output reg out,
                 input in1, in2);
  reg result;
  always @(in1, in2) begin
      result = in1 & in2;
      out = result;
  end
endmodule
```

不一定是寄存器！

Wire类型和reg类型

- 使用wire类型还是reg类型？
  - 如果信号要在always块中被赋值，则必须使用reg类型
  - 如果信号要被连续赋值，则必须是wire类型
  - 模块输入输出信号默认是wire类型，如果要声明为reg类型，需要显式的加上reg声明
- 如何知道信号是用线网信号还是寄存器信号？
  - wire信号只能表示组合逻辑的连接
  - reg信号在always @(*)中也是线网
  - reg信号在always @(posedge clk, negedge clk) 中表示寄存器

连续赋值和非连续赋值

```
assign R = X | (Y & ~Z);

assign r = &X;

assign R = ( a == 1'b0) ? X : Y;

assign P = 8'hff;

assign P = X * Y;

assign P[7:0] = { 4{X[3]}, X[3:0] };

assign {cout, R} = X + Y + cin;

assign Y = A << 2;

assign Y = {A[1], A[0], 1'b0, 1'b0};
```

位操作符

reduction操作符

算术操作符

扩展操作符和拼接操作符

移位操作符

连续赋值和非连续赋值

- 使用always块中的赋值
- 某些仿真器中表现有不同

```
module and_gate (output reg out,
                 input in1, in2);
  reg result;
  always @(in1, in2) begin
    result = in1 & in2;
    out = result;
  end
endmodule
```

触发信号

赋值输出

阻塞式赋值和非阻塞式赋值

- always块中赋值分为阻塞式赋值和非阻塞式赋值

- 阻塞式赋值：=

赋值完成前不往下执行

```
always @(*) begin
    x = a | b;
    y = a ^ b ^ c;
    z = b & ~c;
end
```

- 非阻塞式赋值：=

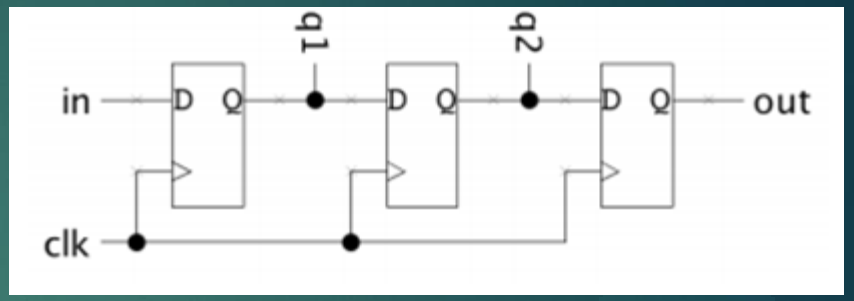
右端表达式同时开始求值，时间步长结束后同时赋值

```
always @(*) begin
    x <= a | b;
    y <= a ^ b ^ c;
    z <= b & ~c;
    //时间步长结束，才开始赋值
end
```

有时候两种赋值逻辑综合得到的结果完全不同！

时序电路中的赋值

- 实现右图的时序电路：



- 下面使用非阻塞赋值和阻塞赋值语句逻辑综合的结果相同吗？

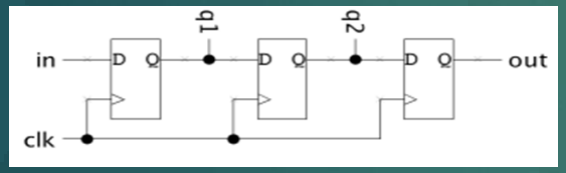
```
module nonblocking(output reg out,
                   input in, clk);
  req q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

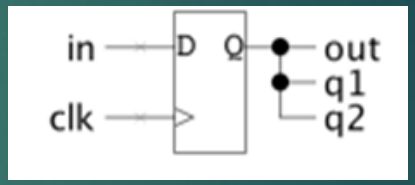
```
module blocking(output reg out,
                input in, clk);
  req q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```

时序电路中使用非阻塞赋值

```
module nonblocking(output reg out,
                   input in, clk);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule
```

```
module blocking(output reg out,
                input in, clk);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```





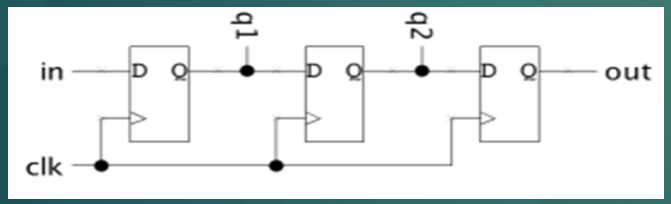
- 阻塞式赋值并不反映时序逻辑的本质行为

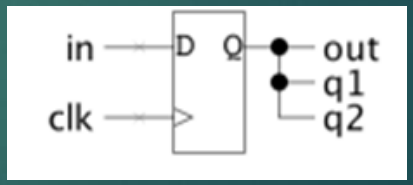
在描述时序逻辑时，使用非阻塞式赋值语句！

时序电路中使用非阻塞赋值

```
module blocking(output reg out,
                input in, clk);
  reg q1, q2;
  always @(posedge clk) begin
    out = q2;
    q2 = q1;
    q1 = in;
  end
endmodule
```

```
module blocking(output reg out,
                input in, clk);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```





锁存器与触发器

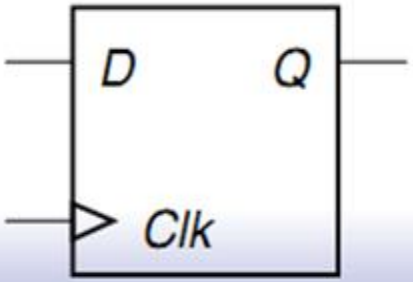
- 触发器
- 锁存器

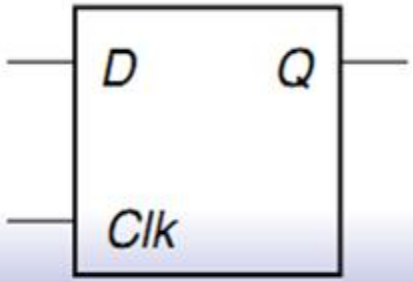
```
module flipflop
(
  input clk,
  input d,
  output reg q
);

 always @(posedge clk)
 begin
   q <= d;
 end
endmodule
```

```
module latch
(
  input clk,
  input d,
  output reg q
);

 always @(clk or d)
 begin
  if ( clk )
    q <= d;
 end
endmodule
```





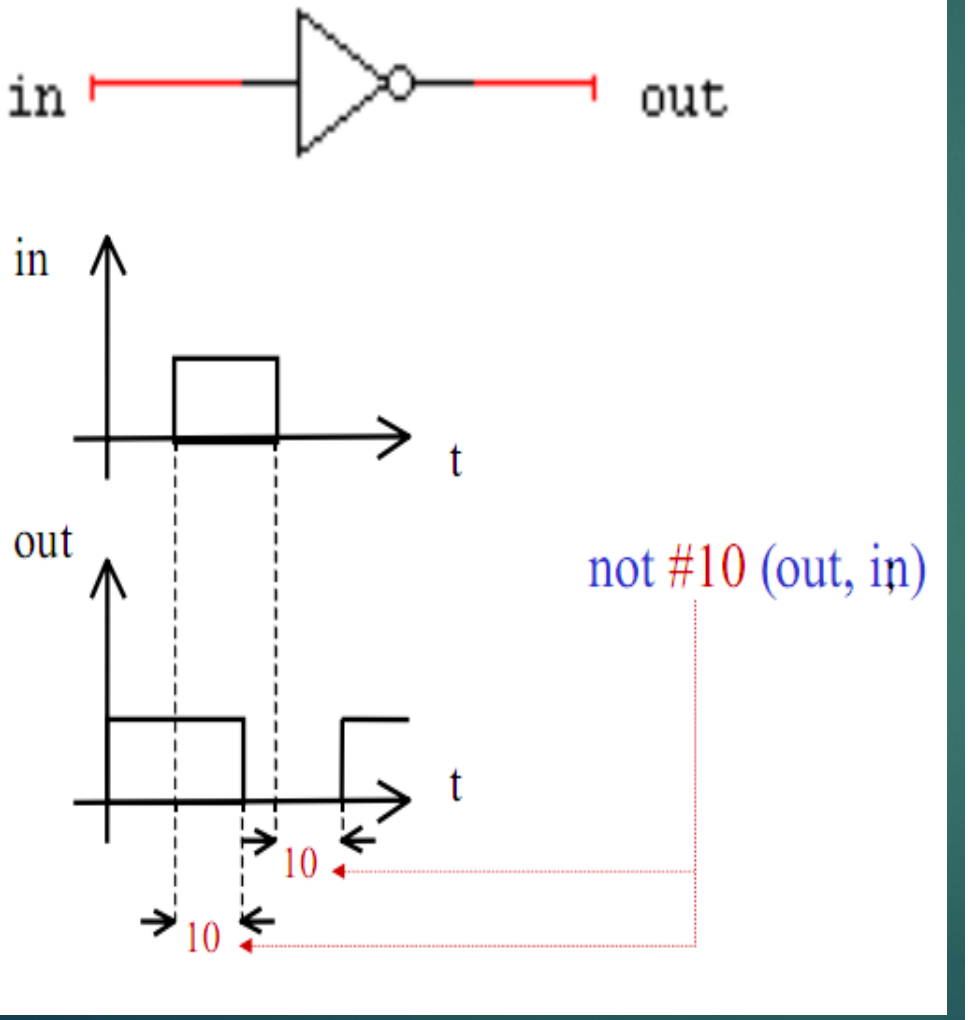
Verilog中的延迟

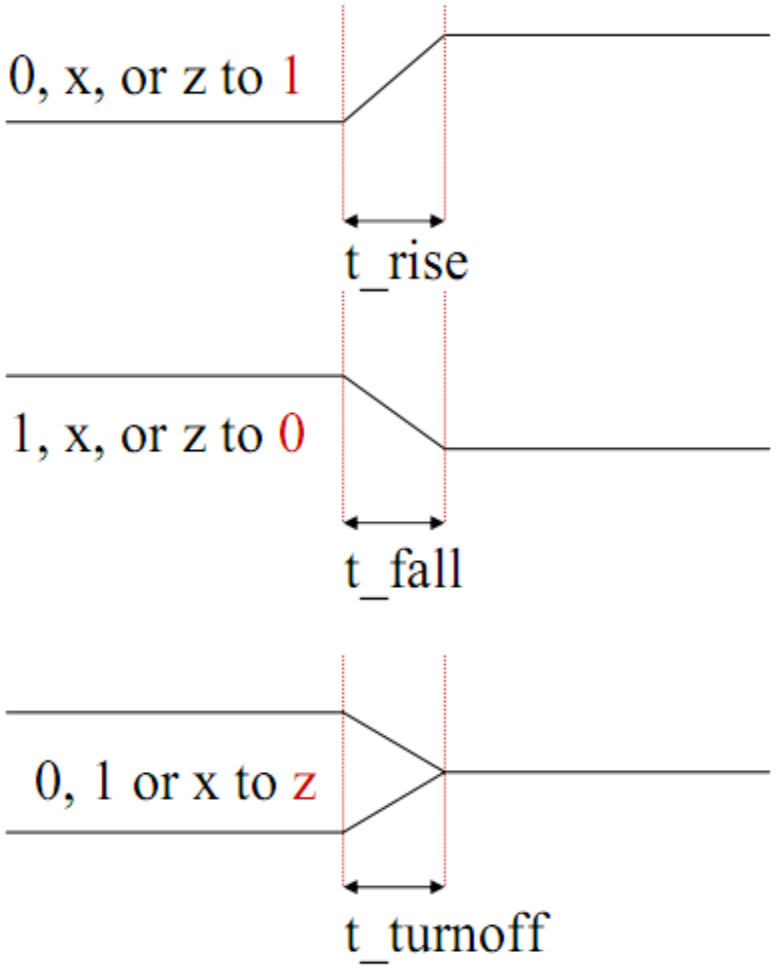
延迟	含义
#t	延迟t
#(tr,tf)	(上升延迟，下降延迟)
#(tr,tf,toff)	(上升延迟，下降延迟，关断延迟)

- **not #10 (out,in)**
- **assign** #5 o = ~i
- **wire** #(5) ready;
- **#t var = expr;**
- **var = #t expr;**

如果指明的延迟值小于三个，那么没有指定的延迟默认为指定的延迟值中最小的那个

延迟


in
out
in
t
out
t
not #10 (out, in)
10
10


0, x, or z to 1
t_rise
1, x, or z to 0
t_fall
0, 1 or x to z
t_turnoff

Clock信号

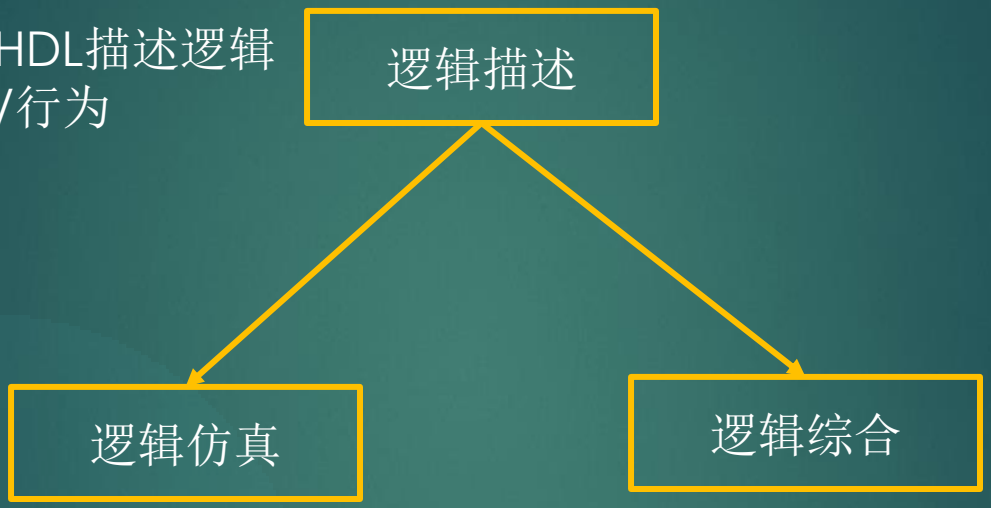
```
initial begin
   clock = 1'b0;
   repeat (30)
   #10 clock = ~clock;
end
```

```
initial begin
clock = 1'b0;
end
initial 300 \$ finish ;
always #10 clock = ~clock;
```

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

逻辑综合介绍

使用HDL描述逻辑结构/行为



逻辑功能验证

将逻辑描述映射到实现资源，如FPGA，ASIC等

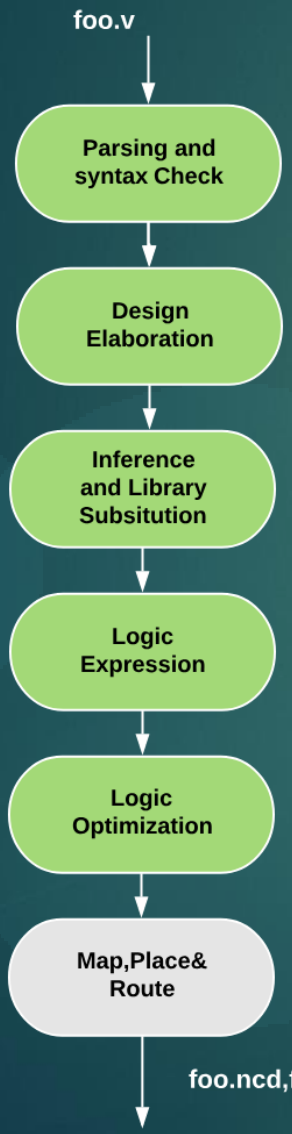
逻辑综合介绍

- Verilog和VHDL开始是作为用于逻辑仿真的语言被提出的。但是很快就开始有程序工具被发明出来用于自动化的将Verilog程序自动转换为底层的电路实现（电路网表）
- 逻辑综合将Verilog（或其他硬件描述语言）代码转换为可实现的电路原语：
  - FPGAs: 查找表（LUTs）,触发器， 内存（RAM）
  - ASICs （Application Specific IC）: 标准单元，触发器，内存

逻辑综合的特点

- 逻辑综合工具提高了设计的自动化水平
  - bug更少
  - 更高的设计效率
- 可以将逻辑描述（HDL）与底层实现技术（FPGA，ASIC）分离开来，有利于设计的快速原型到产品定型
- 部分情况下可以获得比人工综合更好的结果（比如逻辑化简）
- 很多情况下逻辑综合无法得到最优结果
- 不能完全做到对设计透明： 在逻辑描述是还是要考虑逻辑综合工具，才能得到性能较好的综合结果。

逻辑综合步骤



加载HDL代码文件，进行预处理，解析检查语法…

确定参数，生成循环代码，创建模块，连接端口…

识别并插入触发器，运算器，内存等器件

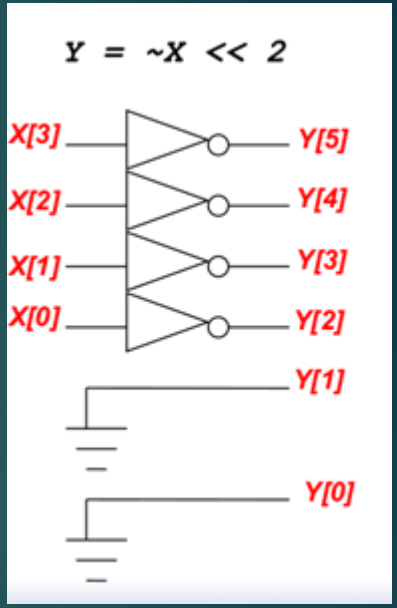
将组合逻辑映射到基本门电路实现

在设计约束下对逻辑电路进行化简

将逻辑电路映射到实现技术（FPGA，ASIC），布局布线…

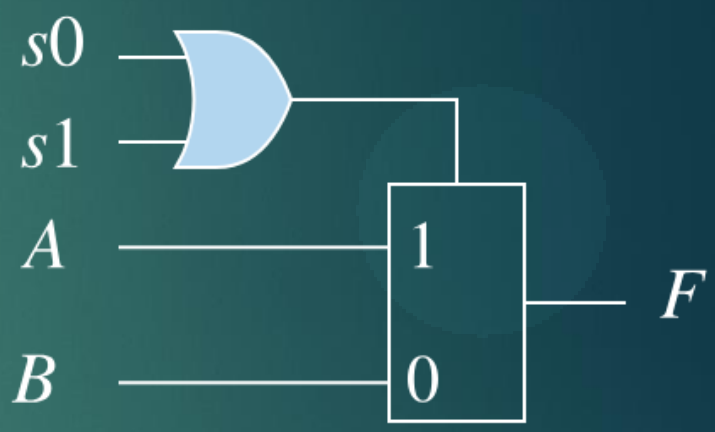
操作符综合

- 操作符被映射为基本门电路
- 算术运算符被映射到加法器、乘法器等运算器
  - 考虑进位，输出信号要比输入信号多一位
- 比较关系操作符被映射为比较器
- 固定位数移位操作符被映射为线网的（错位）连接
- 变量移位操作符被映射为移位寄存器
- 条件操作符被映射为逻辑操作或者选择器



逻辑综合示例I

```
module foo (output reg [3:0] F,
            input [3:0] A, B,
            input s0, s1);
  always @(*)
    if (!s0 && s1 || s0)
      F = A;
    else
      F = B;
endmodule
```

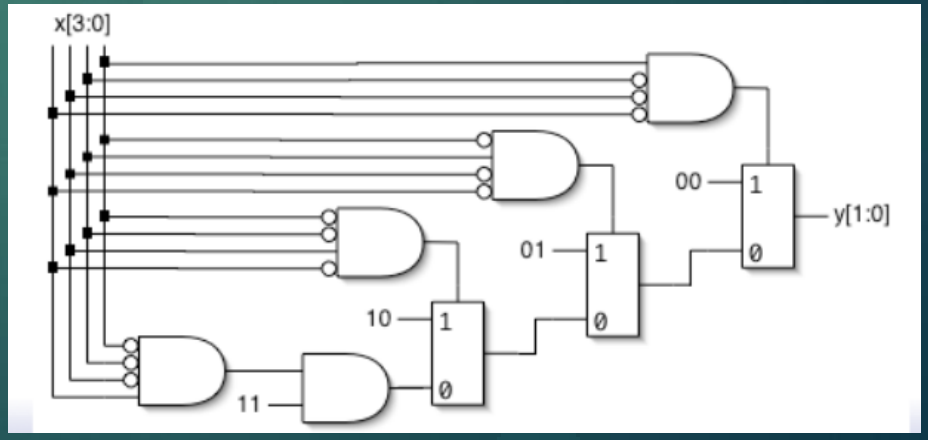


将if-else映射为选择器，并进行了逻辑优化

逻辑综合示例II：编码器

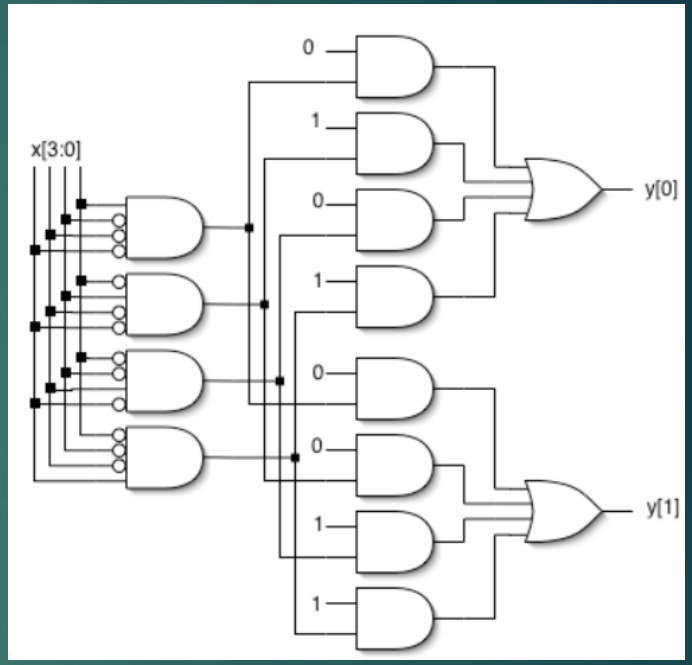
```
always @(x) begin
   if (x == 4'b0001) y = 2'b00;
   else if(x == 4'b0010) y = 2'b01;
   else if(x == 4'b0100) y = 2'b10;
   else if(x == 4'b1000) y = 2'b11;
   else y = 2'bxx;
end
```

嵌套的if-else会生成带有优先级的电路，可能带来性能问题



逻辑综合示例II：编码器

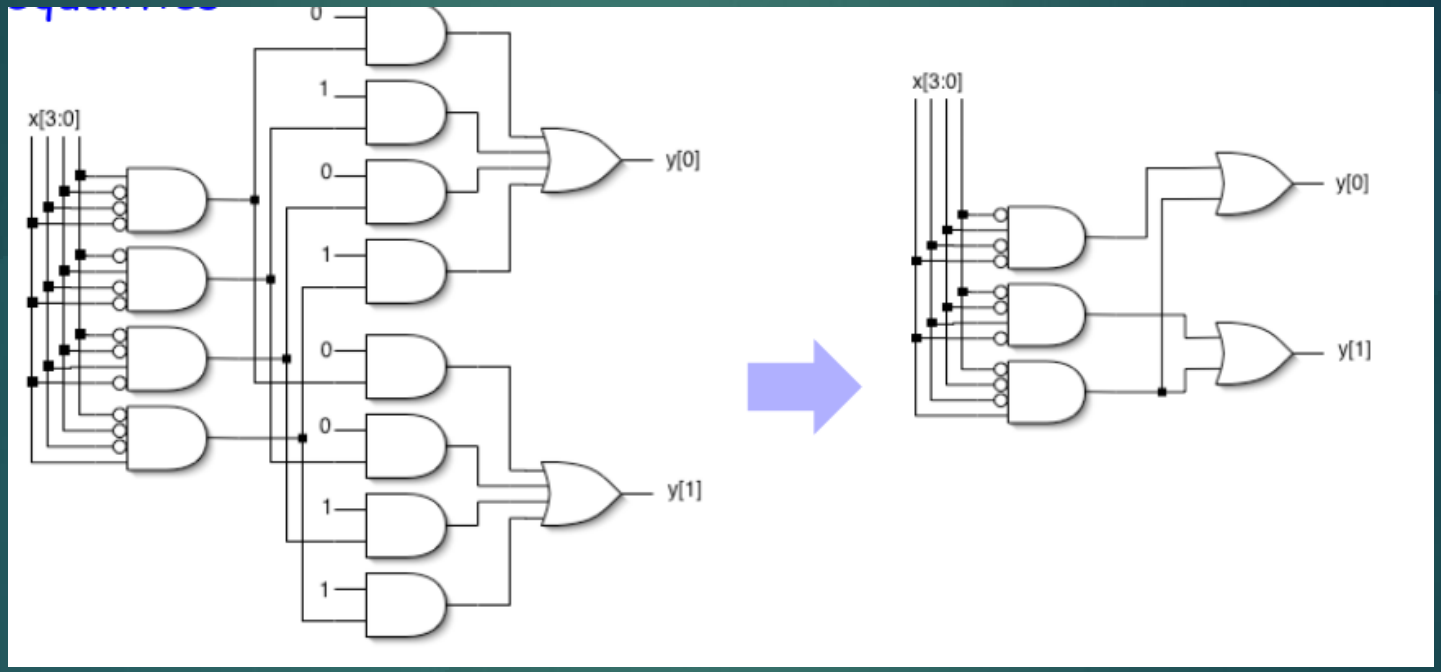
```
always @(x) begin
  case (x)
    4'b0001: y = 2'b00;
    4'b0010: y = 2'b01;
    4'b0100: y = 2'b10;
    4'b1000: y = 2'b11;
    default: y = 2'bxx;
  endcase
end
```



使用case语句，实现所有比较的并行执行

逻辑综合示例II：编码器

逻辑化简



高层次综合（HLS）

Cadence高层次综合工具：Stratus HLS