EDA 技术(B)

基于 Verilog HDL

学科考试急救包

使用说明

急救包的设计以解题方案为主，不以知识掌握为目的；
渠道资料均为人工整合，部分存在差错请及时向生产方提出；
建议配合平时作业或习题使用，急救效果更佳；
急救包为了便于记忆，产生部分非专业用语和操作方法，请学霸绕道。

生产方：歪门邪道研究所　　版本号：1.8 D230214T1927

Direction 1 EDA 技术概念

[1] IP 核的概念和分类

利用 IP 核设计电子系统，引用方便，修改基本元件的功能容易。具有复杂功能和商业价值,IP 核一般**具有知识产权**，尽管 IP 核的市场活动还不规范，但是仍有许多集成电路设让公司从事 IP 核的设计、开发和营销工作。

1) 软 IP（是用 VHDL/Verilog HDL 等硬件描述语言描述的功能块）
2) 固 IP（是完成了综合的功能块）
3) 硬 IP（提供设计的最终阶段产品：掩模）

[2] EDA 技术相关概念

1) 用 EDA 技术进行电子系统设计的目标是最终完成 **ASIC** 的设计与实现
2) EDA 缩写的含义为**电子设计自动化**
3) 基于查找表技术构造的可编程逻辑器件叫做 **FPGA**，基于乘积项
4) 技术构造的可编程逻辑器件叫做 **CPLD**
5) 可编程器件分为 **CPLD** 和 **FPGA**
6) 有限状态机分为 **Moore** 和 **Mealy** 两种类型
7) Verilog 的端口模式有 **input**、**output** 和 **inout** 三种
8) 随着 EDA 技术的不断完善与成熟，**自顶向下**的设计方法更多的被应用于 VerilogHDL 设计当中
9) 完整的条件语句将产生**组合电路**，不完整的条件语句将产生**时序电路**

[3] 同步复位与异步复位

1) 同步复位只有在时钟沿到来时复位信号才起作用
2) 异步复位只要有复位信号系统马上复位。
3) 同步复位的复位信号持续的时间应该超过一个时钟周期才能保证系统复位异步复位抗干扰能力差，有些噪声也能使系统复位，因此有时候显得不够稳定，要想设计一个好的复位最好使用异步复位同步释放，而且复位信号低电平有效

[4] 有限状态机 FSM

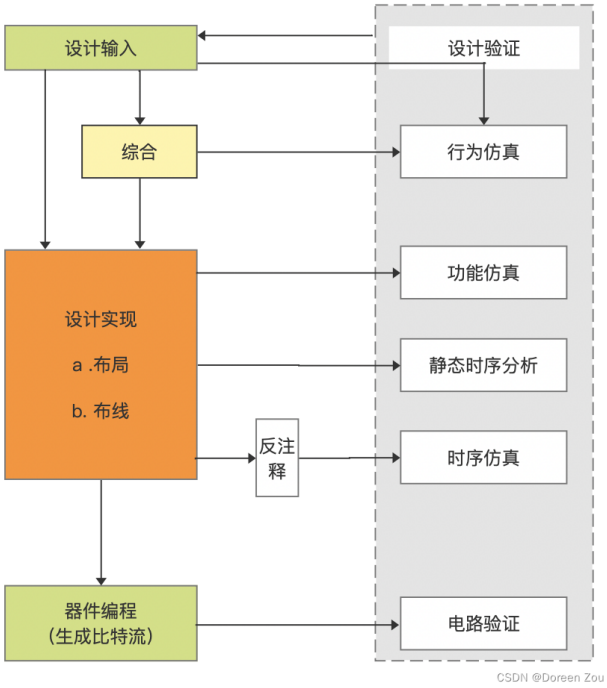
1) 根据内部结构不同可分为摩尔型状态机和米里型状态机两种
2) 摩尔型状态机的输出只由当前状态决定，而次态由输入和现态共同决定
   同步输出状态机，它的输出仅为当前状态的函数，这类状态机在输入发生变化时必须等待时钟的到来，比 Mealy 机要多等待一个时钟周期。(输出只由当前状态决定，而次态由输入和现态共同决定)
3) 米里型状态机的输出由输入和现态共同决定，而次态也由输入和现态决定。
   异步输出状态机，它的输出是当前状态和所有输入信号的函数，它的输出是在输入变化后立即发生的，不依赖时钟的同步。(输出由输入和现态共同决定，而次态也由输入和现态决定)
4) 状态编码主要有三种：连续二进制编码、格雷码和独热码

[5] 数字系统设计流程

1) 设计输入：将设计的结构和功能通过原理图或硬件描述语言进行设计或编程，进行

语法或逻辑检查，通过表示输入完成，否则反复检查直到无任何错误

2) 逻辑综合：将较高层的设计描述自动转化为较低层次描述的过程，包括行为综合，逻辑综合和版图综合或结构综合，最后生成电路逻辑网表的过程
3) 布局布线：将综合生成的电路网表映射到具体的目标器件中，并产生最终可下载文件的过程
4) 仿真：就是按照逻辑功能的算法和仿真库对设计进行模拟，以验证设计并排除错误的过程，包括功能仿真和时序仿真
5) 编程配置：将适配后生成的编程文件装入到 PLD 器件的过程，根据不同器件实现编程或配置



[6] Verilog HDL 电路设计法

1) 自上而下的设计方法(Top-Down) ⭐
即先定义顶层模块功能，进而分析要构成顶层模块的必要子模块；然后进一步对各个模块进行分解、设计，直到到达无法进一步分解的底层功能块。这样，可以把一个较大的系统，细化成多个小系统，从时间、工作量上分配给更多的人员去设计，从而提高了设计速度，缩短了开发周期
2) 自下而上的设计方法(Bottom-Up)
3) 综合设计的方法

[7] 阻塞赋值与非阻塞赋值

1) 非阻塞(non-blocking)赋值方式（b<= a):
b 的值被赋成新值 a 的操作，并不是立刻完成的，而是在块结束时才完成；块内的多条赋值语句在块结束时同时赋值；硬件有对应的电路。
2) 阻塞(blocking)赋值方式（b = a):
b 的值立刻被赋成新值 a；完成该赋值语句后才能执行下一句的操作；硬件没有对应的电路，因而综合结果未知。阻塞赋值是在该语句结束是立即完成赋值操作；非阻塞赋值是在整个过程块结束是才完成赋值操作。

[8] 文件后缀

.v 就是 Verilog 语言编写的程序代码文件，就像 c 语言编程的.c 文件一样

.vwf 仿真文件

.bdf 和.vhdl 都是硬件描述文件，通常将.bdf 文件设置为顶层文件，.vhdl 文件实例化为元件

Direction 2 Verilog 基础语法

[1] 语法文本格式

1) Verilog 是区分大小写的
2) 格式自由，可以在一行内编写，也可跨多行编写
3) 每个语句必须以分号为结束符。空白符（换行、制表、空格）都没有实际的意义，在编译阶段可忽略

[2] 注释

1) 用 // 进行行注释
2) 用 /* 与 */ 进行块注释

[3] 模块结构

1) Verilog 的基本设计单元是“模块”（block）
2) 一个模块由两部分组成，一部分描述接口，另一部分描述逻辑功能
3) 每个 Verilog 程序包括 4 个主要的部分：
   a) 端口定义
   b) IO 说明
   c) 内部信号定义
   d) 功能定义

[4] 标识符与关键字

1) 标识符可以是任意一组字母、数字、$ 符号和 _(下划线)符号的合，但标识符的第一个字符必须是字母或者下划线，不能以数字或者美元符开始。
2) 标识符是区分大小写的
3) 关键字是 Verilog 中预留的用于定义语言结构的特殊标识符
4) Verilog 中关键字全部为小写。

	实例：标识符与关键字
01	**reg** [3:0] counter ; //reg 为关键字， counter 为标识符
02	**input** clk; //input 为关键字，clk 为标识符
03	**input** CLK; //CLK 与 clk 是 2 个不同的标识符

[5] 电平逻辑值

1) 逻辑 0：表示低电平，表示 GND（接地）
2) 逻辑 1：表示高电平，表示 VCC（接电源）
3) 逻辑 X：表示未知，有可能是高电平，也有可能是低电平
   意味着信号数值的不确定，即在实际电路里，信号可能为 1，也可能为 0

4) 逻辑 Z：表示高阻态，没有激励信号，悬空状态

意味着信号处于高阻状态，常见于信号(input, reg)没有驱动时的逻辑结果。例如一个 pad 的 input 呈现高阻状态时，其逻辑值和上下拉的状态有关系。上拉则逻辑值为 1，下拉则为 0

[6] 数值类型

数字声明时，合法的基数格式有 4 中，包括：十进制('d 或 'D)，十六进制('h 或 'H)，二进制（'b 或 'B)，八进制（'o 或 'O)。数值可指明位宽，也可不指明位宽

	实例：指明位宽的数值类型
01	4'b1011 // 4bit 数值
02	32'h3022_c0de // 32bit 的数值

其中，下划线 _ 是为了增强代码的可读性

一般直接写数字时，默认为十进制表示，例如下面的 3 种写法是等效的：

	实例：不指明位宽的数值类型
01	counter = 'd100 ; //一般会根据编译器自动分频位宽，常见的为 32bit
02	counter = 100 ;
03	counter = 32'h64 ;

[7] 字符串

字符串是由双引号包起来的字符队列。字符串不能多行书写，即字符串中不能包含回车符。Verilog 将字符串当做一系列的单字节 ASCII 字符队列

	实例：字符串
01	**reg** [0: 14*8-1] str ;
02	**initial begin**
03	str = "I Love You";
04	**end**

[8] 数据类型

1) 线网型(wire)
   - 表示结构实体之间的物理连线。
   - 门、连续赋值语句、assign 可以驱动线网型(wire)
   - 如果没有驱动原件连接，wire 连接的变量为高阻，值为 Z
2) 寄存器类型(reg)
   - reg 类型的默认初始值为 x
   - 用来表示存储单元，它会保持数据原有的值，直到被改写
   - 只能在 always 和 initial 语句中被赋值

     时序逻辑：always 语句带有时钟信号，寄存器变量对应为触发器

     组合逻辑：always 语句不带有时钟信号，寄存器变量对应为硬件连接
3) 参数类型
   - 本质是一个常量，用关键字 parameter 定义常量
   - 可以一次定义多个参数，参数间用逗号隔开
   - 参数定义的右边必须是一个常数表达式
   - 定义格式：parameter A = 4'd1;

parameter ： 全局参数定义，可在整个设计中传递参数
localparam：仅限于当前模块的参数定义，跨模块不可用

[9] 赋值

1) 阻塞赋值(=)

阻塞赋值的操作也称为阻塞型过程赋值。阻塞型过程赋值语句的特点如下:

① 在 begin-end 串行语句块中的各条阻塞型过程赋值语句将以它们在顺序块后排列次序依次得到执行

② 阻塞型过程赋值语句的执行过程是：首先计算右端赋值表达式的值，然后立即将计算结果赋值给“=”左端的被赋值变量

③ 在串行语句块中，下一条语句的执行会被本条阻塞型过程赋值语句所阻塞，只有在当前这条阻塞型过程赋值语句所对应的赋值操作执行完后下一条语句才能开始执行

2) 非阻塞赋值(<=)

非阻塞赋值的操作过程可以看做两个步骤:

1. 赋值开始时，计算 RHS（右手语句）
2. 赋值结束时，更新 LHS（左手语句）

① 非阻塞赋值的概念是指：在计算非阻塞赋值的 RHS 以及更新 LHS 期间，允许其他的非阻塞赋值语句同时计算 RHS 和更新 LHS。

② 非阻塞赋值只能用于对寄存器类型的变量进行赋值，因此只能用在 initial 块和 always 块等过程块中。

③ 时序逻辑的 always 块用非阻塞赋值(<=)(时序逻辑电路结构往往与触发沿有关系，只有在触发沿时才可能发生赋值变化)

[10] 运算符

关系运算、算术运算、逻辑运算、位逻辑运算、位移运算、三元布尔运算式均与绝大部分编程语言语法相同，只要学过 C、C#、Java、Object-C 等语言及二进制原理都能理解，属于计算机程序设计领域，过于简单，不再阐述，详询 CSDN

[11] 条件语句

条件语句必须在过程块中（initial 和 always 语句引导）中使用

Verilog HDL 语言中存在两种分支语言:

① if-else 条件分支语句

② case 分支控制语句

具体使用均与绝大部分编程语言语法相同，属于计算机程序设计领域，过于简单，不再阐述，详询 CSDN

[12] always 语句

1) always 语句一直不断重复活动
2) 但是只有和一定的时间控制结合在一起才有作用
3) always 的时间控制可以是边沿触发，也可以是电平触发
4) 多个信号用 or 连接

5) always 后连接的多个事件名或信号名组成的列表称为“敏感列表”

边沿触发的 always 块常用于描述时序逻辑

电平触发的 always 块常用于描述组合逻辑

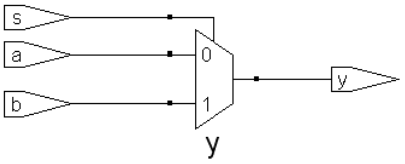
[13] assign

assign 是 Verilog 的关键词，称为连续赋值，assign 后面只能接 wire 型，不能接 reg 型

Direction 3 程序设计详解

[1] 二选一数据选择器

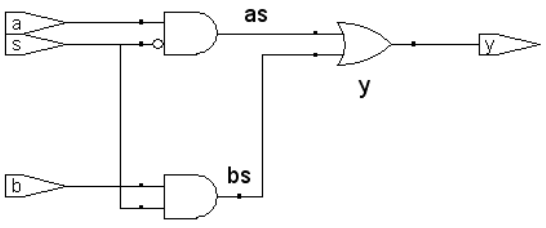
一个二选一数据选择器如图所示，s 是数据选择控制端，a，b 是输入信号，y 是输出信号



	实例：二选一数据选择器
01	module mux2_1(a, b, s, y); //模块名、模块接口名
02	input a, b, s; // 定义输入端口
03	output y; // 定义输出端口
04	/* s 为 0 时，选择 a 输出；
05	s 为 1 时，选择 b 输出。*/
06	assign y = (s == 0) ? a : b; //输出信号
07	endmodule

[2] 对于上面的简单数据选择器还可以表达为逻辑门结构，二选一数据选择器

	实例：简单数据选择器的另一种 always 表达代码
01	module mux2_1(a, b, s, y);
02	input a, b, s;
03	output y;
04	reg y; //reg 表示寄存器
05	
06	always @(a, b, s)
07	begin
08	if(!s) y = a;
09	else y = b;
10	end
11	endmodule

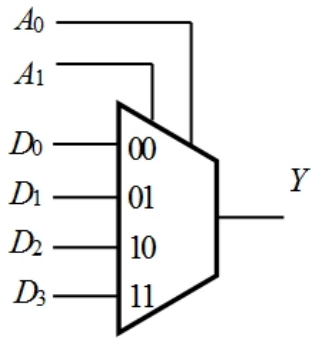


[3] 四选一数据选择器

D0、D1、D2、D3 是四个数据输入端，Y 为输出端，A1、A0 是地址输入端。从表中可见，利用指定 A1A0 的代码，能够从 D0、D1、D2、D3 这四个输入数据中选出任何一个并送到输出端。因此，用数据选择器可以实现数据的多路分时传送

	实例：四选一数据选择器 case 代码
01	module mux4_1 (d0,d1,d2,d3,a0,a1,y); //定义模块和端口变量

```
02     input d0,d1,d2,d3,a0,a1; //定义输入端
03     output y; //定义输出端
04     reg y; //定义寄存器存储结果
05  always@( d0,d1,d2,d3,a0,a1) //载入发生信号变化的端口
06     begin
07       case({a0,a1}) //并位
08         2'b00 : y<=d0;
09         2'b01 : y<=d1;
10         2'b10 : y<=d2;
11         2'b11 : y<=d3;
12           default : y<=d0; //可以不写，默认位
13         endcase
14     end
15  endmodule
```



	实例：四选一数据选择器 assign 代码
01	module mux4_1 (d0,d1,d2,d3,a0,a1,y); //定义模块和端口变量
02	input d0,d1,d2,d3,a0,a1; //定义输入端
03	output y; //定义输出端
04	reg y; //定义寄存器存储结果
05	wire [1:0] SEL; //定义网线型变量用于并位 s0,s1
06	wire d0T, d1T, d2T, d3T; //定义中间变量，用于接收 SEL 触发 d 系列口的变化
07	assign SEL={s1,s0}; //并位操作(wire 型变量从右到左记)
08	assign d0T = (SEL==2'D0); //当 SEL 十进制值为 0 时它为 1，否则为 0，下列一致
09	assign d1T = (SEL==2'D1);
10	assign d2T = (SEL==2'D2);
11	assign d3T = (SEL==2'D3);
12	assign y = (d0&d0T) | (d1&d1T) | (d2&d2T) | (d3&d3T) //d 系列端口都有信号，所以
13	与 dT 系列端口相与，四个逻辑信号再相或，实现选择
14	endmodule

	实例：四选一数据选择器 if 代码
01	module mux4_1 (d0,d1,d2,d3,a0,a1,y); //定义模块和端口变量
02	input d0,d1,d2,d3,a0,a1; //定义输入端
03	output y; //定义输出端
04	reg y; //定义寄存器存储结果
05	always@(d0,d1,d2,d3,a0,a1) // 也可以用并位简化，就不需要 if 嵌套，与 C 语言相同
06	if(!s0)
07	begin
08	if(!s1)
09	y <= d0;
10	else
11	y <= d1;
12	end

```
13    else begin
14      if(!s1)
15      y <= d3;
16      else
17      y <= d4;
18    end
19  endmodule
```

[4] 八选一数据选择器

	实例：八选一数据选择器 case 代码，与后面的 3-8 译码器类似，可类比
01	module Eignth_Select(out,in0,in1,in2,in3,in4,in5,in6,in7,sel);
02	output out;
03	input in0,in1,in2,in3,in4,in5,in6,in7;
04	input[2:0] sel;
05	reg out; //输出信号，可观察输出信号波形判断仿真是否正确
06	always @(in0,in1,in2, in3 ,in4 , in5 , in6 , in7, sel)
07	case(sel) //根据 sel 的不同选通 in0,in1,in2,in3,in4,in5,in6,in7
08	3'b000: out=in0;
09	3'b001: out=in1;
10	3'b010: out=in2;
11	3'b011: out=in3;
12	3'b100: out=in4;
13	3'b101: out=in5;
14	3'b110: out=in6;
15	3'b111: out=in7;
16	default: out=1'bx;
17	endcase
18	endmodule

[5] 半加器(目的是学会看仿真波形图写程序)

半加器电路是指对两个输入数据位相加，输出一个结果位和进位，没有进位输入的加法器电路。 是实现两个一位二进制数的加法运算电路

$$S0 = A \oplus B\ , C0 = A \cdot B$$

	实例：半加器
01	module half_adder(a,b,S0,C0);
02	//注意这个模块名，后面例化要用
03	input a,b;
04	output S0,C0;
05	assign S0=A^B;
06	assign C0=A&B;
	endmodule

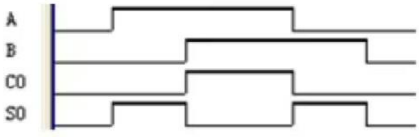


图 3-3 半加器的仿真功能波形图

[6] 全加器(半加器例化)

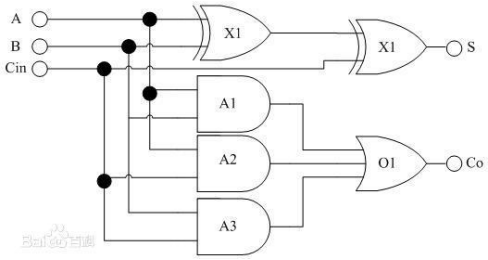
例化可以简单理解为调子模块。如果你用 HDL 语言写了一个模块，在另一个模块中要使用

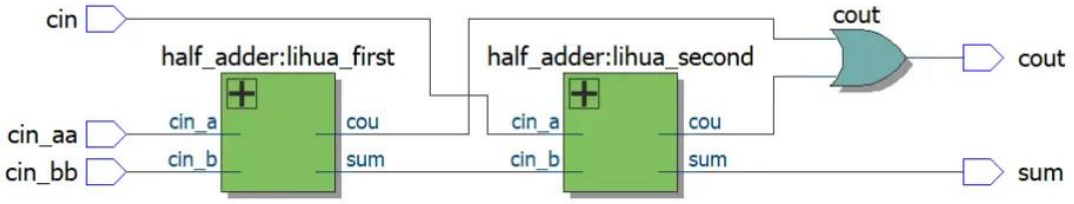
这个模块，那么之前那个模块就要例化到新的模块中
全加器是能够计算低位进位的二进制加法电路。与半加器相比,全加器不只考虑本位计算结果是否有进位,也考虑上一位对本位的进位,可以把多个一位全加器级联后做成多位全加器
A,B 为要相加的数，Cin 为进位输入；S 为和，Co 是进位输出；

$$S = A \oplus B \oplus Cin$$

$$Co = (A\&B)|(B\&Cin)|(A\&Cin)$$

	实例：全加器(非例化)
01	module full_adder(
	//输入信号，ain 表示被加数，bin 表示加数，cin 表示低位向高位的进位
	input ain,bin,cin,
	//输出信号,cout 表示向高位的进位，sum 表示本位的相加和
	output reg cout,sum);
02	reg s1,s2,s3;
03	always @(ain,bin,cin)
04	begin
05	sum=(ain^bin)^cin;//本位和输出表达式
06	s1=ain&cin;
07	s2=bin&cin;
08	s3=ain&bin;
09	cout=(s1|s2)|s3;//高位进位输出表达式
10	end
11	endmodule





	实例：全加器(通过两个半加器和一个或门例化)
01	module full_adder(ainF,binF,cin,cout,sum);
02	input ainF,binF,cin;
03	output cout,sum;
04	reg cout,sum;
05	
06	//定义网线型变量
07	wire first_cout, second_cout; //两个半加器的进位输出
08	wire first_sum; //第一个半加器的求和输出
09	
10	//调用半加器例化模块，端口配对
11	half_adder first_adder(
12	.ain(ainF),

```
13      .bin(binF),
14      .C0(first_cout),
15      .S0(first_sum) );
16
17  half_adder second_adder(
18      .ain(cin),
19      .bin(first_sum),
20      .C0(second_cout),
21      .S0(sum) );
21
23  assign cout = (first_cout | second_cout);
24
25  endmodule
```

[7] 8 位二进制加法器(通过全加器例化)

实现八位的两数相加，获得八位结果以及进位。

实例：8 位二进制加法器(通过非例化的全加器例化)

```
01  module adder_8bit(a,b,cin,sum,cout);
02      input[7:0] a,b; //定义 8 位的输入数
03      input cin;
04      output[7:0] sum;
05      output cout;
06      wire c1,c2,c3,c4,c5,c6,c7; //每一位依次经过了全加器的进位输出
07
08      full_adder u1(a[0],b[0],cin, c1,sum[0]); //第一位的 cin 为当前的进位输入
09      full_adder u2(a[1],b[1],c1,c2,sum[1]); //前一位的进位输出作为后一位的进位输入
10      full_adder u3(a[2],b[2],c2,c3,sum[2]);
11      full_adder u4(a[3],b[3],c3,c4,sum[3]);
12      full_adder u5(a[4],b[4],c4,c5,sum[4]);
13      full_adder u6(a[5],b[5],c5,c6,sum[5]);
14      full_adder u7(a[6],b[6],c6,c7,sum[6]);
15      full_adder u8(a[7],b[7],c7,cout,sum[7]); //最终的进位输出为当前进位输入
16
17  endmodule
```

[8] 4 位二进制加法器

实例：4 位二进制加法器(通过非例化的全加器例化)

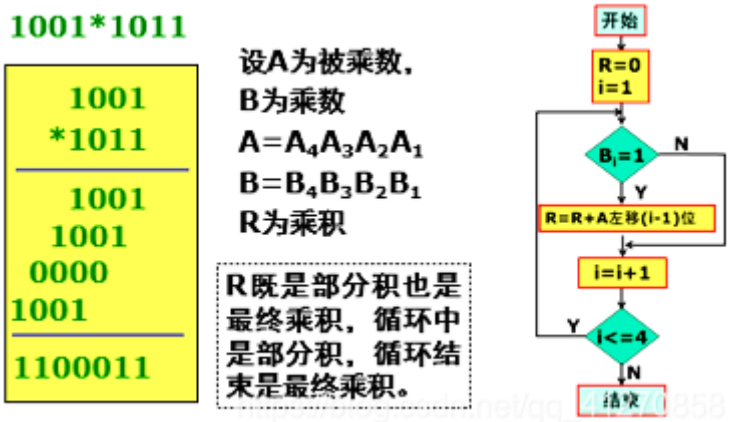
```
01  module adder_4bit(a,b,cin,sum,cout);
02      input[3:0] a,b; //定义 4 位的输入数
03      input cin;
04      output[7:0] sum;
05      output cout;
06      wire c1,c2,c3; //每一位依次经过了全加器的进位输出
07
```

```
08      full_adder u1(a[0],b[0],cin, c1,sum[0]); //第一位的 cin 为当前的进位输入
09      full_adder u2(a[1],b[1],c1,c2,sum[1]); //前一位的进位输出作为后一位的进位输入
10      full_adder u3(a[2],b[2],c2,c3,sum[2]);
11      full_adder u4(a[3],b[3],c3,cout,sum[3]); //最终的进位输出为当前进位输入
12  endmodule
```

[9] 两则 4 位乘法器

实例：4 位乘法器 for 循环增值法

```
01  module MULT4B(R,A,B);   //4 位乘法器
02  parameter S = 4;   //参数定义关键词 parameter（将常数用字符表示称为参数）
03  input [S:1] A,B; //A 为被乘数，B 为乘数
04  output [2*S:1] R; //R 为乘积
05  integer i; //i 为循环变量
06  reg [2*S:1] R; //always 语句中的赋值目标必须为 reg 型
07  always @ (A,B)
08      begin
09          R = 0;
10          for(i=1;i<=S;i=i+1) //循环 4 次
11              if(B[i]) R = R + (A<<(i-1)); //被乘数左移，与部分积相加
12              else R = R;
13      end
14  endmodule
```



实例：4 位乘法器 for 循环减值法

```
01  module MULT4B(R,A,B);
02  parameter S = 4;
03  input [S:1] A,B;
04  output [2*S:1] R;
05  reg [2*S:1] TA,R;//TA 为 A 的 2S 位扩展
06  reg [S:1] TB,TC;
07  always @ (A , B)
08      begin
```

```
09              R = 0;
10              TA = {{S{1'b0}},A};//将 A 扩展成 2S 位
11              TB = B;
12              TC=S
13              for(TC=S; TC>0 ; TC=TC-1)
14                  begin
15                      if(TB[1]) R = R + TA;//如果乘数右移后的最低位为 1
16                      else R = R;
17                      TA = TA << 1;//被乘数左移
18                      TB = TB >> 1;//乘数右移
19                  end
20      end
21  endmodule
```

[10] 2 位乘法器

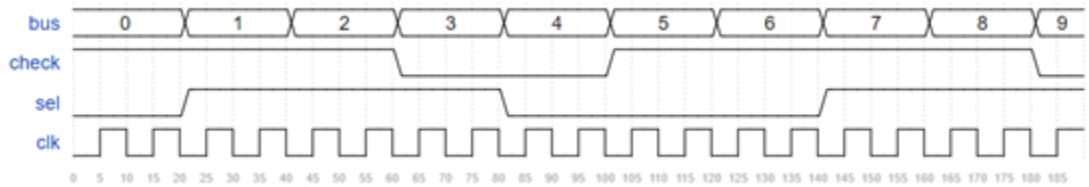
	实例：2 位乘法器 for 循环增值法
01	module MULT2B(R,A,B);　//2 位乘法器
02	parameter S = 2;　//参数定义关键词 parameter（将常数用字符表示称为参数）
03	input [S:1] A,B; //A 为被乘数，B 为乘数
04	output [2*S:1] R; //R 为乘积
05	integer i; //i 为循环变量
06	reg [2*S:1] R; //always 语句中的赋值目标必须为 reg 型
07	always @ (A,B)
08	begin
09	R = 0;
10	for(i=1;i<=S;i=i+1) //循环 2 次
11	if(B[i]) R = R + (A<<(i-1)); //被乘数左移，与部分积相加
12	else R = R;
13	end
14	endmodule

[11] 奇偶校验模块

对输入的 32 位数据进行奇偶校验,根据 sel 输出校验结果（1 输出奇校验，0 输出偶校验）

奇校验：原始码流+校验位 总共有奇数个 1

偶校验：原始码流+校验位 总共有偶数个 1



	实例：奇偶校验模块
01	module odd_sel(
02	input [31:0] bus,

```
03  input sel,
04  output check
05  );
06
07  /* 若没有说是几位的
08  input [width-1:0] bus;
09  parameter width = 8; //设置位数
10  */
11
12  wire odd;
13  assign odd = ^bus;
14  assign check = sel?odd:~odd;
15
16  endmodule
```

[12] 同步清零十进制计数器

在同步计数器中，当时钟脉冲输入时，触发器的翻转是同时发生的。而在异步计数中，触发器的翻转有先有后，不是同时发生的(CLK 时钟脉冲；RES 复位端；Q 计数器显示器；COUT 进位标识)

实例：同步清零十进制计数器(无使能端)

```
01  module CNT10(clk,rst,q,cout);
02        input clk,rst;
03        output cout;
04        output [3:0] q; //一枚十进制数需要 4 个位宽(9 为 1001)
05        reg [3:0] q;
06  always@(posedge clk) //上升沿触发，下降沿为 negedge
07     begin
08        if(!rst)
09        q <= 0;
10     else if(q >= 4'd9)
11        q <= 0;
12     else
13        q <= q + 1;
14  end
15        assign cout = (q == 4'd9 )? 1'b1: 1'b0; //判断是否有进位
    endmodule
```

使能的意思是负责控制信号的输入和输出。使能是芯片的一个输入引脚，或者电路的一个输入端口，只有该引脚激活，例如置于高电平时，整个模块才能正常工作。你可以想象成手枪或者灭火器的保险拴之类的东西。

实例：同步清零十进制计数器(有使能端)

```
01  module CNT10(clk, clr, en, q, cout);
02  //CLR 是芯片的复位输入，同 RES
03        input clk,clr,en;
```

```
04        output cout;
05        output[3:0] q;
06        reg[3:0] q;
07  always@(posedge clk)
08          begin
09          if(clr) q <= 0;  //有无感叹号取决于题目条件或者仿真波形图，一般情况是 1
10             else if (en)
11                  begin
12                    if(q == 4'd9) q <= 0;
13                    else q <= q + 1;
14                    end
15          end
16             assign cout = (q == 4'd9 )? 1'b1: 1'b0;
17  endmodule
```

[13] 异步清零十进制计数器

	实例：异步复位的同步计数器(有使能端，可预置型)
01	module CNT10 (CLK, RST, EN, LOAD, COUT, DOUT, DATA);
02	input CLK, EN, RST, LOAD; //时钟，时钟使能，复位，数据加载控制信号
03	input [3: 0] DATA; //4 位并行加载数据
04	output [3: 0] DOUT; //4 位计数输出
05	output COUT; //进位
06	reg [3: 0] Q1; //计数数值
07	reg COUT;
08	assign DOUT=Q1; //将内部寄存器的计数结果输出至 DOUT
09	always @ (posedge CLK , negedge RST) //时序过程
10	begin
11	if (!RST) Q1<=0; //RST=0 时，对内部寄存器单元异步清 0，看题
12	else if (EN)
13	begin //同步使能 EN=1，则允许加载或计数
14	if (!LOAD) Q1<=DATA; //当 LOAD=0,向内部寄存器加载数据
15	else if (Q1<9) Q1<=Q1+1; //当 Q1 小于 9 时，允许累加
16	else Q1<=4`b0000;
17	end //否则一个时钟后清 0 返回初值
18	end
19	assign COUT = (q == 4'd9)? 1'b1: 1'b0;
20	endmodule

[14] 60 进制计数器(例化)

无论是同步还是异步，60 进制计数器的例化和 100 进制计数器的例化代码均相同，区别只在于十进制计数器的同步或异步

60 进制例化，就是一个六进制计数器和一个十进制连上，所以要先准备一个六进制计数器，六进制计数器就把十进制的 9 改成 5，以下为 60 进制计数器表达

	实例：60 进制计数器(使用六进制和十进制的例化)拼合数显模块

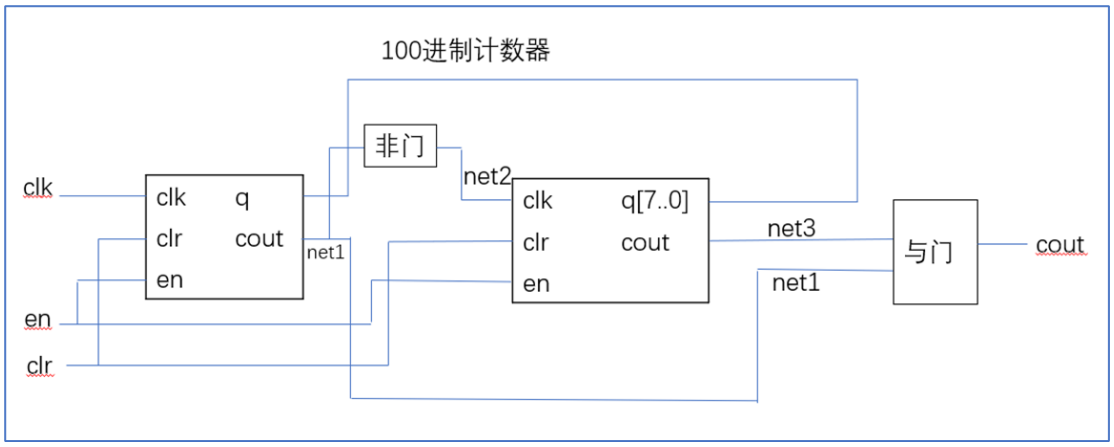
```
01  module CNT60(clk, clr, en, q1, q0, cout);
02      input clk;
03      input clr;
04      input en;
05      output[3:0] q1;   //高四位，十位
06      output[3:0] q0;   //低四位，个位
07      output cout;
08  wire net1, net2, net3; //1 是个位的进位，2 是 1 的非，3 是十位的进位
09      CNT10 U1(.clk(clk), .clr(clr), .en(en), .q(q0), .cout(net1));
10          not U2(net2, net1);   //非门（输出，输入）
11      CNT6 U3(.clk(net2), .clr(clr), .en(en), .q(q1), .cout(net3));
12          and U4(cout, net1, net3); //与门
13  endmodule
```

[16] 60 进制计数器(非例化)

```
    实例：60 进制计数器(非例化，异步清零，有使能端)非拼合数显模块
01  module counter60(in_en,in_clk,rest,out_munb);
02      input in_clk;//输入时钟
03      input rest;//清零
04      input in_en;//使能计数
05      output [5:0] out_munb;//输出计数
06      reg [5:0] out_munb;
07  always @(posedge in_clk,negedge rest)
08      begin
09              if(!rest)//异步清零
10                      begin
11                              out_munb <= 6'b0;
12                      end
13              else
14                      begin
15                              if(out_munb== 6'd59)//0-59 计数六十进制计数器
16                                      begin
17                                              out_munb <= 6'b0;
18                                      end
19                              else
20                                      begin
21                                              if(in_en)//计数使能判断  1  有效
22                                              begin
23                                              out_munb <= out_munb + 6'b1;
24                                              end
25                                      end
26                      end
27  end
28  endmodule
```

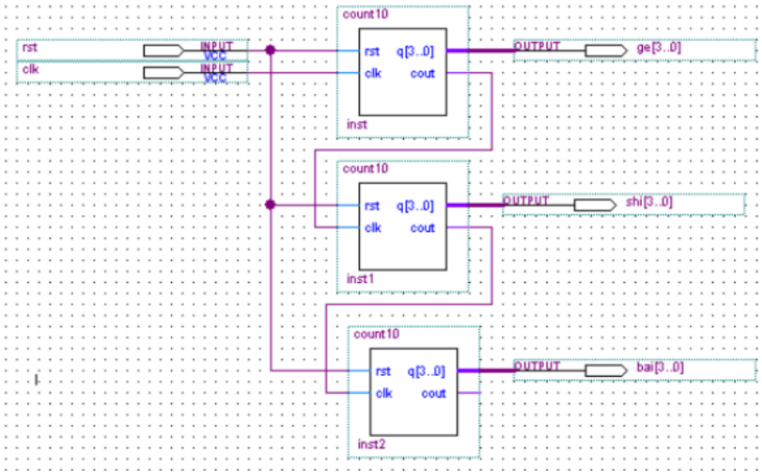
[17] 100 进制计数器(例化)

	实例：100 进制计数器
01	module CNT100(clk, clr, en, q1, q0, cout);
02	input clk;
03	input clr;
04	input en;
05	output[3:0] q1; //高四位，十位
06	output[3:0] q0; //低四位，个位
07	output cout;
08	wire net1, net2, net3; //1 是个位的进位，2 是 1 的非，3 是十位的进位
09	CNT10 U1(.clk(clk), .clr(clr), .en(en), .q(q0), .cout(net1));
10	not U2(net2, net1); //非门（输出，输入）
11	//这里默认低电平触发 clk，所以要用非门，否则直接连接，net2 就是 q0
12	CNT10 U3(.clk(net2), .clr(clr), .en(en), .q(q1), .cout(net3));
13	and U4(cout, net1, net3); //与门（输出，输入，输入…）
14	endmodule



[18] 999 计数器

无非就是三个 10 进制计数器叠在一起，也就是 100 进制计数器再加一个 10 进制变成 1000



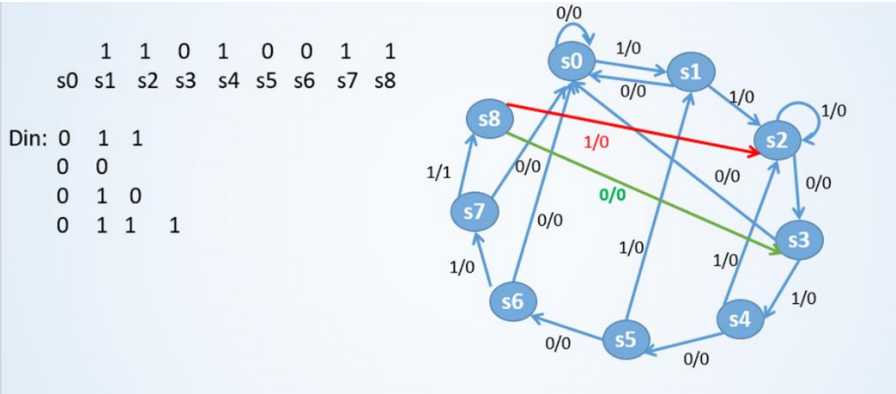
	实例：1000(999)进制计数器
01	module CNT1000(clk, clr, en, q2,q1, q0, cout);
02	input clk;
03	input clr;
04	input en;
05	output[3:0] q2; //百位
06	output[3:0] q1; //十位
07	output[3:0] q0; //个位
08	output cout;
09	wire net1, net2, net3,net4,net5;
10	//1 是个位的进位，2 是 1 的非，3 是十位的进位，4 是 2 的非 5 是百的进位
11	CNT10 U1(.clk(clk), .clr(clr), .en(en), .q(q0), .cout(net1));
12	not U2(net2, net1); //非门（输出，输入）
13	//这里默认低电平触发 clk，所以要用非门，否则直接连接，net2 就是 q0
14	CNT10 U3(.clk(net2), .clr(clr), .en(en), .q(q1), .cout(net3));
15	not U4(cout, net1, net3); //与门（输出，输入，输入…）
16	CNT10 U5(.clk(net4), .clr(clr), .en(en) .q(q2), .cout(net5));
17	and(cout,net1,net3,net5);
18	endmodule

[19] 序列检测器

太简单了，不想解释，例如 11010011

	实例：11010011 八位序列检测器
01	module SCHK(input CLK,DIN,RST,output SOUT);
02	parameter s0=40,s1=41,s2=42, s3=43, s4=44,s5=45, s6=46,s7=47,s8=48;
03	//设定 9 个状态参数（这个我也不理解，老师说是习惯下标记法）
04	reg[8:0] ST, NST; //设定现态变量和次态变量
05	always @(posedge CLK or posedge RST)
06	if (RST) ST<=s0;
07	else ST<=NST;
08	always @(ST,DIN)
09	begin
10	case(ST) //11010011 串行输人，高位在前
11	s0 : if (DIN==1'b1) NST<=s1; else NST<=s0;
12	s1 : if (DIN==1'b1) NST<=s2; else NST<=s0;
13	s2 : if (DIN==1'b0) NST<=s3; else NST<=s2;
14	s3 : if (DIN==1'b1) NST<=s4; else NST<=s0;
15	s4 : if (DIN==1'b0) NST<=s5; else NST<=s2;
16	s5 : if (DIN==1'b0) NST<=s6; else NST<=s1;
17	s6 : if (DIN==1'b1) NST<=s7; else NST<=s0;
18	s7 : if (DIN==1'b1) NST<=s8; else NST<=s0;
19	s8 : if (DIN==1'b0) NST<=s3; else NST<=s2;
20	default : NST<=s0; //写回 s0
21	endcase

22	end
23	assign SOUT=(ST==s8);　//这个我也不知道什么意思就背着吧
24	endmodule



[20] 移位寄存器

用于移动位并完成 2 的幂的乘法或除法，左移或右移，以 0 填充空位

	实例：8 位移位寄存器(同步预置)
01	module Regist(clk,load,din,qb);
02	input clk,load; //时钟信号，移位信号
03	input[7:0]din; //8 位数据
04	output qb;
05	reg[7:0] r; //用于运算
06	always@(posedge clk)
07	if(load) r<=din;
08	else r[6:0]<=r[7:1]; //位移信号为低电平时执行位移
09	assign qb=r[0]; //接收结果
10	endmodule

	实例：流水灯
01	module led_water(led,clk);
02	input clk;
03	output [7:0]led;
04	reg[7:0]led_r;
05	assign led=led_r[7:0];
06	always @(posedge clk)
07	begin
08	led_r={led_r[5:0],led_r[7:6]};
09	if(led_r==8'd0)led_r=8'b00111111;
10	end
11	endmodule

[21] 三八译码器

3-8 译码器，就是把 3 种输入状态翻译成 8 种输出状态，译码器是将输入的具有特定含义的二进制代码翻译成输出信号的不同组合，实现电路控制功能的逻辑电路

	实例：3-8 译码器(不含控制器)
01	module decoder(
02	input wire [2:0] a,//输入信号，3 位
03	output reg [7:0] b//输出信号，8 位
04	);
05	//译码器组合逻辑
06	always@(*)begin
07	case(a)
08	3'b000: b=8'b11111110;
09	3'b001: b=8'b11111101;
10	3'b010: b=8'b11111011;
11	3'b011:b=8'b11110111;
12	3'b100: b=8'b11101111;
13	3'b101:b=8'b11011111;
14	3'b110:b=8'b01111111;
15	3'b111:b=8'b10000000;
16	default: b=8'b00000000;
17	endcase
18	end
19	endmodule

	实例：3-8 译码器(含使能与译码电平控制)
01	module yi_ma_qi138(INA,Y_01, EN,Y_DOUT);
02	input [2:0] INA;//三位　输入
03	input Y_01;//输出有效 电平控制
04	input EN;//使能控制端
05	output [7:0] Y_DOUT;//输出
06	reg [7:0] Y_DOUT;//输出
07	always @(INA)
08	begin
09	if(EN)
10	begin
11	if(Y_01==0）//译码电平 0（否则是 1）
12	begin
13	case(INA)
14	3'b000: Y_DOUT = 8'b1111_1110;
15	3'b001: Y_DOUT = 8'b1111_1101;
16	3'b010: Y_DOUT = 8'b1111_1011;
17	3'b011: Y_DOUT = 8'b1111_0111;
18	3'b100: Y_DOUT = 8'b1110_1111;
19	3'b101: Y_DOUT = 8'b1101_1111;
20	3'b110: Y_DOUT = 8'b1011_1111;
21	3'b111: Y_DOUT = 8'b0111_1111;
22	endcase

```
23              end
24              else //译码电平  1
25              begin
26                  case(INA)
27                      3'b000: Y_DOUT = 8'b0000_0001;
28                      3'b001: Y_DOUT = 8'b0000_0010;
29                      3'b010: Y_DOUT = 8'b0000_0100;
30                      3'b011: Y_DOUT = 8'b0000_1000;
31                      3'b100: Y_DOUT = 8'b0001_0000;
32                      3'b101: Y_DOUT = 8'b0010_0000;
33                      3'b110: Y_DOUT = 8'b0100_0000;
34                      3'b111: Y_DOUT = 8'b1000_0000;
35                  endcase
36              end
37          end
38  end
39  endmodule
```