数字电路

Digital Circuits and System

李文明

liwenming@ict.ac.cn

中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

中国科学院大学
University of Chinese Academy of Sciences

硬件描述语言

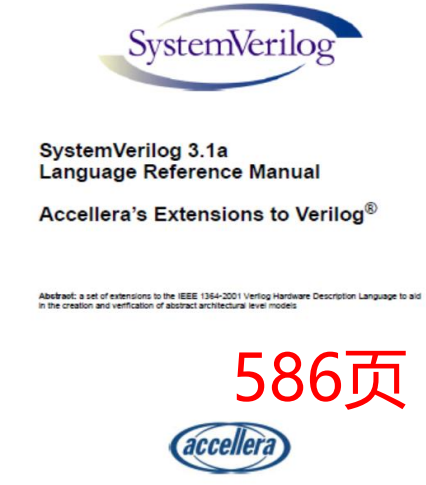
Reference

- David Culler, EECS 150 - Components and Design Techniques for Digital Systems
- Ronald Fearing, EECS150 - Digital Design, http://www-inst.eecs.berkeley.edu/~cs150
- Philip Levis, EE108A: Digital Systems I, http://web.stanford.edu/class/archive/ee/ee108a/ee108a.1082/
- Verilog® HDL Quick Reference Guide, Sutherland HDL, Inc.
- IEEE Std 1364-2001, IEEE Standard Verilog® Hardware Description Language, IEEE Computer Society, Sponsored by the Design Automation Standards Committee

HDL发展过程

- 1985年起源于Gateway，1989年被Cadence收购
- 起初作为仿真语言，后成为可综合，Berkeley在80年代开发，90年代商用
- 同一时代，VHDL在DoD支持公开使用，使其快速流行
- 为避免失去市场份额，Cadence公司1990也把Verilog公开
- 1993年IEEE成立标准工作组，于1995年发布IEEE-1394(Verilog)，目前标准是IEEE 1364-2001
- 硅谷的大部分公司采用Verilog，类C语言，具有更广的仿真工具软件支撑
- VHDL仍在大学、欧洲、日本等使用
- 目前大多数EDA工具对两种语言都支持
- 最新的Verilog语言是System Verilog
- 最新的HDL：基于C++，OSCI(Open System C Initiative)

791页



586页

Verilog与VHDL比较

VHDL：采用自顶向下原则，把一个设计分解成小块“Components”

- Entity：描述接口信号和基本构件模块
- Architecture：描述行为，每个Entity可以包括多个Architecture
- Configuration：部件列表，其行为可被Entity使用
- Package：用于构建一个设计的工具箱

Verilog：整个设计一个构建模块

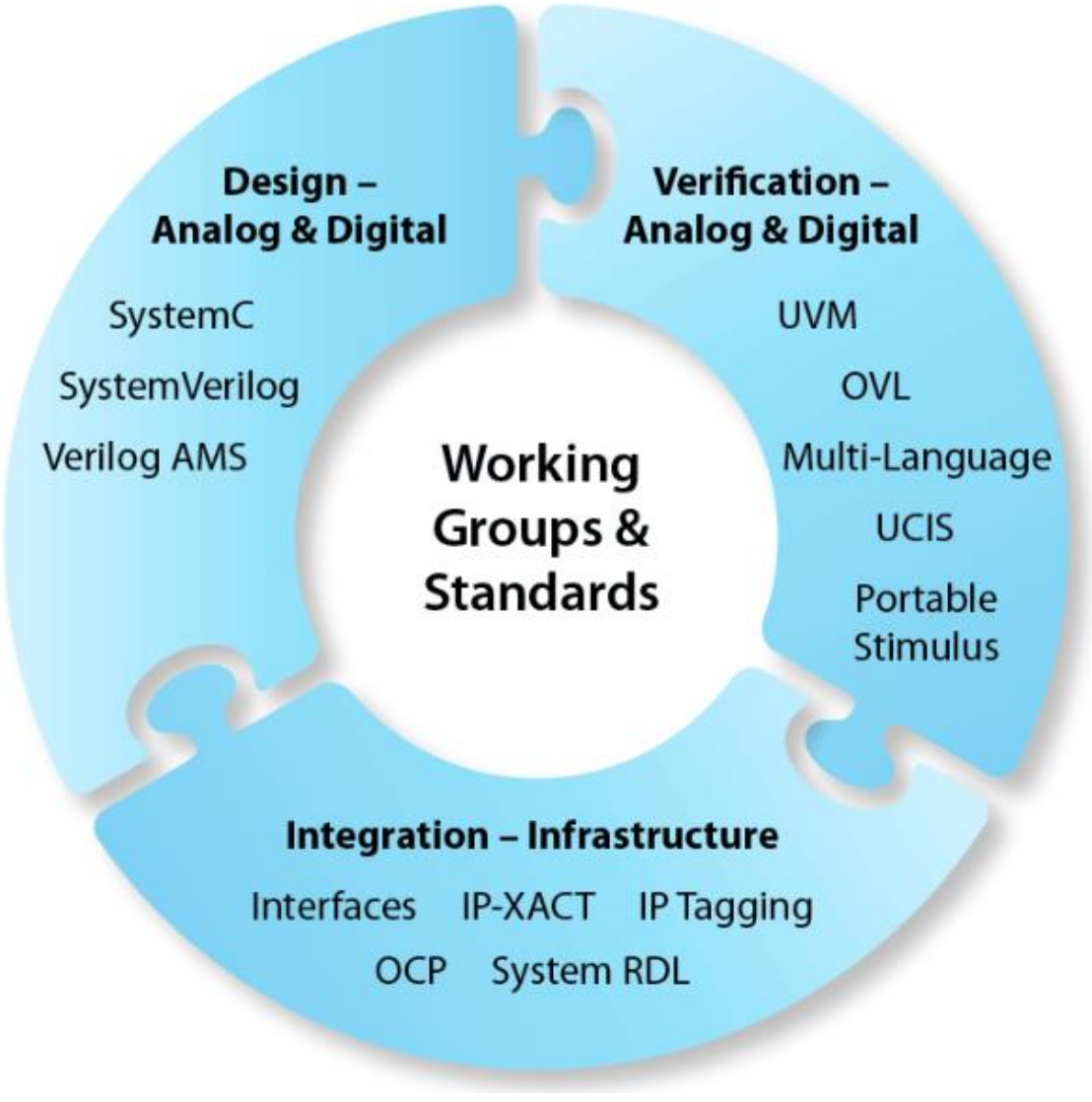
- Module：通过端口实现模块间连接
- 一般每个文件描述一个Module
- 最顶层Module通过例化包含其他Module
- Module可以采用行为级，或结构化方法描述电路
- 行为级描述规范定义的数字系统行为
- 结构规范采用分级的方法连接各个子模块

Verilog与VHDL的相似性

- 两种语言都可以实现从顶层逻辑抽象，到底层逻辑实现细节的描述
- “仿真”和“综合”是两种最基本的设计工具都可以采用Verilog和VHDL语言类似实现
- 语言本身不是设计工具集，也不是设计方法，但语言必须与工具集和设计方法结合发挥作用
- 最新的语言：System C, Universal Verification Methodology(UVM), 和IP-XACT

Accellera 生态系统

https://www.accellera.org/downloads



Verilog与VHDL是否有区别?

- 从语言的能力角度看，两者没有区别

- 语言的选择主要取决于设计或者自身的偏好，以及能够获得的可用工具链或商业上的考虑等

- VHDL相对“难”学，像ADA语言
- Verilog比较“容易”学，像C语言

综述

- 摩尔定律，芯片内晶体管数每18个月翻一翻，新技术的推动改变了设计取舍的原则
- ASIC，可编程器件，微处理器技术
- 可编程逻辑器件，缩短了设计开发和上市时间
- FPGA技术
  - 互连可编程
  - 逻辑可配置：LUT+存储
  - 块RAM
  - IO块

提纲

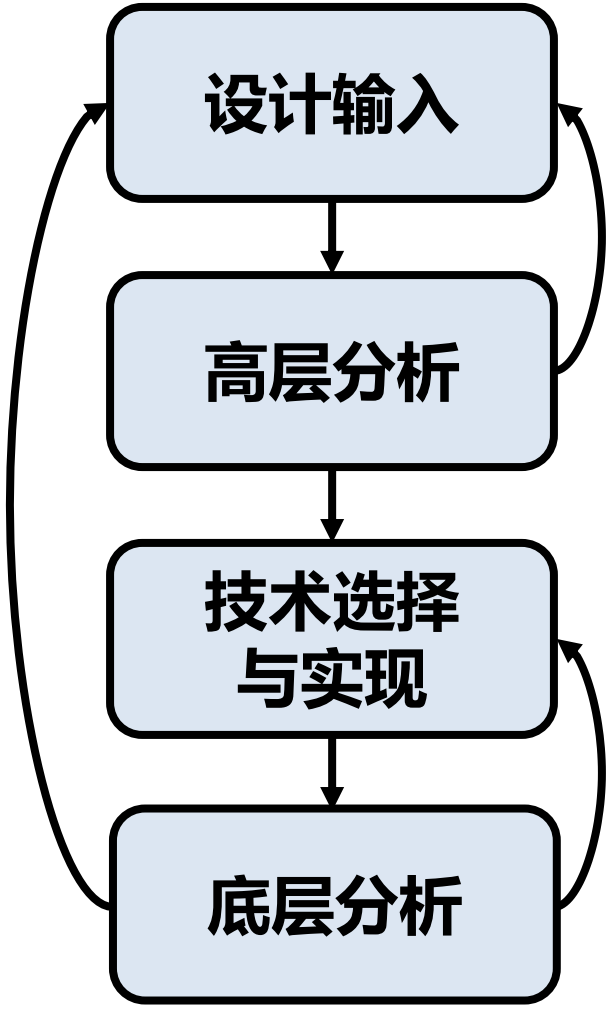
- 网表
- 设计流程
- 什么是HDL
- Verilog
  - 声明
  - 结构模型
  - 行为模型
  - 语言的元素
  - 若干举例

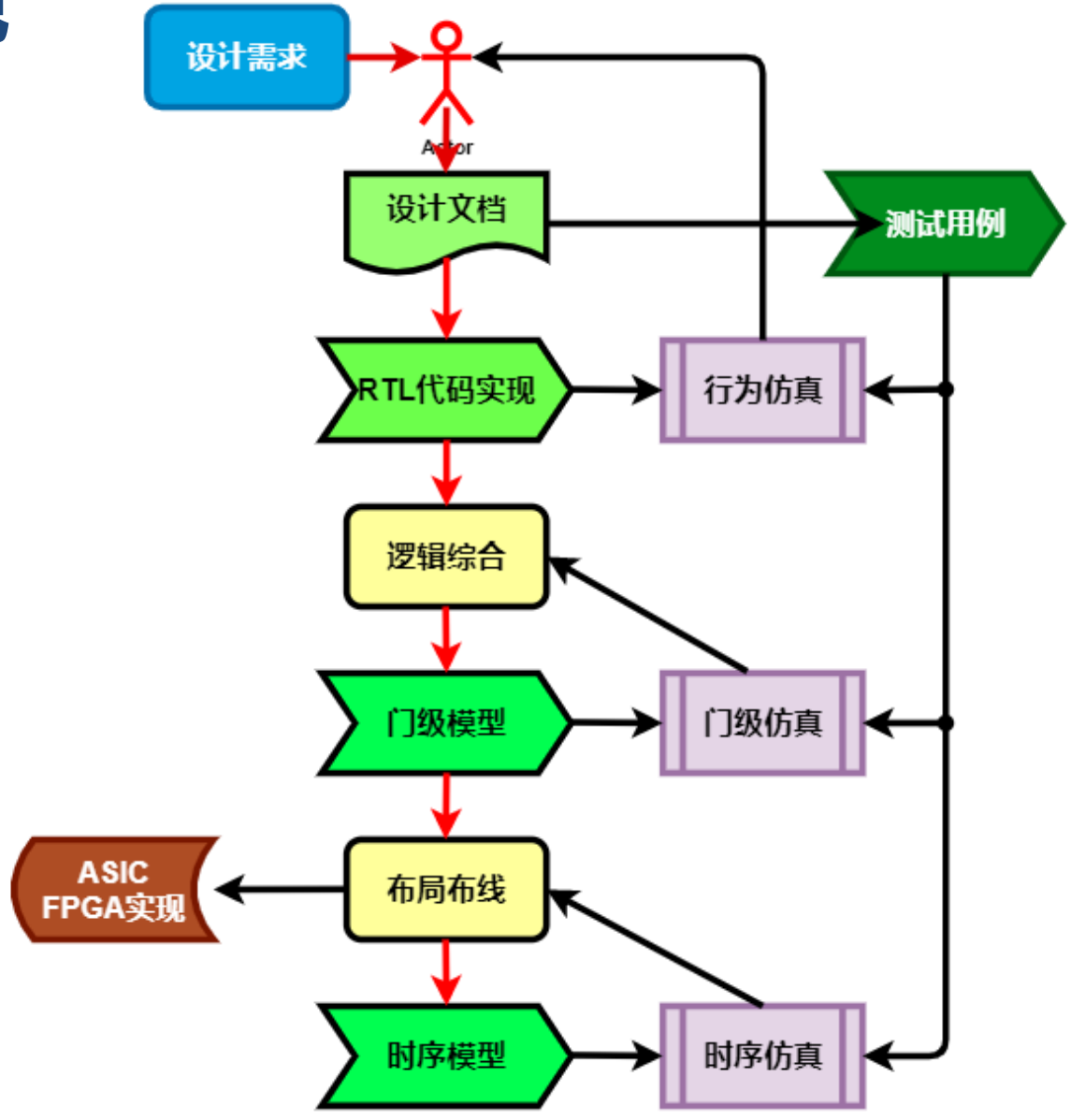
牢记：设计是一种表示

- 如何表示一个数字逻辑系统?
- 部件
  - 逻辑表达式，真值表
  - 存储的符号，时序图
- 连接关系
  - 电路图

- 最重要的问题：面向人的可读性，还是面向机器的可读性?

设计流程-从需求到实现

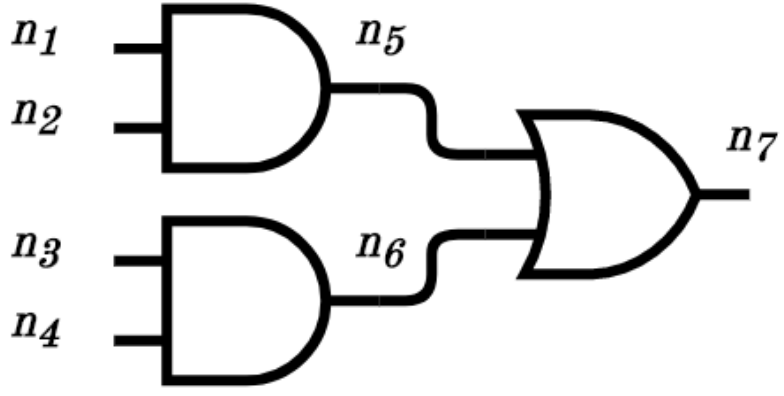

设计输入
高层分析
技术选择
与实现
底层分析


设计需求
Actor
设计文档
测试用例
RTL代码实现
行为仿真
逻辑综合
门级模型
门级仿真
ASIC
FPGA实现
布局布线
时序模型
时序仿真

网表(Netlist)

- 网表是设计过程中实现设计表示的关键数据结构，网络连接表
- 网表是部件端点、以及端点间连接关系的数据表示
- 逻辑仿真和实现时都需要网表
- 网表可以是门级，也可以是晶体管级
- 可以是层级的，也可以是无层级平铺的
- 问题：如何生成网表?

g1 “and” n1 n2 n5
g2 “and” n3 n4 n6
g3 “or” n5 n6 n7

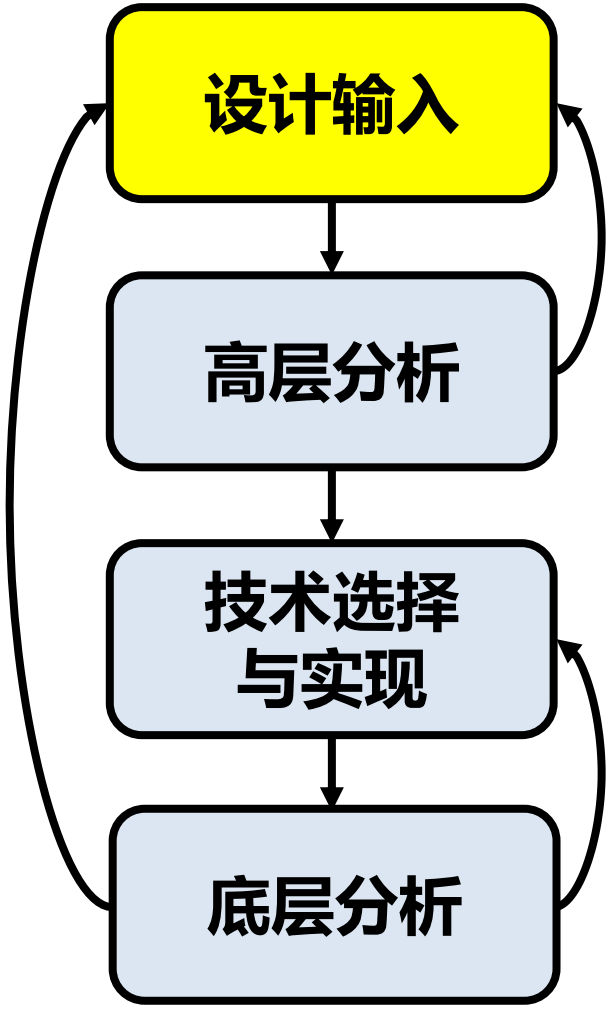


另一种格式
n1 g1.in1
n2 g1.in2
n3 g2.in1
n4 g2.in2
n5 g1.out g3.in1
n6 g2.out g3.in2
n7 g3.out
g1 “and”
g2 “and”
g3 “or

设计流程—电路如何表示

- 电路的描述和表示方法
  - 几何法：电路图
  - 文本法：HDL

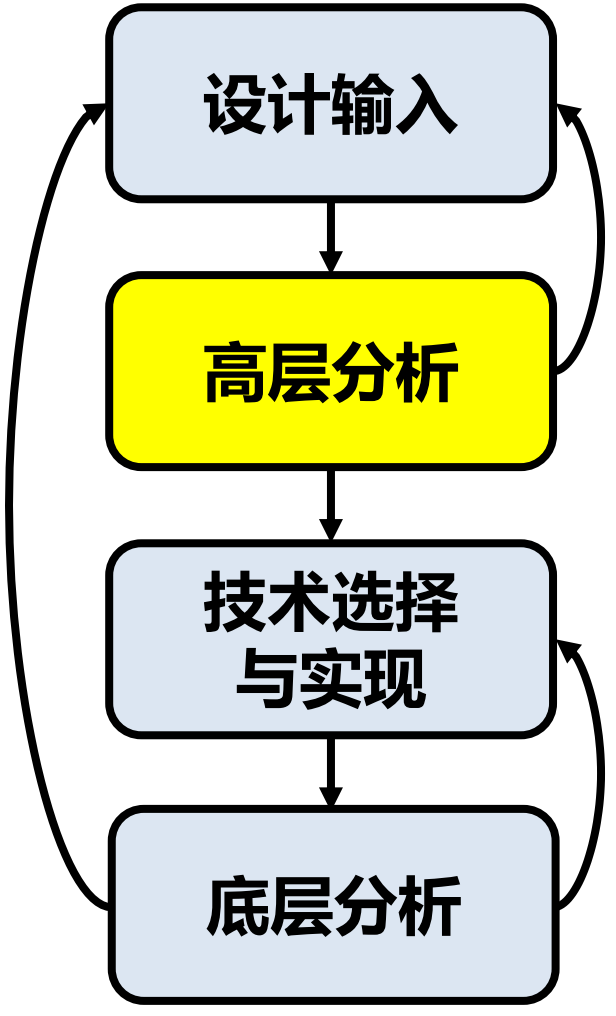
- 电路的规范和编译结果是网表
  - 通用部件：逻辑门、触发器
  - 与特定技术相关的部件：LUT/CLBs，晶体管、分散门
  - 高层库原件：加法器、ALUs、寄存器堆、译码器等



设计流程—高层分析

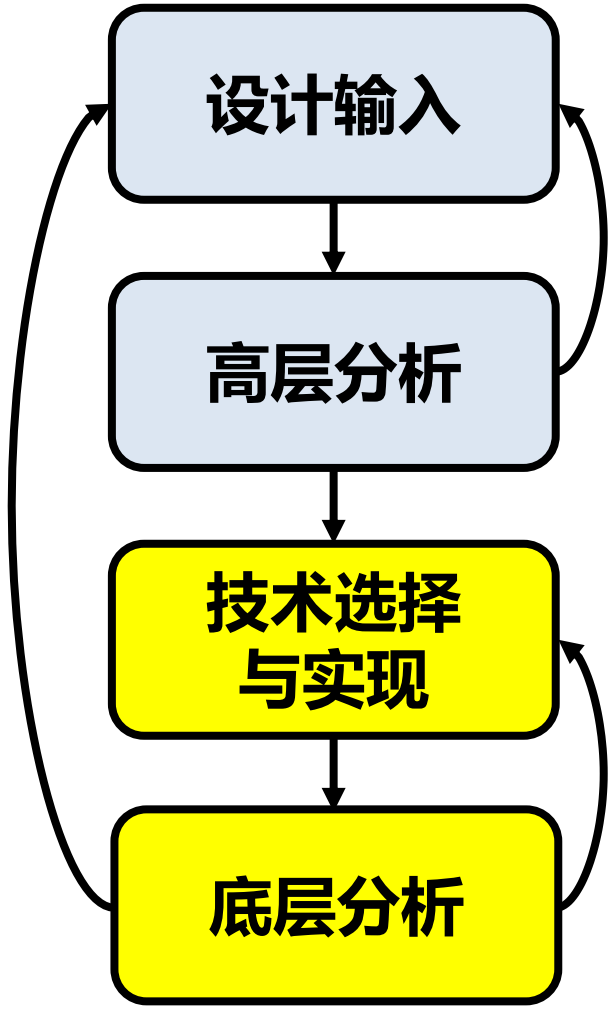
- 高层分析的目的
  - 功能正确性
  - 初步进行
    - 时序分析
    - 功耗分析
    - 成本分析（资源占用、芯片规模、实现方式）

- 采用的通用工具
  - 模拟器：检查功能正确性
  - 静态时序分析：基于时间模型、以及库原件参数初步估算电路延迟时间



设计流程—实现与底层分析

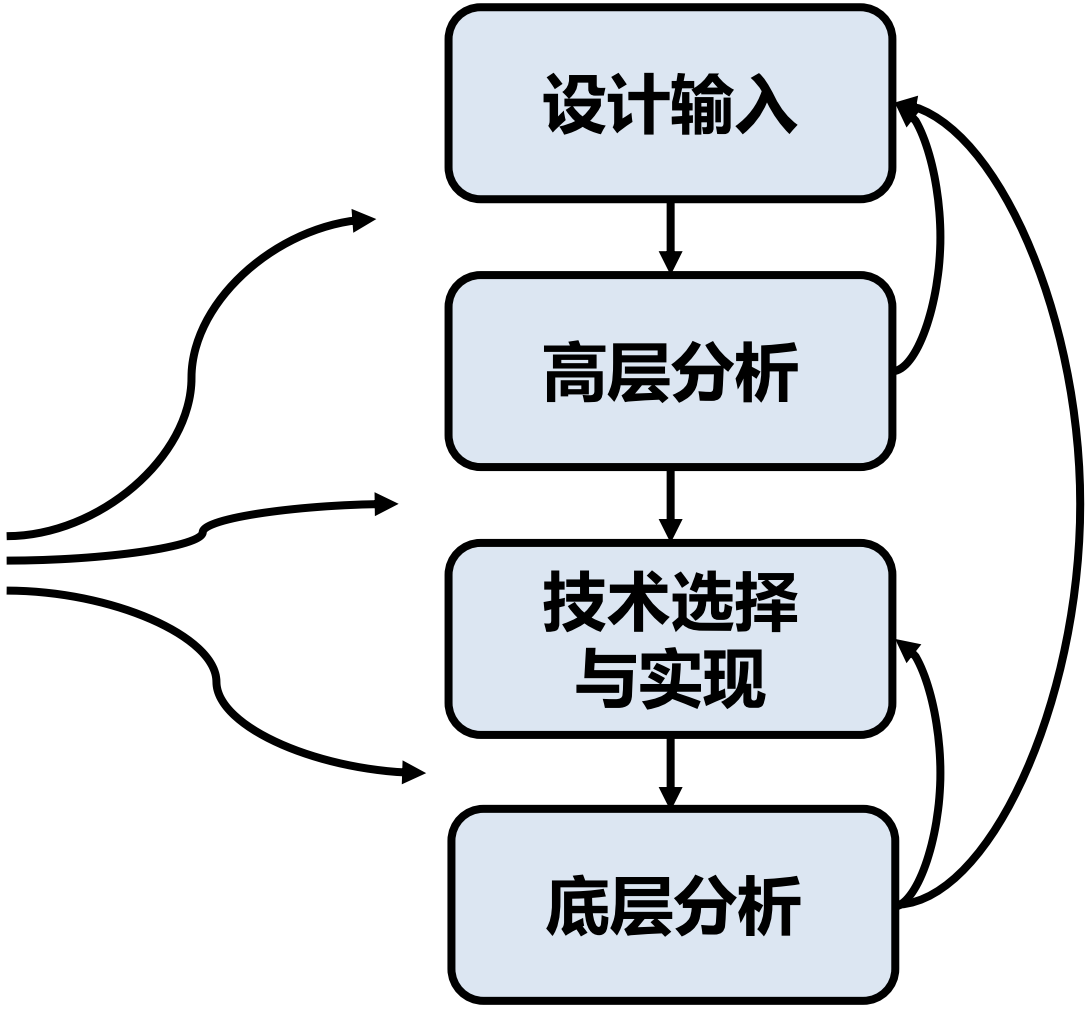
- 芯片实现（技术相关）
  - 把网表转化成详细的技术实现
    - 扩展库原件
    - 性能相关的细节：分区、摆放、连线
- 底层分析
  - 仿真和分析工具用来实现底层检查
    - 精确的时序模型
    - 线延时
  - 对FPGA来说，这一步可以用实物芯片测试



网表在设计流程中的作用

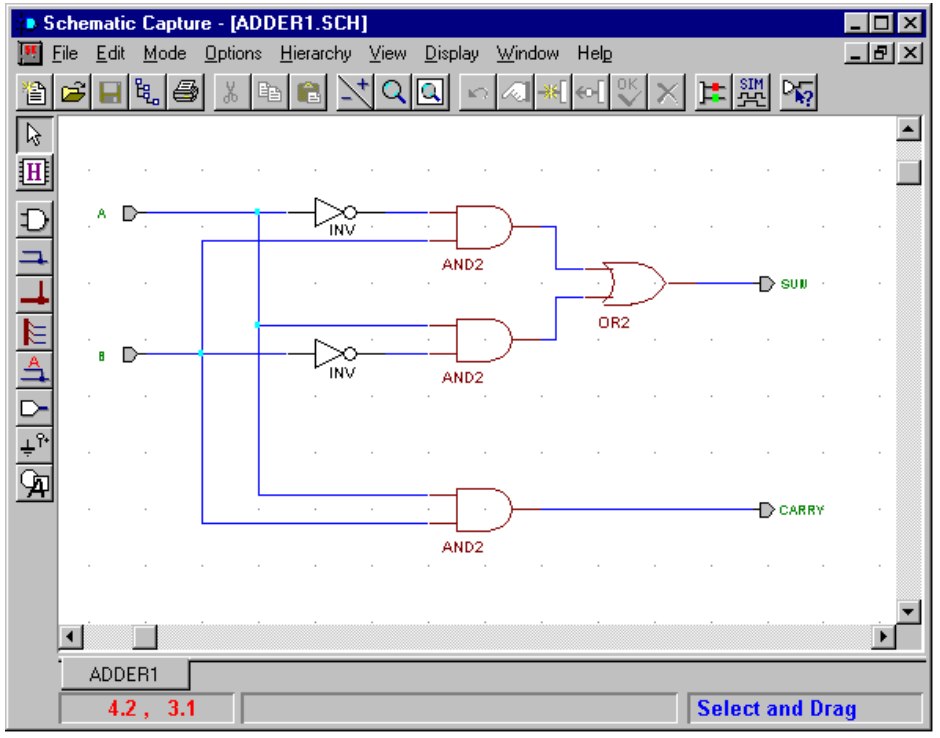
网表：

用来实现各设计步骤内部、以及设计步骤间的数据交互



设计输入

- 电路图是工业界广泛采用的标准，其特点为
  - 电路图直观，能很好匹配门级或框图级描述
  - 一定程度上是面向物理实现的
  - 需要专用的编辑软件来画
  - 对复杂系统，除非使用很好的层级设计方法，电路图很难被理解



- 芯片设计领域，硬件描述语言(HDLs)是新的标准

- 但对于板级设计，仍使用电路图

硬件描述语言(HDLs)

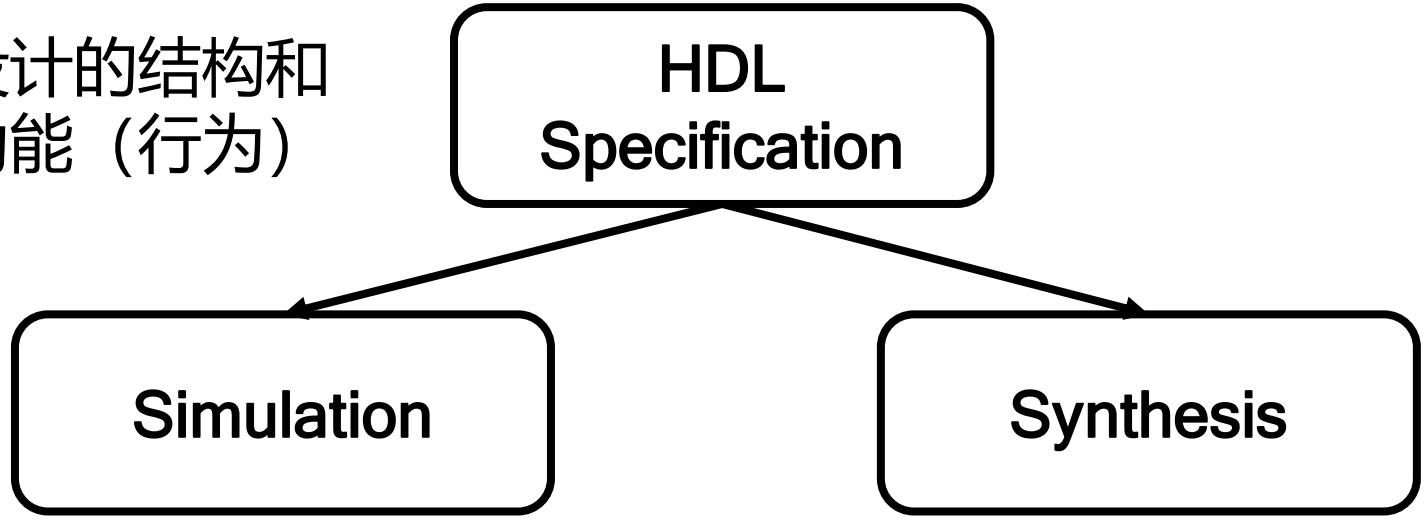
- 基本想法
  - 采用两种基本形式的语言结构来描述电路
  - 结构描述类似于于层级网表
  - 行为描述采用高层结构模型（类似于传统的编程）

- 起源于用来帮助进行设计抽象、以及仿真
  - 现在通过“综合工具”可以自动把HDL描述的电路行为直接转换成门级网表
  - 极大提高了设计者的生产率
  - 这可能会使人误以为硬件设计已经退化到编程

```
1  Decoder(output x0,x1,x2,x3;
2                inputs a,b)
3  {    wire abar, bbar;
4       inv(bbar, b);
5       inv(abar, a);
6       nand(x0, abar, bbar);
7       nand(x1, abar, b   );
8       nand(x2, a,    bbar);
9       nand(x3, a,    b   );
10 }
11  "Behavioral"  example:
12 Decoder(output x0,x1,x2,x3;
13               inputs a,b)
14 {    case [a b]
15           00: [x0 x1 x2 x3] = 0x0;
16           01: [x0 x1 x2 x3] = 0x2;
17           10: [x0 x1 x2 x3] = 0x4;
18           11: [x0 x1 x2 x3] = 0x8;
19      endcase;
20 }
```

设计方法学

设计的结构和
功能（行为）



设计验证：电路行为是否与需要功能一致的?

- 功能、I/O状态
- 寄存器级（结构Architecture）
- 逻辑门级（门）
- 晶体管级（电气特性）
- 时序波形行为

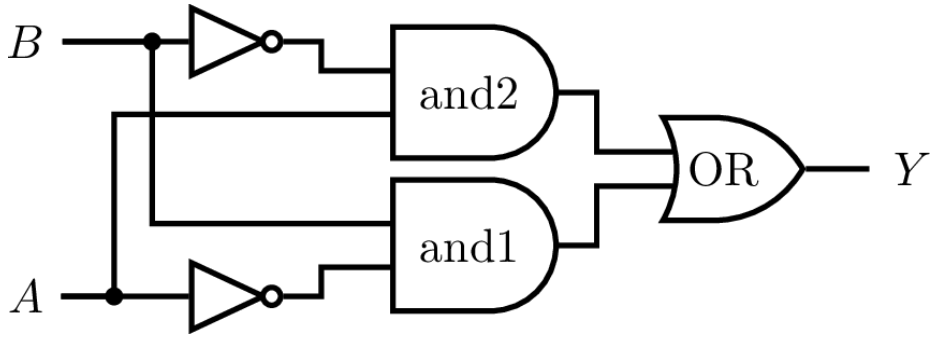
生成门级网表
映射到具体的
电路实现

Verilog 简介

- Module 用于描述电路的一个部件
- 采用两种方法
  - 结构化（Structural Verilog）
    - 列出元件以及他们之间的连接关系
    - 与电路图类似，所不同是采用文本：网表
    - 写起来繁琐，很难读懂
    - 如果没有集成设计工具，这种方法很必要
  - 行为级（Behavioral Verilog）
    - 描述部件做什么，而不是怎么做
    - 综合成具备该行为的电路
    - 生成电路的结果取决于综合工具的质量
- 用多个Module 建立一个层次化的设计

结构模型- XOR

```
1  module xor_gate ( out, a, b );
2  input      a, b;
3  output     out;
4  wire       abar, bbar, t1, t2;
5          inverter invA (abar, a);
6          inverter invB (bbar, b);
7          and_gate and1 (t1, a, bbar);
8          and_gate and2 (t2, b, abar);
9          or_gate  or1 (out, t1, t2);
10 endmodule
```

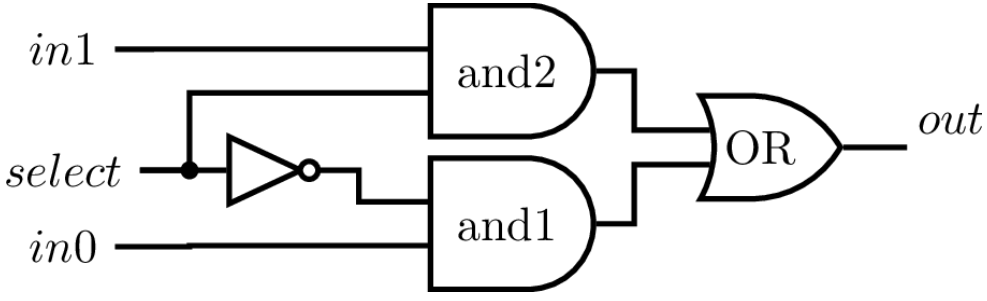


- 采用基本门电路，组成更复杂的电路
- 注意使用声明的线（Wire）
- 默认标识符是线

结构模型: 2-to1 mux

```
1  //2-input multiplexor in gates
2  module mux2 (in0, in1, select, out);
3      input in0,in1,select;
4      output out;
5      wire s0,w0,w1;
6
7      not (s0, select);
8      and (w0, s0, in0),
9          (w1, select, in1);
10     or  (out, w0, w1);
11 endmodule // mux2
```

- 注释
- “module”
- 端口列表
- 端口声明
- 线类型
- 基本门
- 实例名称
- 类型声明



简单的行为级模型

- 组合逻辑
  - 描述输出是输入的逻辑函数
  - 注意使用 assign 关键词，连续赋值

```
1 module and_gate (out, in1, in2);
2 input in1, in2;
3 output out;
4 assign out = in1 & in2;
5 endmodule
```

基本部件的输出位于端口列表的前面

该限制对于一般的Module不适用

When is this "evaluated"?

2-to-1 mux 的行为级描述

```
1  // Behavioral model of 2-to-1
2  // multiplexor.
3  module mux2 (in0,in1,select,out);
4  input in0,in1,select;
5  output out;
6  //
7  reg out;
8
9  always @ (in0 or in1 or select)
10     if (select) out=in1;
11     else out=in0;
12
13 endmodule // mux2
```

- 行为级描述采用关键词*always*，随后是一个赋值过程块
- 赋值的目标输出必须是Reg类型
- 但不是真正的寄存器
- 与 wire 类型不同，wire类型的目的端口的赋值可以连续更新，Reg类型的目的端口赋值只有在赋值执行时才会更新
- 初始化声明可选择

行为级描述 4-to1 mux

```
1  //Does not assume that we have
2  //defined a 2-input mux.
3  //4-input mux behavioral description
4  module mux4 (in0, in1, in2, in3, select, out);
5      input in0,in1,in2,in3;
6      input [1:0] select;
7      output      out;
8      reg         out;
9
10 always @ (in0 in1 in2 in3 select)
11     case (select)
12         2'b00: out=in0;
13         2'b01: out=in1;
14         2'b10: out=in2;
15         2'b11: out=in3;
16     endcase
17 endmodule // mux4
```

- 没有实例化其他Module
- Case 结构与嵌套的 if 结构等价
- 定义：结构描述是指由多个例化的子模块及其连接关系定义的，实现特定逻辑功能的模块
- 行为描述使用高级语言的结构体和操作符实现
- Verilog程序可以包含行为结构体、以及子模块例化的混合描述

结构和行为描述混合的模块

4-bit ripple adder

```
1  module full_addr (S, Cout, A, B, Cin );
2  input    A, B, Cin;
3  output   S, Cout;
4  assign {Cout, S} = A + B + Cin;
5  endmodule
```

行为描述

```
1  module adder4 (S, Cout, A, B, Cin);
2  input    [3:0] A, B;
3  input    Cin;
4  output   [3:0] S;
5  output   Cout;
6  wire     C1, C2, C3;
7  full_addr fa0 (S[0], C1,  A[0], B[0], Cin);
8  full_addr fa1 (S[1], C2,  A[1], B[1], C1);
9  full_addr fa2 (S[2], C3,  A[2], B[2], C2);
10 full_addr fa3 (S[3], Cout,A[3], B[3], C3);
11 endmodule
```

结构描述

注意：加法器的端口顺序

Verilog数据类型和取值

- Bits – wire上的允许取值
  - 0, 1
  - X - 无关/步确定
  - Z - 未驱动, 三态
- Vectors of bits
  - A[3:0] – 4位向量：A[3], A[2], A[1], A[0]
  - 当作无符号整型值处理
    - 例如，A < 0 ??
  - 合并多个bits或vectors成为一个新的向量(vector)
    - e.g., sign extend
    - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
    - B[7:0] = {3{A[3]}, A[3:0]};
  - Style: Use a[7:0] = b[7:0] + c;
    Not: a = b + c; // 需要看一下声明

Verilog 数

- 14　　- 普通十进制数
- -14　　- 2的补码表示
- 12’b0000_0100_0110 - 12 bits 二进制数，忽略“_”
- 12’h046 - 12 bits 16进制数

- Verilog 中的值是无符号数
  - 例如，C[4:0] = A[3:0] + B[3:0];
  - if A = 0110 (6) and B = 1010(-6)
    C = 10000 not 00000
    其中B 在高位补“0”，而不是符号位扩展

Verilog 操作符

>	greater than	Relational
>=	greater than or equal to	Relational
<	less than	Relational
<=	less than or equal to	Relational
==	logical equality	Equality
!=	logical inequality	Equality
===	case equality	Equality
!==	case inequality	Equality
&	bit-wise AND	Bit-wise
^	bit-wise XOR	Bit-wise
^~ or ~^	bit-wise XNOR	Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Verilog Operator	Name	Functional Group
[]	bit-select or part-select	
()	parenthesis	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
|	reduction OR	Reduction
~&	reduction NAND	Reduction
~|	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift

Verilog 变量

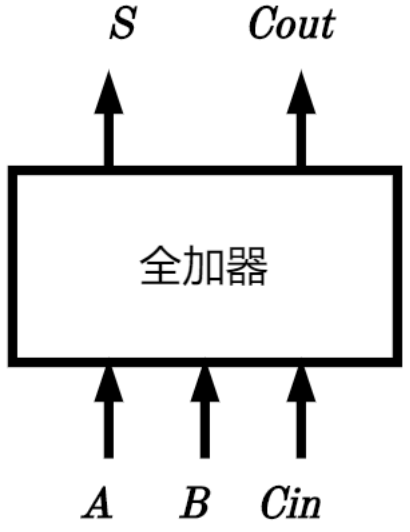
- wire
  - Wire变量用于实现部件端口信号间的连接
- reg
  - Reg变量，在行为描述中用于存储一个值
  - 通常与电路中的一个线对应
  - Reg变量并不意味着在电路中一定需要一个寄存器

- 使用注意
  - 不要把reg赋值与组合逻辑中的连续赋值混淆
  - Reg 一定用在Always模块中（时序逻辑，后续会讲到）

Verilog Module

- 与一个电路的部件对应
  - 参数列表（Parameter list）是该部件对外连接的端口列表
  - 端口声明为三种： “input”，“output” or “inout”
  - “inout”端口用于实现三态总线
  - 端口声明隐含着该端口是一个wire型变量，如果用在Always模块中，需要单独声明称reg类型

```
1  module full_addr (A, B, Cin, S, Cout);
2  input      A, B, Cin;
3  output     S, Cout;
4      assign  {Cout, S} = A + B + Cin;
5  endmodule
```



Verilog 连续赋值

- 赋值计算是连续的
- 赋值与某个连接对应，或者与一个给定的简单功能部件关联
- 赋值目标不是一个reg变量
- 是一种数据流特征

```
1  assign A = X | (Y & ~Z);
2  assign B[3:0] = 4'b01XX;
3  assign C[15:0] = 4'h00ff;
4  assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;
```

使用布尔操作符
(~ 逐位取反, ! 逻辑非)

每一位可以有4种取值
(0, 1, X, Z)

变量可以多位数据宽度
(MSB:LSB)

多个赋值语句可以合并成一个

使用算数操作符

延迟执行计算操作，只用于仿真，不可综合

Comparator 举例(1)

```
1  module Compare1 (A, B, Equal, Alarger, Blarger);
2  input      A, B;
3  output     Equal, Alarger, Blarger;
4      assign Equal = (A & B) | (~A & ~B);
5      assign Alarger = (A & ~B);
6      assign Blarger = (~A & B);
7  endmodule
```

什么时候执行计算?

综合后生成什么?

Comparator 举例(2)

```
1  // Make a 4-bit comparator from 4 1-bit comparators
2  module Compare4(A4, B4, Equal, Alarger, Blarger);
3  input [3:0] A4, B4;
4  output Equal, Alarger, Blarger;
5  wire e0, e1, e2, e3, Al0, Al1, Al2, Al3, Bl0, Bl1, Bl2, Bl3;
6      Compare1 cp0(A4[0], B4[0], e0, Al0, Bl0);
7      Compare1 cp1(A4[1], B4[1], e1, Al1, Bl1);
8      Compare1 cp2(A4[2], B4[2], e2, Al2, Bl2);
9      Compare1 cp3(A4[3], B4[3], e3, Al3, Bl3);
10 assign Equal = (e0 & e1 & e2 & e3);
11 assign Alarger = (Al3 | (Al2 & e3) |
12                    (Al1 & e3 & e2) |
13                    (Al0 & e3 & e2 & e1));
14 assign Blarger = (~Alarger & ~Equal);
15 endmodule
```

是否有更好的方法?

行为级描述如何写?

简单行为模型- always 块

- *always* 块
  - 总是等待触发信号的变化
  - 然后执行块内的赋值操作

```
1  module and_gate (out, in1, in2);
2  input    in1, in2;
3  output   out;
4  reg      out;
5  always @(in1 or in2)
6      begin
7       out = in1 & in2;
8      end
9  endmodule
```

不是一个真正的寄存器
因为赋值目标在“Always”块内，
所有需要一个register型变量

指明块内的语句什么时候执行，
哪些信号会触发执行过程

always 块

- 用于描述电路功能的一段程序
  - 可以包含很多语句，如if, for, while, case等
  - 在“Always”块中的语句顺序执行
    - 阻塞赋值
    - 连续赋值“<=”会并行执行，非阻塞
  - 整个”Always”块会在同一时刻立即执行
    - 但其执行含义会被顺序解释
      - 仿真时的微时刻，和宏时刻
      - 综合
    - 电路功能的最终输出取决于当前的一组输入
    - 中间的赋值结果不重要，只关心最终结果
- begin/end 是块的起始和结束标志语句

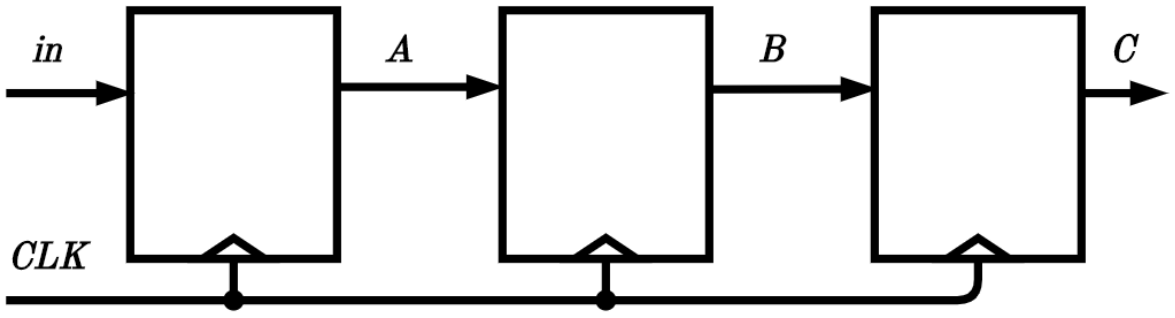
Verilog 生成什么存储元件

- 语句的表达方式生成组合逻辑
  –完成输入到输出间的映射关系

- 存储元件携带相同的值及时向前传输

状态举例——阻塞赋值

```
1  module shifter (in,clk,A,B,C);
2  input in, clk;
3  output A,B,C;
4  reg A, B, C;
5  always @(posedge clk)
6      begin
7          C = B;
8          B = A;
9          A = in;
10     end
11 endmodule
```

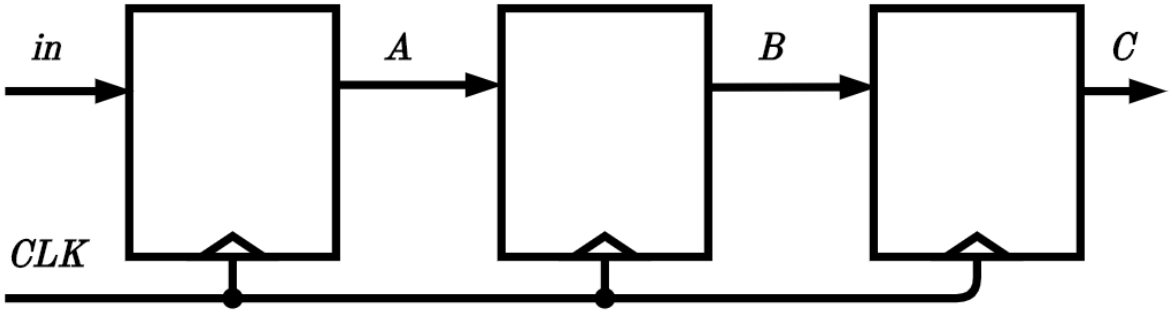


阻塞赋值语句顺序解释

但执行动作同时发生

状态举例– 非阻塞赋值

```
1  module shifter (in,clk,A,B,C);
2  input in, clk;
3  output A,B,C;
4  reg A, B, C;
5  always @(posedge clk)
6      begin
7          C <= B;
8          B <= A;
9          A <= in;
10     end
11 endmodule
```



- 非阻塞: 所有语句被并行解释
  - 所有赋值语句右侧的计算先完成
  - 然后所有赋值语句同时

“完全” 赋值

- 如果一个“Always”块执行了，某个变量未被赋值
  - 变量将保留原来的取值（设想隐含的状态）
  - 非组合逻辑，会插入锁存器（隐含着存储器）
  - 这通常不是所期望的，尤其对新手来说

- 每个在“Always”块内被赋值的变量都应该包含了该变量可能发生改变的所有情况

“不完全” 触发

- 遗漏输入触发条件会导致生成时序逻辑
- 如，以下代码中的“与”门输出依赖于输入的历史取值

```
1  module and_gate (out, in1, in2);
2  input          in1, in2;
3  output         out;
4  reg            out;
5  always @(in1)
6      begin
7          out = in1 & in2;
8      end
9  endmodule
```

这段代码会生成什么样的硬件电路?

多位向量行为模型

```
1  //Behavioral model of 32-bitwide 2-to-1 multiplexor.
2  module mux32 (in0,in1,select,out);
3  input [31:0] in0,in1;
4  input    select;
5  output [31:0] out;
6  reg [31:0] out;
7  always @ (in0 or in1 or select)
8      if (select) out=in1;
9      else out=in0;
10 endmodule // Mux
11
12 //Behavioral model of 32-bit adder.
13 module add32 (S,A,B);
14 input [31:0] A,B;
15 output [31:0] S;
16 reg [31:0] S;
17 always @ (A or B)
18     S = A + B;
19 endmodule // Add
```

输入、输出都是32位向量

层级描述与位向量

```
1  //Assuming we have already
2  // defined a 2-input mux (either
3  // structurally or behaviorally,
4  //4-input mux built from 3 2-input muxes
5  module mux4 (in0, in1, in2, in3, select, out);
6  input in0,in1,in2,in3;
7  input [1:0] select;
8  output      out;
9  wire        w0,w1;
10 mux2
11     m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
12     m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
13     m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
14 endmodule // mux4
```

例化方法类似于元件引用

Select信号两位宽

采用命名的端口赋值方法

Verilog `if`

```
1  // Simple 4-1 mux
2  module mux4 (sel, A, B, C, D, Y);
3  input [1:0] sel;      // 2-bit control signal
4  input A, B, C, D;
5  output Y;
6  reg Y;                // target of assignment
7    always @(sel or A or B or C or D)
8      if (sel == 2'b00) Y = A;
9      else if (sel == 2'b01) Y = B;
10     else if (sel == 2'b10) Y = C;
11     else if (sel == 2'b11) Y = D;
12 endmodule
```

- 与C语言的 if语句语法相同
- 顺序语义，同时执行

Verilog `if`

```
1  // Simple 4-1 mux
2  module mux4 (sel, A, B, C, D, Y);
3  input [1:0] sel;     // 2-bit control signal
4  input A, B, C, D;
5  output Y;
6  reg Y;               // target of assignment
7    always @(sel or A or B or C or D)
8      if (sel[0] == 0)
9        if (sel[1] == 0) Y = A;
10       else             Y = B;
11     else
12       if (sel[1] == 0) Y = C;
13       else             Y = D;
14 endmodule
```

Verilog case

```
1  // Simple 4-1 mux
2  module mux4 (sel, A, B, C, D, Y);
3  input [1:0] sel;      // 2-bit control signal
4  input A, B, C, D;
5  output Y;
6  reg Y;                // target of assignment
7    always @(sel or A or B or C or D)
8      case (sel)
9        2'b00: Y = A;
10       2'b01: Y = B;
11       2'b10: Y = C;
12       2'b11: Y = D;
13     endcase
14 endmodule
```

- 顺序执行的 cases
  - 只有第一个匹配的Case条件才会被执行
  - 可以用默认的Case
- Case条件自顶向下逐一检查

Verilog case

```
1  // Simple binary encoder (input is 1-hot)
2  module encode (A, Y);
3  input  [7:0] A;      // 8-bit input vector
4  output [2:0] Y;      // 3-bit encoded output
5  reg    [2:0] Y;      // target of assignment
6    always @(A)
7      case (A)
8        8'b00000001: Y = 0;
9        8'b00000010: Y = 1;
10       8'b00000100: Y = 2;
11       8'b00001000: Y = 3;
12       8'b00010000: Y = 4;
13       8'b00100000: Y = 5;
14       8'b01000000: Y = 6;
15       8'b10000000: Y = 7;
16       default:     Y = 3'bX;   // Don't care when input is not 1-hot
17     endcase
18 endmodule
```

- 如果没有默认case选项，本例将为Y逻辑变量生成锁存器
- 牢记，我们是在设计硬件，不是在写程序！
- 为变量赋值“X”,意味着综合器可以为该变量分配任何值

Verilog case (续)

```
1  // Priority encoder
2  module encode (A, Y);
3  input  [7:0] A;      // 8-bit input vector
4  output [2:0] Y;      // 3-bit encoded output
5  reg    [2:0] Y;      // target of assignment
6    always @(A)
7      case (1'b1)
8        A[0]:    Y = 0;
9        A[1]:    Y = 1;
10       A[2]:    Y = 2;
11       A[3]:    Y = 3;
12       A[4]:    Y = 4;
13       A[5]:    Y = 5;
14       A[6]:    Y = 6;
15       A[7]:    Y = 7;
16       default: Y = 3'bX; // Don't care when input is all 0's
17     endcase
18 endmodule
```

- Cases语句被顺序执行
- 例中的Case会实现一个优先级编码器

并行 Case

```
1  // simple encoder
2  module encode (A, Y);
3  input  [7:0] A;      // 8-bit input vector
4  output [2:0] Y;      // 3-bit encoded output
5  reg    [2:0] Y;      // target of assignment
6    always @(A)
7      case (1'b1)          // synthesis parallel-case
8        A[0]:    Y = 0;
9        A[1]:    Y = 1;
10       A[2]:    Y = 2;
11       A[3]:    Y = 3;
12       A[4]:    Y = 4;
13       A[5]:    Y = 5;
14       A[6]:    Y = 6;
15       A[7]:    Y = 7;
16       default: Y = 3'bX;// Don't care when input is all 0's
17     endcase
18 endmodule
```

- 优先级编码器要比简单编码器占用更多逻辑资源
- 如果已知编码器输入是单bit有效，可以告诉综合器
- 并行case写法表示Case条件的顺序无关，可以用普通编码器实现该设计

Verilog casex

- 与case类似，但Casex条件中允许包含“X”取值
- 当计算Case条件时，取值为“X”的条件不考虑
- 换句话说，可以不考虑那些位的取值

casex 举例

```
1  // Priority encoder
2  module encode (A, valid, Y);
3  input  [7:0] A;      // 8-bit input vector
4  output [2:0] Y;      // 3-bit encoded output
5  output valid;        // Asserted when an input is not all 0's
6  reg    [2:0] Y;      // target of assignment
7  reg    valid;
8    always @(A) begin
9      valid = 1;
10     casex (A)
11       8'bXXXXXXX1: Y = 0;
12       8'bXXXXXX10: Y = 1;
13       8'bXXXXX100: Y = 2;
14       8'bXXXX1000: Y = 3;
15       8'bXXX10000: Y = 4;
16       8'bXX100000: Y = 5;
17       8'bX1000000: Y = 6;
18       8'b10000000: Y = 7;
19       default:  begin
20          valid = 0;
21          Y = 3'bX; // Don't care when input is all 0's
22       end
23     endcase
24   end
25 endmodule
```

Verilog for

```
1  // simple encoder
2  module encode (A, Y);
3  input  [7:0] A;      // 8-bit input vector
4  output [2:0] Y;      // 3-bit encoded output
5  reg    [2:0] Y;      // target of assignment
6  integer i;       // Temporary variables for program only
7  reg [7:0] test;
8    always @(A) begin
9      test = 8'b00000001;
10     Y = 3'bX;
11     for (i = 0; i < 8; i = i + 1) begin
12         if (A == test) Y = N;
13         test = test << 1;
14     end
15   end
16 endmodule
```

- 与C中的for类似
- for 语句在编译的时候执行（类似宏扩展）
  - 只关注计算结果
  - 不关注如何计算
  - 只用在测试用例中，不可综合

另一种行为模型例子

```
1  module life (neighbors, self, out);
2  input          self;
3  input [7:0]    neighbors;
4  output         out;
5  reg            out;
6  integer        count;
7  integer        i;
8  always @(neighbors or self) begin
9      count = 0;
10     for (i = 0; i<8; i = i+1) count = count + neighbors[i];
11     out = 0;
12     out = out | (count == 3);
13     out = out | ((self == 1) & (count == 2));
14     end
15 endmodule
```

- 计算conway游戏的生命规则
  - 一个没有邻居、或有4个邻居的细胞会死亡
  - 有2个、或3个邻居的细胞会活命

整数是编译时的临时变量

Always块会立即执行，如果没有延迟，会使用最后的结果

Verilog `while/repeat/forever`

- `while` (expression) statement
  - Execute statement while expression is true

- `repeat` (expression) statement
  - Execute statement a fixed number of times

- `forever` statement
  - Execute statement forever

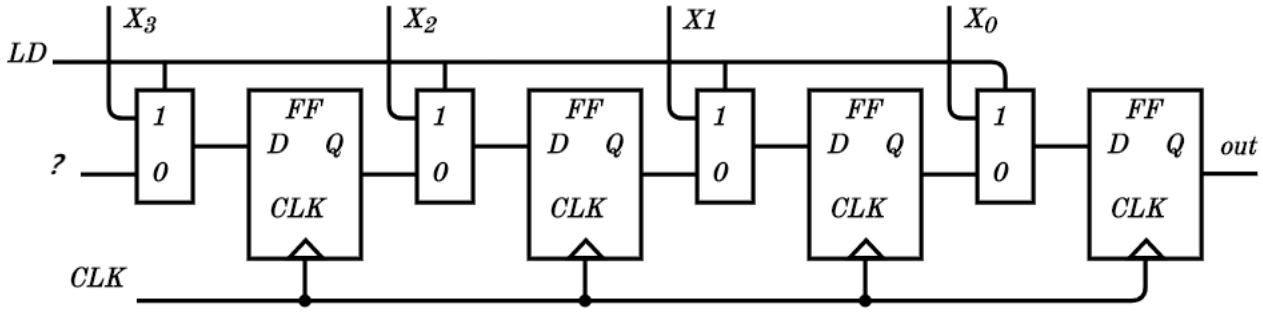
完全case 与并行 case

- // synthesis parallel_case
  - 告诉编译器Case中的条件顺序无关紧要
  - 也就是Case中的条件选项不会相互覆盖
  - 如在状态机中，不可能同时处于多个状态，某一时刻只能处于某一个状态
  - 可给出一种最简单的实现方式

- // synthesis full_case
  - 通知编译器漏掉了某些无关紧要的Case条件
  - 从而避免不完全声明而导致的锁存器

时序逻辑

```
1  //Parallel to Serial converter
2  module ParToSer(LD, X, out, CLK);
3  input [3:0] X;
4  input LD, CLK;
5  output out;
6  reg out;
7  reg [3:0] Q;
8  assign out = Q[0];
9  always @ (posedge CLK)
10     if (LD) Q=X;
11     else Q = Q>>1;
12 endmodule // mux2
13
14 module FF (CLK,Q,D);
15 input D, CLK;
16 output Q; reg Q;
17     always @ (posedge CLK) Q=D;
18 endmodule // FF
```



- “always @ (posedge CLK)” 强制使 Q 寄存器的输出在每个仿真周期被更新
- “>>” 操作符不是算数右移，算数右移时左侧补“0”
- 在非寄存器变量右移时可以用合并方法实现，例如：
  - wire [3:0] A, B;
  - assign B = {1’b0, A[3:1]}

测试用例

Top-level modules written specifically to test sub-modules. Generally no ports.

```
1  module testmux;
2  reg a, b, s;
3  wire f;
4  reg expected;
5  mux2 myMux (.select(s), .in0(a), .in1(b), .out(f));
6      initial
7          begin
8              s=0; a=0; b=1; expected=0;
9            #10 a=1; b=0; expected=1;
10           #10 s=1; a=0; b=1; expected=1;
11         end
12     initial
13         $monitor(
14             "select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
15              s, a, b, f, expected, $time);
16     endmodule // testmux
```

- initial模块与always模块很像，只是在仿真开始时仅执行一次
- #n 指延迟n个单位时间（仿真周期）
- $monitor 打印输出
- 还有很多其他系统功能用来实现输出显示，以及仿真控制

总结

- Verilog看起来和C很像，但它是用来描述硬件的
  - 多种物理元素，并行的动作
  - 严格的时间关系
  - 仿真与可综合的基本功能
  - 先设计出你所期望的电路，再用Verilog表达出来
- 理解语言的元素
  - Modules, ports, wires, reg, primitive, continuous assignment, blocking statements, sensitivity lists, hierarchy
  - 通过实验来学习，能更好地理解
- 行为模型描述方法可以隐藏具体的电路结构，但设计者仍需掌控电路结构、数据通信、并行执行、以及时序设计等关键要素

问题和建议?