Verilog语言笔记

**Vivado的使用**

1. 选择板子的时候,请选择正确的板子

2.Vivado的文件分成三个部分:一个是设计文件,一个是约束文件,一个是仿真文件,这些都可以右键添加

3.在左侧的 *Flow Navigator*（流程导航器）中，展开 *RTL Analysis*（RTL 分析）任务下

*Elaborated Design*（精密设计）条目，点击 **Schematic**（原理图）。 可以展开原理图

4.在 *Flow Navigator* 的 **Simulation** 任务下选择 **Simulation Settings**（仿真设置），将弹

出一个显示仿真属性的 **Project Settings** 窗口。 在这里可以设置仿真时间

5.在 *Flow Navigator* 的 **Simulation** 任 务 下 点 击 **Run Simulation > Run Behavioral**

**Simulation**（运行仿真>运行行为仿真）。

6.在 *Flow Navigator* 的 *Program and Debug* 任务下，点击 **Generate Bitstream**（生成比

特流）。

比特流生成过程将基于被实现的设计进行。处理结束，具有两个选项的 *Bitstream*

*Generation Completed*（比特流生成完成）对话框将被弹出。

剩下的部分可以在lab1的文件里查看

**模块**

模块（module）是Verilog的基本描述单位，用于描述某个设计的功能或结构，及其与其他模块通信的外部端口。

一个模块可以在另一个模块中使用。

模块由关键词module和endmodule进行定义

Verilog程序由关键词module和endmodule进行定义

Verilog HDL 大小写敏感

**怎么实现一个模块**

Verilog语言的关键词不能用作模块名

（例如：module/port/signal 等）

1.使用一个有意义的模块名

2.定义模块的端口（connectivity）

每个端口都会连接一个信号（Signal）

2.1申明端口的类型

Input&output&inout （双向）

2.1.1单位- 不需要给出信号的位数

input cin;

2.1.2 向量形式- 需要定义具体的位数

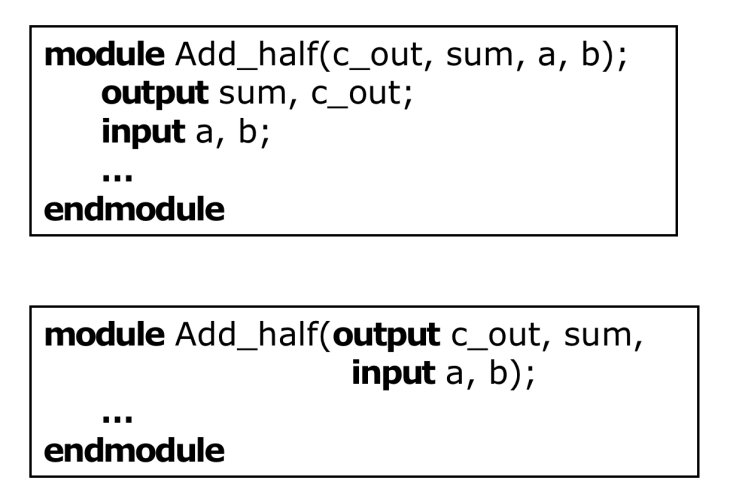
Range is MSB to LSB (left to right)

可以不从0开始定义 (D[2:1])

output [7:0] OUT;

input [0:4] IN;

2.1.3 括号里面的定义方式

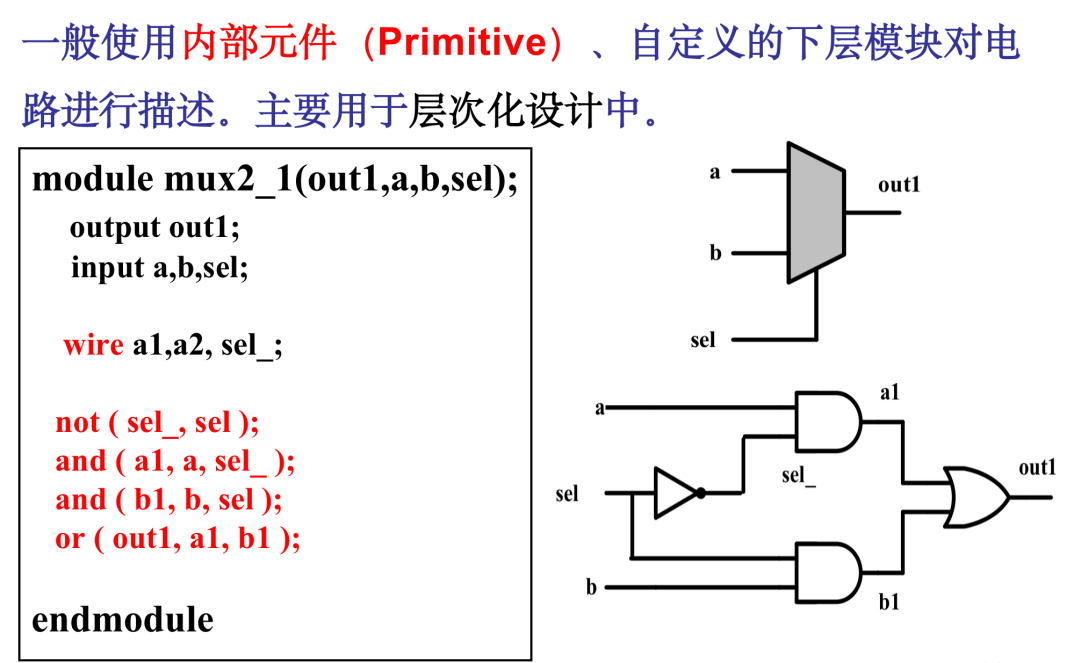


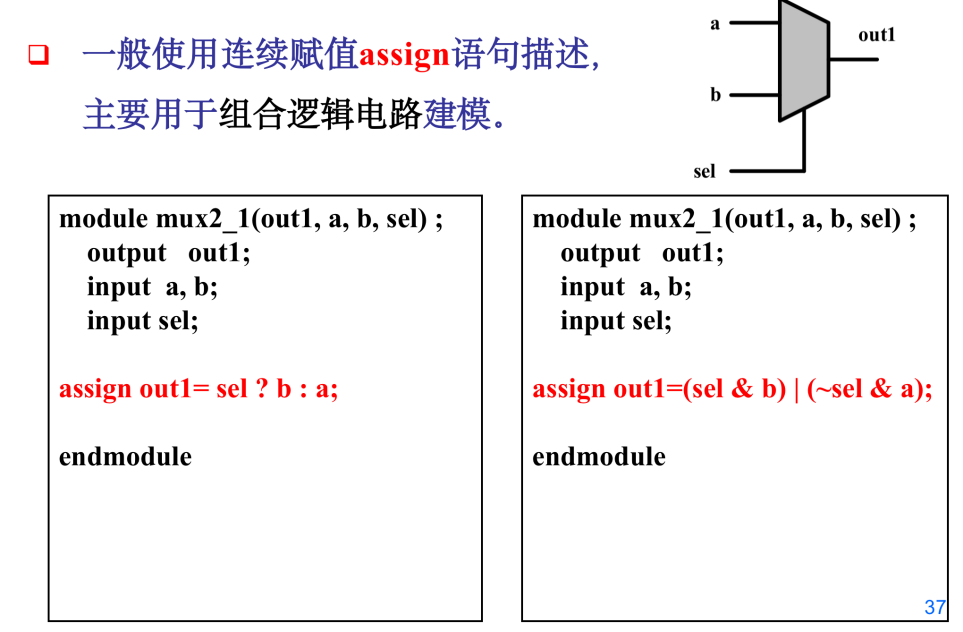
3.定义模块内部连接到端口的信号类型

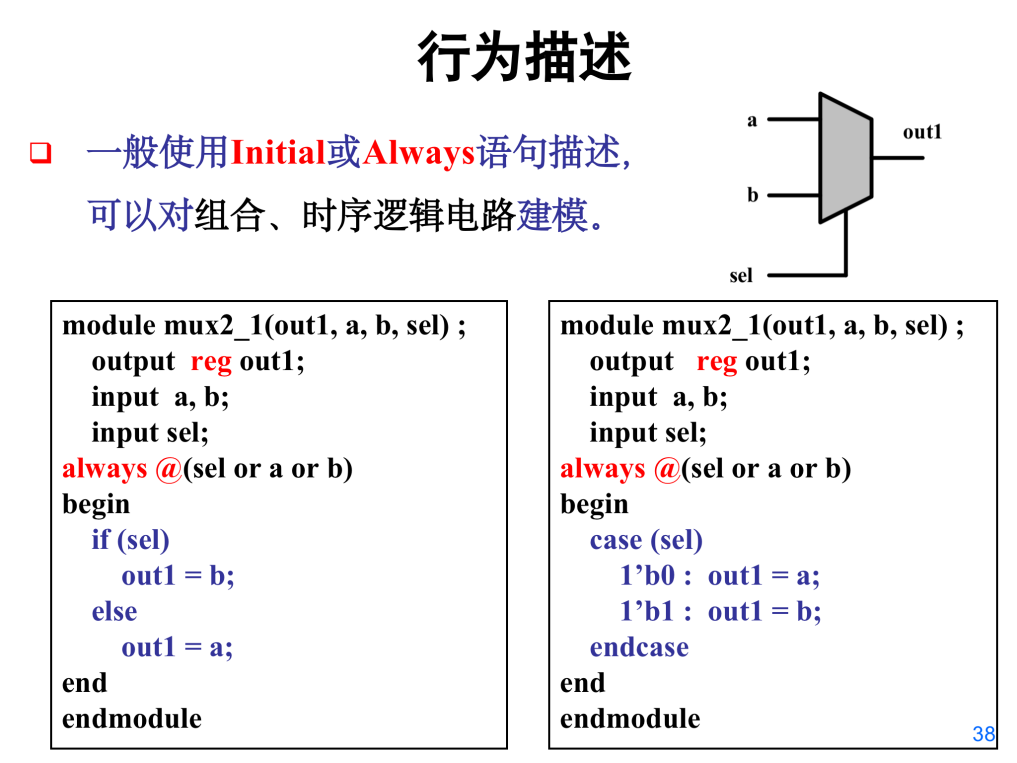
3.1 结构描述(图里面wire表示导线)

3.2 数据流描述

3.3 行为描述(initial表示只进行一次,always表示满足就进行)







4.定义内部信号

5.描述模块内部实现的功能（ functionality）

**怎么测试一个模块**

这个测试模块应包括以下三个方面的内容：

测试模块中要调用到设计块，只有这样才能对它进行测试；

测试模块中应包含测试的激励信号源；

测试模块能够实施对输出信号的检测，并报告检测结果。

测试程序会在之后提到

**注释**

可以使用//或者/\*

**一个verilog工程的流程**

步骤1 创建一个Vivado工程

步骤2 添加设计文件(.v)

步骤3 添加约束文件(.xdc-Xilinx Design Constraints)

步骤4 添加testbench文件(.v)

步骤5 仿真：测试设计块

步骤6 综合：将语言转化成电路（网表文件）

步骤7 实现：将网表配置到具体的FPGA芯片

步骤8 生成比特流，下载到板子进行功能验证

**标识符**

可以是任意一组字母、数字、$符号和\_(下划线)符号的组合；

必须是由字母或下划线开头，长度小于1024字符；

转义标识符以反斜杠“\”开头，以空白符结尾的任何字符序列；

标识符区分大、小写。

**四种基本的逻辑值**

0：逻辑0或“假”

1：逻辑1或“真”

x：未知

z：高阻

**三类常量**

整型数：简单的十进制格式，基数格式（5’O37，4’B1x\_01）

实数：十进制计数法，科学计数法

字符串：字符串是双引号的字符序列，字符串不能分成多行书写

**数字**

Examples:前面加n’表示n位

4’b1101 // this is a 4-bit binary number equal to 13

10’h2e7 // this is a 10-bit wide number specified in hex

Available bases:

d = decimal (please only use in test benches)十

h = hex (use this frequently)

b = binary (use this frequently for smaller #’s)二

Numbers can have x or z characters as values

x = unknown, z = High Impedance(高阻)

12’h13x // 12-bit number with lower 4-bits unknown

**参数**

parameter：是个常量

范围 -> 本module内有效

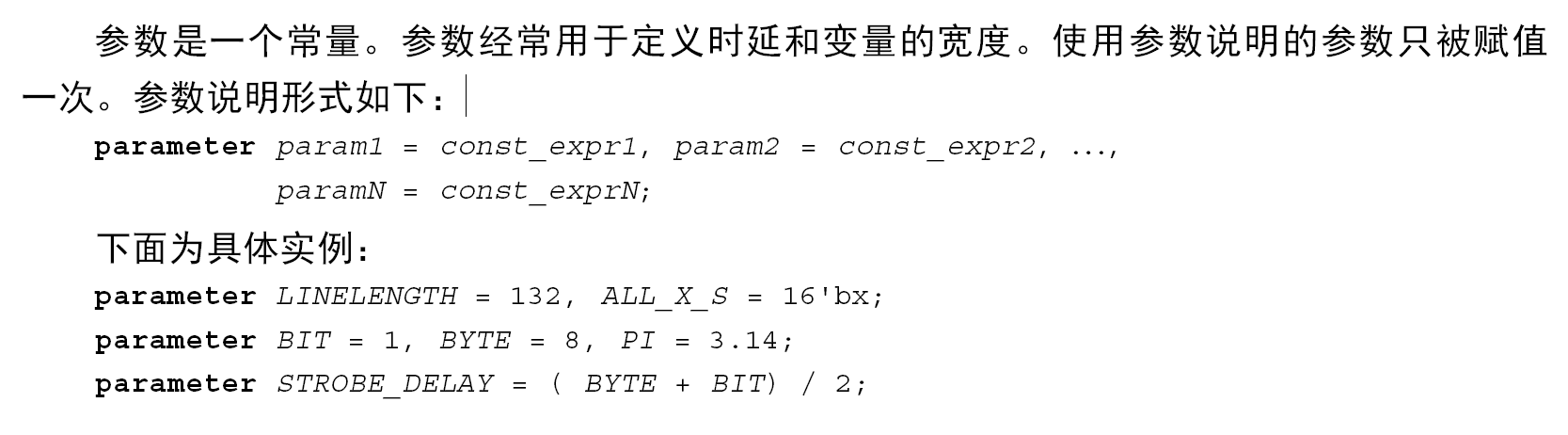
作用 -> 模块间参数传递

在顶层模块中#(N)表示传参

例子 register .....

parameter WIDTH=8

register #(32)(传递参数)



**define**

`define：定义常量

范围 -> 整个工程

作用 -> 常用于定义常量可以跨模块、跨文件

**本地参数**

localparam：常量

范围 -> 本module内有效的定义，不可用于参数传递;

作用 -> 常用于状态机的参数定义;

**线网类型（wire）**

net type表示Verilog结构化元件间的物理连线。它的值由驱动元件的值决定；如果没有驱动元件连接到线网，线网的缺省值为z。**就是输入输出可以连接到线网**

wire和tri线网：是最常见的线网类型。

wor和trior线网：如果某个驱动源为1，那么线网的值也为1。

wand和triand线网：如果某个驱动源为0，那么线网的值为0。

**寄存器类型（reg）**

register type表示一个抽象的数据存储单元，它只能在always语句和initial语句中被赋值，并且它的值从一个赋值到另一个赋值被保存下来。寄存器类型的变量具有x的缺省值。

**数组**

可以定义这样的数组:reg [7:0] mem[0:99][0:3];

可以传入传出数组

**r e g** [0:3 ] *M y M e m* [ 0 : 6 3 ]

/ /*M y M e m*为6 4个4位寄存器的数组。

**存储形式**

RF, SRAM, ROM, Flash, Cache

有时候会很有用

**系统任务和函数**

以$字符开始的标识符表示系统任务或系统函数；

任务提供了一种封装行为的机制，任务可以返回0个或多个值；

函数除只能返回一个值以外与任务相同；

函数在0时刻执行，即不允许延迟，而任务可以带有延迟。

$display:printf

$display(“At time %t count = %h”,$time,cnt);

$stop  停止仿真,但是允许你单步调试

$finish  完全停止

**时延**

Verilog HDL模型中的所有时延都根据单位定义。

下面是带时延的连续赋值语句实例：

assign #2 Sum = A ^ B; // #2指2个时间单位。

and #1 A0(OUT, A, B);

如果没有说明时延时间单位，Verilog HDL模拟器会指定一个缺省时间单位。

IEEE Verilog HDL标准中没有规定缺省时间单位。

可以设置0延时,但不可以设置1.5ns,延时取决于物理特性

延迟分为传输延迟和惯性延迟,不过没关系

wire #2 z\_out; //

and #3 (z\_out, x\_in, y\_in); // 3 for 门, 2 for 导线

**运算符（9类）**

算术运算符： +、-、\*、/、%

位运算符： ~、&、|、^、^~ 或 ~^（异或非）

缩位运算符（单目）： &、~&(与非)、|、~|、^、^~、~^

逻辑运算符： !、&&、||

关系运算符（双目）： <、>、<=、>=

相等与全等运算符： ==（逻辑相等）、!=（逻辑不等）、

===（全等）、!==（非全等）

逻辑移位运算符： <<、>>

连接运算符： { }

条件运算符： ?:

{}:

module concatenate(out, a, b, c, d);

input [2:0] a;

input [1:0] b, c;

input d;

output [9:0] out;

assign out = {a[1:0],b,c,d,a[2]};

endmodule

[]:[a1a0b1b0c1c0da2]

?:

assign {cout,dst} = (op==`add) ? src1+src2+cin :

(op==`and) ? {1’b0,src1 & src2} :

(op==`xor) ? {1’b0,src1 ^ src2} :

(op==`shft\_l) ? {src1,cin} :

(op==`shft\_r) ? {src1[0],src1[15],src1[15:1]} :

17’h00000;

**用户定义的原语**

primitive UDP 名 ( 输出端口名, 输入端口名);

output 输出端口名; //只允许一个输出端口

input 输入端口名;

reg 输出端口名; //可选，时序逻辑的UDP才需要

initial 输出端口名=值 //可选，时序逻辑的UDP才需要

table

状态表

endtable

endprimitive

在组合电路UDP中，表规定了不同的输入组合和相对应的输出值。

在时序电路UDP中，使用1位寄存器描述内部状态。该寄存器的值是时序电路UDP的输出值。共有两种不同类型的时序 电路UDP：

模拟电平触发行为

模拟边沿触发行为

时序电路UDP使用寄存器当前值和输入值决定寄存器的下一状态（和后继的输出）。

原语就是操作系统层面上的

**结构描述的方法**

Verilog提供很多自定义的门

and, nand, or, nor, xor, xnor, not, buf, bufif1, etc.

First定义模块的接口.模块关键词和模块名,Port names/types/sizes(接口的名字,类型和大小)

Next, 把所有的中间变量全部定义成wire类型

Like:wire [3:0] partialsum;

Then instantiate the primitives/submodules(定义子模块)

Indicate which signal is on which port(确认哪个信号在哪个模块)

可以按照下面三个步骤进行：

（1）把系统划分成模块；

（2）规划各模块的接口；

（3）对模块编程并连接各模块完成系统设计。

模块的结构：

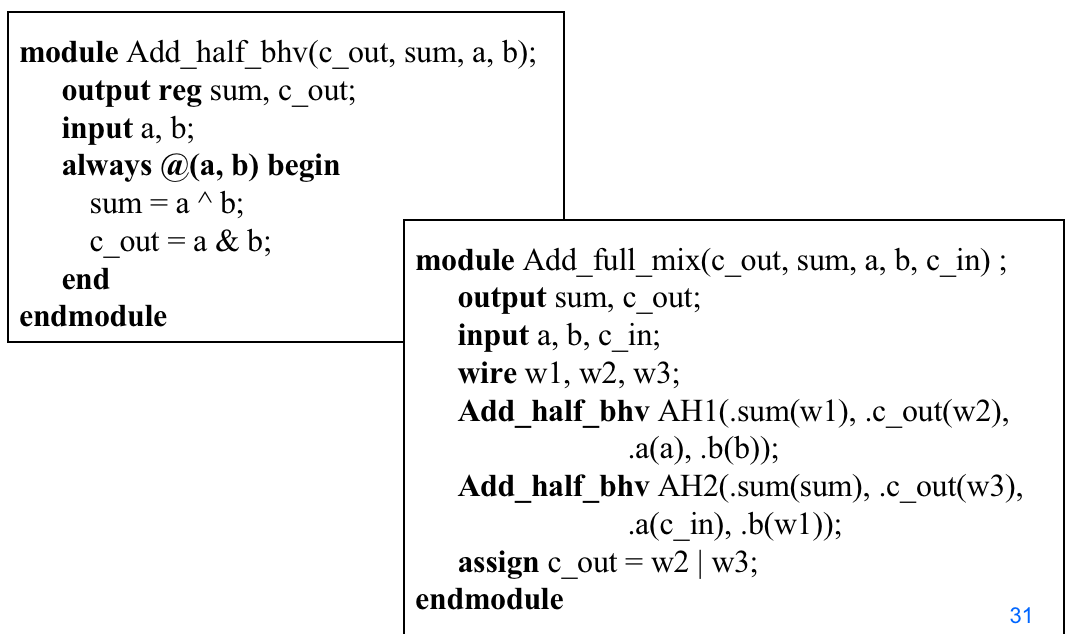
module <模块名>（<端口列表>）；

<定义>：一段程序，用来指定数据对象为寄存器型、存储器型、线型以及过程块，诸如函数块和任务块；

<模块条目>：一段程序，将上面<定义>和<端口>组合起来，是说明这个模块要做什么的语句；

endmodule

可以使用wire类型的变量来进行模块间信息的传递



**模块的调用**

1）引用时，严格按照模块定义的端口顺序来连接，不用标明原模块定义时规定的端口名。

Design u\_1(u\_1的端口1，u\_1的端口2，……)；

2）引用时用”.”符号，标明原模块定义时规定的端口名， 这样当被调用的模块管脚改变时不易出错。

Design u\_2( .(端口1(u\_1的端口1),.(端口2(u\_1的端口2), …… );

**传入空的数据**

Empty input ports => high impedance state (z) 高阻

Empty output ports => output not used

**数据流形式的例子**

module mac(output [31:0] Z, output overflow,

input [15:0] A, B, input [31:0] C);

assign {overflow, Z} = A\*B + C;

endmodule

利用assign{}和一些运算符号来表示

**initial**

module full\_adder\_tb;

reg [3:0] stim;

wire s, c;

full\_adder(sum, carry, stim[2], stim[1], stim[0]); // instantiate DUT

// monitor statement is special - only needs to be made once,

initial $monitor(“%t: s=%b c=%b stim=%b”, $time, s, c, stim[2:0]);

// tell our simulation when to stop

initial #50 $stop;

initial begin // stimulus generation

for (stim = 4’h0; stim < 4’h8; stim = stim + 1) begin

#5;

end

end

endmodule

每一个initial块都是从时间t=0开始,并且只会执行一次,在前面加上#...在第几秒的时候值为什么,并且每个initial块都会值执行一次

[当然还可以使用@(....)来进行处理](mailto:当然还可以使用@(....)来进行处理)

初始陈述不综合

**Always**

在零时间执行，但连续循环

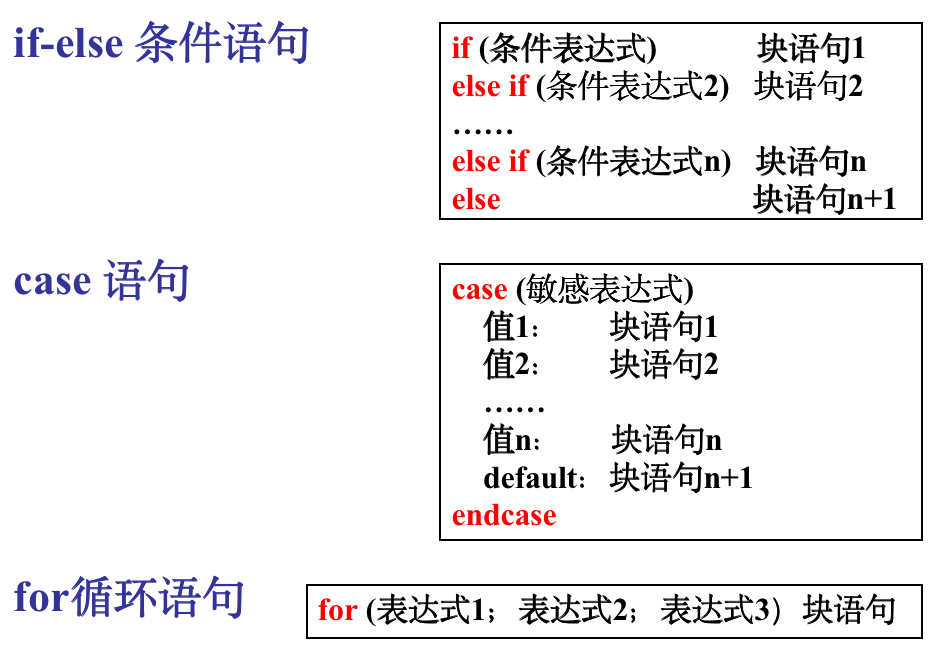
可以使用触发器列表来控制操作； @（a，b，c）(当触发器里面的内容改动了就进入always里面的操作)

在没有触发器列表的情况下，它将在最后一个<LHS>分配完成时重新判断要不要进入.

@()对于每一个输入都敏感,任何一个发生变化都会进入

并行运行的

**条件语句**



对于case类型的语句,可以侦测x和z

对于casez类型的语句,可以把z和?给忽略掉

对于casex类型的语句,可以把z和x和?给忽略掉

**循环语句**

while

while ((index<16) && (!found)) begin

if (flag[index]) found = 1’b1;

else index = index + 1;

end

repeat

repeat(4095) @(posedge clk); // bring DAC right up to point of rollover

inc\_DAC = 1’b0;

这里是重复4095次(一次时钟改变算一次)

forever:一直在做的循环

**赋值语句**

阻塞型过程赋值

**赋值算符“＝” ：前一条语句没有完成赋值过程之前，后面的语句不可能被执行。**

**非阻塞型过程赋值**

**赋值算符“<＝” ：一条非阻塞型赋值语句的执行，并不会影响块中其它语句的执行。并行的赋值!**

**连续赋值语句(这是一定在always和initial使用)**

只要输入端操作数的值发生变化，该语句就重新计算并刷新赋值结果。用关键词assign来区分。(如果不在always和initial里面)

1、时序逻辑，使用“非阻塞赋值”。

2、组合逻辑，使用“阻塞赋值”

3、 当在同一个always块里面既为组合逻辑又为时序逻辑时，使用“非阻塞赋值”。

**时序控制语句**

由“#”符号引入的延迟控制。它将程序的执行过程中断一定的时间，时间的长度由<time>的值来确定。

#<time><statement>

由“@”符号引入的事件控制。一个事件可以通过运行表达式“-> event”变量被激发。

@ (<posedge>|<negedge>|<signals> ) <statement>；

等待语句。直到表达式计算为真之前，都延时下一个语句的执行。

wait (<expression>) <statement> ；

**函数和任务**

函数（function）

任务（task）

函数和任务结构之间的差异：

一个任务块可以含有时间控制结构，而函数块则没有；

一个任务块可以有输入和输出，而函数块必须有至少一个输入，没有任何输出；

任务块的引发是通过一条语句，而函数块只有当它被引用在一个表达式中时才会生效。

function <位宽说明> 函数名；

输入端口与类型说明；

局部变量说明；

块语句

endfunction

task 任务名；

端口与类型说明；

局部变量说明；

块语句

endtask

**加法器**

1位全加器

module full\_adder(a, b, c\_in, sum, c\_out);

input a, b, c\_in;

output sum, c\_out;

// 学习拼接运算符的用法

assign { c\_out, sum } = a + b + c\_in;

endmodule

(数据流描述)

module full\_adder(a, b, c\_in, sum, c\_out);

// 学习矢量的申明,注意是[大数:小数]

input [3:0] a, b;

input c\_in;

output [3:0] sum;

output c\_out;

assign { c\_out, sum } = a + b + c\_in;

endmodule

(数据流描述)

module full\_adder(a, b, c\_in, sum, c\_out);

// 学习 parameter 的申明及使用

**parameter WIDTH = 8;(函数的临时参数)**

input [WIDTH-1:0] a, b;

input c\_in;

output [WIDTH-1:0] sum;

output c\_out;

assign { c\_out, sum } = a + b + c\_in;

endmodule

**数值比较器**

// 调用语法

full\_adder **#(.WIDTH(32))** dut(a, b, c\_in, sum, c\_out);//学习如何传递参数(parameter)

或者直接full\_adder **#(32)**

module comparator(a, b, is\_equal, is\_great, is\_less);

parameter WIDTH = 8;

input [WIDTH-1:0] a, b;

output is\_equal, is\_great, is\_less;

assign is\_equal = (a == b) ? 1'b1 : 1'b0;

assign is\_great = (a > b) ? 1'b1 : 1'b0;

assign is\_less = (a < b) ? 1'b1 : 1'b0;

endmodule

**译码器**

// 3-8 译码器

module decoder\_38(A, Y);

input [2:0] A;

output reg [7:0] Y;

always @(A)

begin

case(A)

3'd0: Y = 8'b1111\_1110;

3'd1: Y = 8'b1111\_1101;

3'd2: Y = 8'b1111\_1011;

3'd3: Y = 8'b1111\_0111;

3'd4: Y = 8'b1110\_1111;

3'd5: Y = 8'b1101\_1111;

3'd6: Y = 8'b1011\_1111;

3'd7: Y = 8'b0111\_1111;

endcase

end

endmodule

在initial和always块里面声明的变量一定得是reg类型的变量

**选择器**

// 2-1 选择器

module mux\_21(a, b, sel, out);

parameter WIDTH = 8;

input [WIDTH-1:0] a, b;

input sel;

output [WIDTH-1:0] out;

// 三元运算符

assign out = sel == 0 ? a : b;

//这里就不能把assign去掉

endmodule

// 4-1 1 bit 选择器设计方案I

module mux\_41(a, b, c, d,

sel, out);

input a, b, c, d;

input [1:0] sel;

output reg out;

always @(a, b, c, d, sel)

begin

case(sel)

2'b00: out = a;

2'b01: out = b;

2'b10: out = c;

default: out = d;

endcase

end

endmodule

// 4-1 选择器设计方案 II

module mux\_41(a, sel, out);

parameter WIDTH = 8;

**input [WIDTH-1:0] a[0:3];**//注意二维数组的生成int a[x][y] [y]a[x]

[维度1:0] ... [0:维度2]

input [1:0] sel;

output [WIDTH-1:0] out;

// 利用数组

assign out = a[sel];

Endmodule

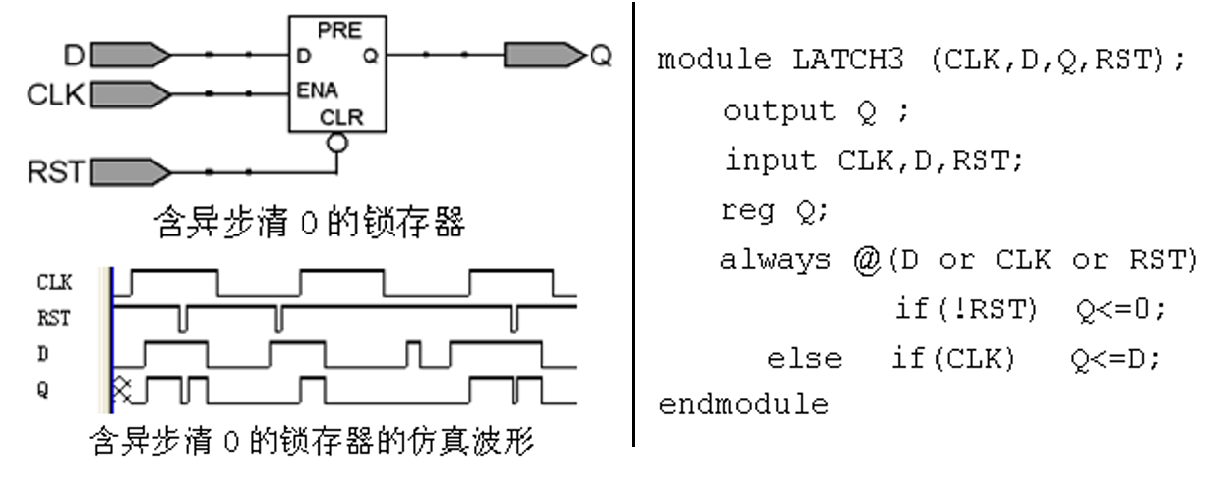
**触发器**

含有异步复位和使能

手机屏幕截图

描述已自动生成

锁存器



**计数器**

// 实用模 10 计数器

module cnt10(clk, rst\_n, en, load\_n, data, cout, dout);

//load是直接赋值

input clk, rst\_n, en, load\_n;

input [3:0] data;

output cout;

output reg [3:0] dout;

always @(posedge clk, negedge rst\_n) begin

if (!rst\_n) dout <= 0;//置0脉冲

else if (en) begin

if (!load\_n) dout <= data;//置初值

else if (dout < 4'd9) dout <= dout + 1;//进1

else dout <= 4'd0;//进位清零

end

end

assign cout = (dout == 4'd9) ? 1'b1 : 1'b0;//利用数据流描述来进行输出的定义,assign可以完美定义输出

endmodule

test的代码

module cnt10\_tb( );

reg clk,rst\_n,en,load\_n;

reg [3:0] data;

wire cout;

wire [3:0] dout;

cnt10 dut(clk,rst\_n,en,

load\_n,data,cout,dout);

// 产生 10ns 周期的时钟信号

initial clk = 0;

always #5 clk = ~clk;

initial begin

// #100

rst\_n = 1;

en = 1;

load\_n = 1;

data = 3;

// 使能为0期间，不计数

#10 en = 0;

#15 en = 1;

// 异步复位，不受时钟影响

#6 rst\_n = 0;

#2 rst\_n = 1;

// 预置数改变不影响计数

#10 data = 7;

// 直到预置数被装载

#20 load\_n = 0;

#10 load\_n = 1;

// 大于10的预置数

#30 data = 11;

#30 load\_n = 0;

#10 load\_n = 1;

end

endmodule

**寄存器**

module register(clk, rst\_n, en, d, q);

parameter WIDTH = 8;

input clk, rst\_n, en;

input [WIDTH-1:0] d;

output reg [WIDTH-1:0] q;

always @(posedge clk) begin

if (!rst\_n) q <=0;//置0脉冲

else if (en) q <= d;//如果en存在,那么就保存

end

endmodule

**存储器**

module ram(data, read\_addr, write\_addr, clk, we, q);

parameter DATA\_WIDTH = 8;

parameter ADDR\_WIDTH = 3;

input clk, we;

input [DATA\_WIDTH-1:0] data;

input [ADDR\_WIDTH-1:0] read\_addr, write\_addr;

output reg [DATA\_WIDTH-1:0] q;

// 申明存储器数组

reg [DATA\_WIDTH-1:0] ram[2\*\*ADDR\_WIDTH-1:0];

always @(posedge clk) begin

if (we)

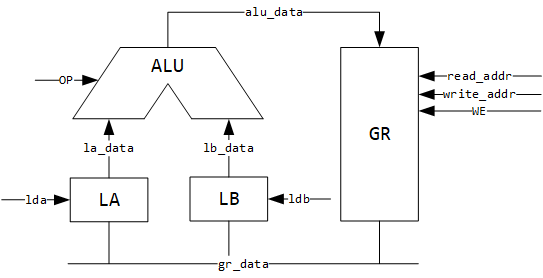
ram[write\_addr] <= data;//存入数据

q <= ram[read\_addr];//读取数据

end

endmodule

**数据通路的示例**



module alu(a, b, op, q);

parameter WIDTH = 8;

input [WIDTH-1:0] a, b;

input [1:0] op;

output reg [WIDTH-1:0] q;

always @(\*) begin

case(op)

2'b00: q = a + b;

2'b01: q = a & b;

2'b10: q = a ^ b;

2'b11: q = a | b;

default: q = 0;

endcase

end

endmodule

module datapath\_top(clk, rst, lda, ldb,

read\_addr, write\_addr,

we, op);

input clk, rst, lda, ldb, we;//输入

input [4:0] read\_addr, write\_addr;

input [1:0] op;

wire [31:0] gr\_data, alu\_data;//连线

wire [31:0] la\_data, lb\_data;

//利用模块声明来定义模块

register #(32) LA (clk, rst, lda, gr\_data, la\_data);

register #(32) LB (clk, rst, ldb, gr\_data, lb\_data);

ram #(32, 5) GR (alu\_data, read\_addr, write\_addr,

clk, we, gr\_data);

alu #(32) ALU (la\_data, lb\_data, op, alu\_data);

endmodule

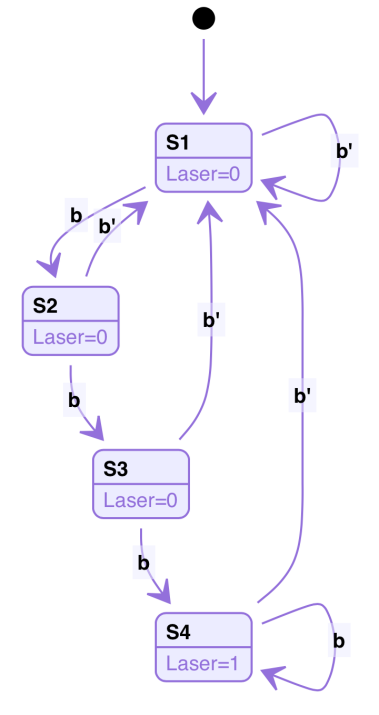
**有限状态机**

有限状态机的行为可以分为三部分：

计算下一个状态。下一个状态和当前状态及输入条件相关，与 clk 无关，可以用“组合逻辑”实现。

从当前状态迁移到下一个状态。在 clk 的上升沿完成迁移，是典型的“同步时序逻辑”。

计算输出。有两类 FSM，输出只和当前状态相关的，称为 Moore 型 FSM；输出和当前状态、输入相关的，称为 Mealy 型 FSM。可以用 “组合逻辑”实现。

module laser\_timer(clk,rst,b,x);

input clk, rst, b;

//时钟,

output reg x;//输出

localparam S0 = 0, S1 = 1,

S2 = 2, S3 = 3;

reg [1:0] State, StateNext;//状态,下一个状态

always @(State, b) begin

case(State)

//初态为S0:

S0: begin

x <= 0;

if (b == 1)

StateNext <= S1;

else

StateNext <= S0;

end

S1: begin

x <= 0;

if (b == 1)

StateNext <= S2;

else

StateNext <= S0;

end

S2: begin

x <= 0;

if (b == 1)

StateNext <= S3;

else

StateNext <= S0;

end

S3: begin

这个是Moore型的,每一次进入的时候就进行输出

x <= 1;

if (b == 1)

StateNext <= S3;

else

StateNext <= S0;

end

endcase

end

//两段有限状态机的处理

always @(posedge clk)

//这个是rst的处理,如果rst为1,就置0

begin

if (rst == 1)

State <= S0;

else

State <= StateNext;

end

endmodule

**仿真程序的注意事项**

总是使用 timescale 规定时间

在全局置位、复位脉冲释放之前就确保时钟源已经开始工作。

在仿真时间的 0 时刻，将所有的设计输入初始化位为一个确定的值；

在综合后和实现后的时序仿真中，会自动触发全局置位/复位脉冲（GSR），这会让所有的寄存器在仿真的前 100ns 内锁定其值。因此在 100ns 之后再赋值激励数据；

**退出循环的方式**

disable语句可以退出任何循环，能够终止任何begin..end块的执行，用于仿真验证中。

$finish和$stop都可以进行仿真

一个有限状态机的设计

module test3(

input clk,

input reset,

input w,

output [2:0] state,

output [2:0] next\_state,

output z

);

// 请完成模块的设计

reg [2:0]state;

reg [2:0]next\_state;

reg z;

always @(posedge clk)begin

if(reset == 1)begin

state <= 3'b000;

next\_state <= 3'b000;

z <= 0;

end

else

state <= next\_state;

end

always @(w or state) begin

z <=0;

case(state)

3'b000:

if(w == 1) next\_state <= 3'b000;

else next\_state <= 3'b001;

3'b001:

if(w == 1) next\_state <= 3'b011;

else next\_state <= 3'b010;

3'b010:

if(w == 1) next\_state <= 3'b011;

else next\_state <= 3'b100;

3'b011:

if(w == 1) next\_state <= 3'b000;

else next\_state <= 3'b101;

3'b100:begin

if(w == 1) next\_state <= 3'b011;

else next\_state <= 3'b100;

z <= 1;

end

3'b101:begin

if(w == 1) next\_state <= 3'b011;

else next\_state <= 3'b010;

z <=1;

end

default:begin

next\_state = 3'b000;

end

endcase

end

endmodule

