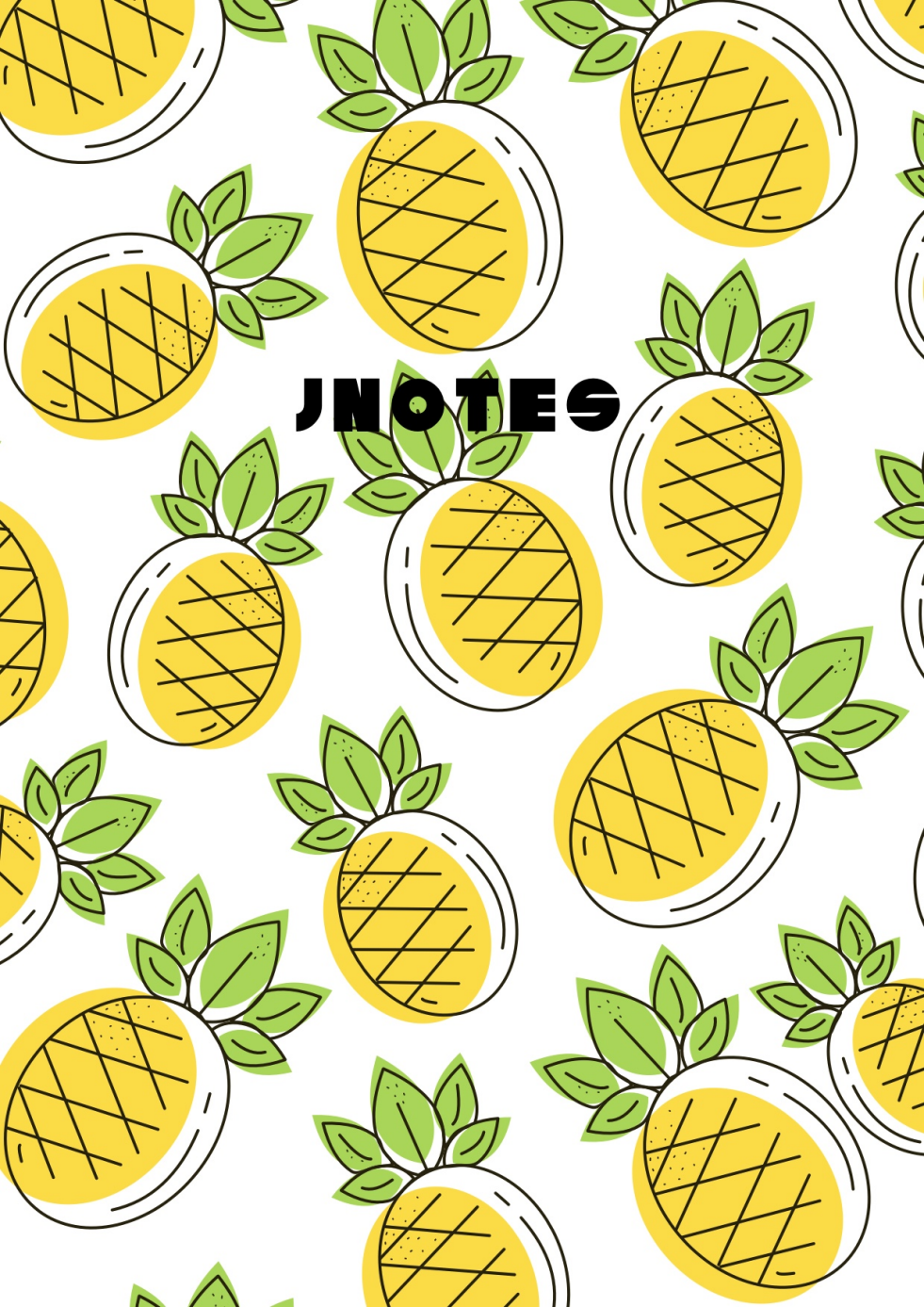

JNOTES

日期: /

Verilog 基础知识

1. 数字表示形式:

① 无符号数: <位宽>'<进制><数值>

Eg: 2'b00 → $(00)_2$

5'd8 → $(01000)_2$

② 有符号数: <位宽>'<sb><数值> signed 有符号

Eg: 8'sb10111011 → $(-69)_2$

01111010

10001011 → -69

有符号数是按补码表示的,即第一位为符号位.

2. if 语句

3. case 语句: case (表达式)

取值1 : 语句1

取值2 : 语句2

default : 默认语句

endcase

4. 三种描述方式:

① 结构化描述(也称门级描述): 全部用门原语,然后模块调用

② 数据流级描述: 全部用 assign 语句

③ 行为级描述: 全部用 always 语句

if, case 等

5. 阻塞,非阻塞

(1) 阻塞赋值语句:

① 格式: 变量 = 表达式

a1 = 4'b00

a2 = 4'b01

> 一个一个赋值,需要排队

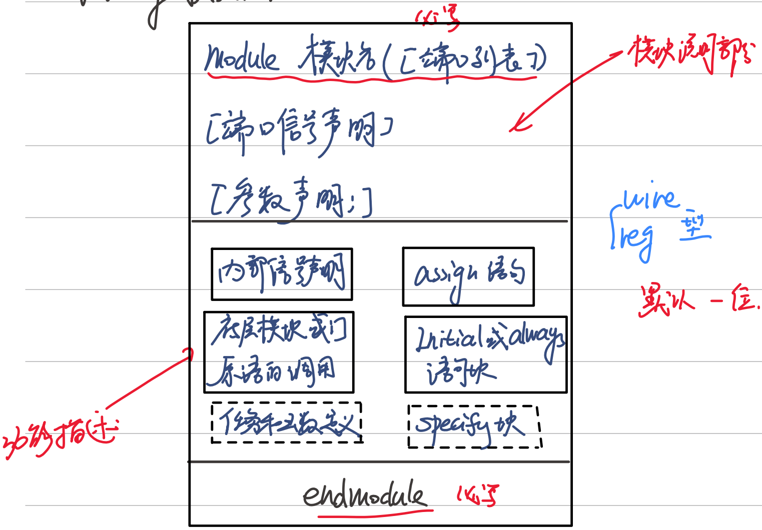
(2) 非阻塞赋值语句

①格式：变量 <= 表达式

$a_1 <= 4'b00$

$a_2 <= 4'b01$ > 同时发生，不用顺序

二、Verilog 模块结构



wire / reg 型

reg[3:0] res; 位宽

默认一位，wire型

二选一 数据选择器：

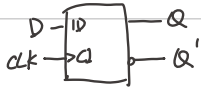
```
module MUX21a(a,b,s,y);
    input a,b,s;
    output y;
    assign y=(s?a:b);
endmodule
```

模块名：MUX21a；端口列表：(a,b,s,y)

input / output > 参数端口声明

assign y=(s?a:b); → assign语句

日期: /



边沿D触发器:

```
module DFF1 ( D, clk, Q);
    input D, clk;
    output Q;
    reg Q;
    always @ (posedge clk)      // clk上升沿 > always语块
        Q <= D;
endmodule
```

2 assign语句: 连续赋值语句

一般格式: assign 赋值目标 = 表达式

四特点: ① 总是处于激活状态. 只要表达式中操作数有变化, 立即进行计算, 赋值

② 赋值目标必须是 wire 型, wire 表示电路间的连线.

拼接 复制运算符:

a1	a0	{a1,a0}
0	0	00
0	1	01
1	0	10
1	1	11

Eg: Y = {4'b01, 2'b11};

Y = 10111

Y = {4{2'b01}}; Y = 01010101

4为1

日期: /

条件运算符.

格式: 表达式1 ? 表达式2 : 表达式3

①原理. 表达式值为0 (结果为表达式3) ②可嵌套

× ×

3. always 语句块 (连和块)

①基本认识:

格式: always @ (敏感时间列表)

多条顺序语句

Eg: always @ (posedge clk) 上升沿.

Q<=D

赋值时必须是reg型

②敏感条件: 边沿敏感, 电平敏感

1. 边沿敏感: (posedge 信号名) 上升沿

(negedge 信号名) 下降沿

2. 电平敏感: (信号列表) 信号列表中任一信号变化.

Eg: (a, b, c) (a or b or c) 2种写法一致

日期: /

assign 和 always 区别：

① assign 总处于激活状态，always 需要触发的词列表。

② assign 赋值目标为 wire，always 赋值目标为 reg

4. 底层模块和门原语调用

(1) 底层模块调用：

端口映射有2种方法。

① 端口名关联法（命名法）

.a(w_a)
.b(w_b)
.c(w_c)

格式：(.底层端口名1(外部信号名1), .底层端口名2(外部信号名2), ...)

② 顺序法。

格式：(外部信号1, 外部信号2, ...)　必须按顺序

(2) 门原语调用：

格式：门原语名　实例名（端口连接）　y, a, b　y = a & b

实例名　输出在前，输入在后

日期: /

<<计数器>> 任意进制模值计数器.

时序逻辑 = 组合逻辑 + 触发器

例1. 用反馈清零法设计一个模55的加法计数器.

模55 → 计数范围 0~54 → 考虑输出=进制位宽

54(10)=进制)是 110110 → 输出[5:0]位宽

```
module count_5 (clk, res y);
    input clk;
    input res;
    output [5:0] y;
    reg [5:0] y;
always @ (posedge clk or negedge res)
  begin
      if(!res)    y <= 6'b0    (6位0)
    else if (y== 6'b110110   y<=6'b0
    else y<=y+6'b1   (+1)
  end
endmodule
```

日期: /

例2：利用参数重定义语句设计一个模5加法计数器

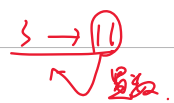
模5 → 计数范围0~4 → 输出位宽[3:0]

```
module cont_5 (clk, res, y);
  input clk;
  input res;
  output y;
  reg [3:0] y;
  always @ (posedge clk or negedge res)
    begin
      if (!res)  y <= 4'b1000;
      else if (y==4'b0)  y<=4'b100;
      else  y <= y-1'b1
    end
endmodule.
```

日期: /

例3：利用阻塞赋值设计一个模9加法计数器

```
module count (clk, rst_n, cnt);
    input clk;
    input rst_n;
    output cnt;
    reg [3:0] cnt;
always @(posedge clk or negedge rst_n)
    begin
        if(!rst_n)  cnt <= 4'b0011
        else if (cnt==4'b1011)  cnt<=4'b0011;
        else  cnt <= cnt + 1'b1
    end
endmodule
```



日期: /

Test bench 仿真激励

测试指：

激励信号产生 → 待测试模块 → 输出响应监视

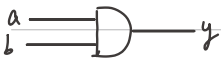
Eg: `timescale 1ns/1ps → 时间单位/时间精度

```
module tb_mux2_1 ( );
  reg a,b, sel;        // 待测试模块输入激励信号，声明为 reg 型
  wire out;            // 待观察信号，声明为 wire 型
                       // 输出
  initial begin
  a=0; b=0; sel=0;
  end
  always #10 a=~a      // 延时控制 (10ns)
  always #20 b=~b
  always #80 sel=~sel
  mux2_1 u1 (a,b,sel,out)
endmodule
```

← 激励信号产生（待测模块输入信号的产生）

日期: /

例1. 实现一个2输入与门. 并搭建仿真环境



源代码:

```
module And(a,b,y);
  input a;
  input b;
  output y;
assign y = a&&b;
endmodule
```

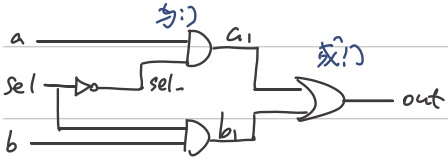
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

仿真环境:(testbench)

```
`timescale 1ns/1ps
module tb_And();
reg a,b;
wire y;
initial
  begin
    {a,b}=00;
#50, {a,b}=01;
#50, {a,b}=10;
#50, {a,b}=11;
  end
And u1(a,b,out)
endmodule
```

日期: /

例2. 搭建一个2选1数据选择器的仿真环境(用门级描述实现)



源代码:

```
module MUX2_1(out, a, b, sel);
input a, b, sel;
output out;
wire a1, b1, sel_, out;   输出在前
not(sel_, sel);           输入在后
and(a1, a, sel_);
and(b1, b, sel);
or(out, a1, b1);
endmodule
```

仿真环境(testbench)

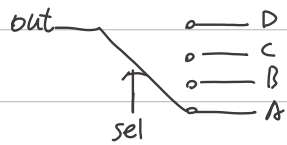
```
`timescale 1ns/1ps
module tb_MUX2_1( );
reg a, b, sel;
wire out;
MUX2_1 U1(out, a, b, sel);
initial
begin
  a=1'b0; b=1'b1; sel=1'b0;
  #5 b=1'b0;
  #5 b=1'b1; sel=1'b1;
  #5 a=1'b1;
end
endmodule
```

a	b	sel	
0	1	0	选a
0	0	0	
0	1	1	选b
1	1	1	

日期: /

例：搭建4选1多according选择器的仿真环境（根据sel信号值的不同而切换输出）

sel	A	B	C	D	out
00	0	x	x	x	0
	1	x	x	x	1
01	x	0	x	x	0
	x	1	x	x	1
10	x	x	0	x	0
	x	x	1	x	1
11	x	x	x	0	0
	x	x	x	1	1



源程序：

```
module Mux4_1(A,B,C,D, sel, out);
input A, B, C, D;
input [1:0] sel;
output out;
reg out;
always @(A,B,C,D)
  begin
    case(sel)
      2'b00 : out = A;
      2'b01 : out = B;
      2'b10 : out = C;
      2'b11 : out = D;
    end
endmodule
```

电平敏感

仿真环境(testbench)

```
`timescale 1ns/1ps
module tb_Mux4_1( );
reg A,B,C,D;
reg [1:0] sel;
wire out;
Mux4_1 U1(A,B,C,D,sel, out);
initial
  begin
    sel = 2'b00;
    {A,B,C,D} = 4'b0xxx;
#10 {A,B,C,D} = 4'b1xxx;
    以此类推 ---
    sel = 2'b01;
  end
endmodule
```

日期: /

case 精确匹配

casez 对z(或?)不关心，匹配

casex 最宽松 x.z(或?)都不关心.

任务与函数 task function

task: 定义时无端口列表. 调用时需给输入.输出列表.

function: 定义时只定义输入端.输出端口为函数名本身.

generate 语句. 通过generate扩展语句块

声明 genvar 类型变量

```
module demo (...);
  - --
  genvar i;
  generate
      for (i=0; i<5; i++) begin: generate_name   // 示例名
        initial begin
           :
        end
      end
  endgenerate
endmodule
```

日期: /

always 语句:

① 用always来描述组合逻辑时，应该用阻塞赋值 (=)

② 时序 非阻塞 (<=)

③ 在同一个always 语句，不要混合使用阻塞非阻塞.

task、function:

① 当没有消耗仿真时间，任务会瞬时仿真时间，

② 当没有可以通过返回值输出，也可定义output类型端口。

任务只可以有output类型

③ 任务可以调用函数，当函数不可以调用任务

日期:　　　/

系统函数

display 可以自动换行

1. $display、$write：将信息打印在屏幕上

(其中 $time：时间单位取整数后，而 $realtime 无这种要求)

int a 10;

%a 显示 0000---1010（32位）

%0a 显示 1010

\n：换行符。 \t：制表符。

\ddd：八进制数ddd对应ASCII字符。

$random：返回一个32位的有符号整型随机数。可以配合取余运算来达到所需范围内

$finish：结束仿真

$readmemb 和 $readmemh

若有一个数据文件，用 $readmemh 将其读到用于存储的一个二维变量中去

$readmemb 操作的数据文件的数字必须是二进制。$readmemh → 十六进制

$fopen、$fclose、$fdisplay、$fwrite。

用于文件的输入、输出

↓ 将格式化内容写入文件。

$fdisplay 会自动换行

而 $fwrite 不会

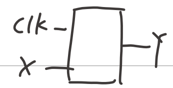
$test$plusargs、$value$plusargs

免除编译，可以传入仿真运行参数以改变代码逻辑。

传递开关参数　　传递值参数。

日期: /

有限状态机(FSM)



1101的序列检测器

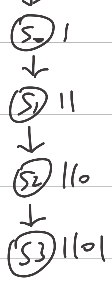
1. Mealy Moore

状态跳转: 均与输入+现态有关

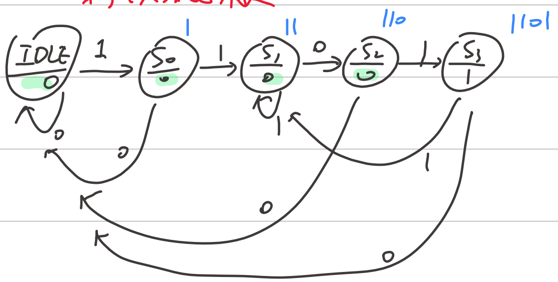
状态输出: Mealy: 与输入+现态有关. Moore: 与现态有关

reset

IDLE: 没有检测到任何与1101有关的输入



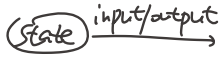
Moore: 输出只与现态有关



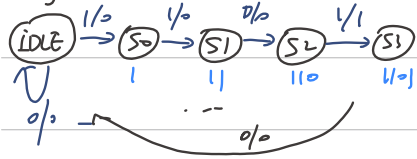
① IDLE 要标记 ② Moore: output

日期: /

Mealy



输出与状态/输入有关



Moore: output 比状态跳转晚一个 cycle. 而 Mealy 不会

3. Verilog 代码实现:

一段式 — 同步模板 always @(po---clk or po---reset)

两段式 — 组合逻辑 always @(*)

三段式 — 输出逻辑

↳ Moore: 时序逻辑.
Mealy: 组合逻辑

case
状态1: ① 转移条件.
② 输出.
状态2 --
:
n - -

Moore: if(reset) begin
output <= 0
end else case ( )
output - -
default - -

Mealy: always (*)
if(reset) begin
output <= 0;
end else begin
case (state)
state A:
if (input==1) output = -;

日期

```
reg state, next_state;

always @(posedge clk or posedge reset)
begin
    if(reset) begin
        state <= IDLE;
    end else begin
        state <= next_state;
    end
end
```

固定模板.

```
always @(*)
begin
    case(state)
        IDLE: begin
            if(input == 1) begin
                next_state = STATE_A
            end else begin
                next_state = IDLE
            end
        end
        B:
        default: next_state =
    endcase
end
```

状态转移

转移条件

此处不变

日期: /

```
always @(posedge clk or posedge reset)
begin
    if (reset) begin
        output_signal <=0;
    end else begin
        case (state)
            STATE_A: output_signal <=0;
            default: output_signal <=0;
    end;
```

输出逻辑

日期: /

时序逻辑:

一. 触发器

1. D触发器:

```
module dff (clk, din, dout)
  input clk, din;
  output dout;
  reg dout;
always @ (posedge clk) begin
    dout <= din;
end
endmodule
```

2. 带异步复位端D触发器:

```
module dff_rst (clk, rst_n, din, dout)
  input clk, din, rst_n;
  output reg dout;
always @ (posedge clk) begin
  if (!rst_n) begin
      dout <= din;
  end else begin
      dout <= 0;
  end
endmodule
```

日期: /

3. 移位寄存器:

```
module shifter(clk, rst_n, load_enable, load_data, dout);
input clk, rst_n, load_enable;
input reg[7:0] load_data;
output [7:0] dout;
reg [7:0] shift_data;
always @(posedge clk) begin
  if(!rst_n)
      shift_data <= 1'b0;
  else begin
    if(load_enable)
      shift_data <= load_data;
    else
      shift_data <= {shift_data[6:0], shift_data[7]};
    end
end
  assign dout = shift_data;
endmodule
```

日期:　　/

基础知识:

CPLD: Complex Programmable Logic Device.

复杂可编程逻辑器件.　　乘积项(PT)

规模较小,逻辑复杂度低,适用于逻辑控制.

FPGA: Field Programmable Gate Array

现场可编程门阵列　　查找表(LUT)

规模大,逻辑复杂度高,适用于完成复杂算法.

FPGA优点: 并行结构. 低延时. 可重构. 开发灵活. 扩展性高

比较: VS. 单片机(或DSP): FPGA的价格更高,但可以覆盖单片机(侧重于系统控制)和DSP(为高速数据处理定制)无法满足的场景. 现在FPGA上集成ARM和DSP越来越普遍. 进行运算的调度.

VS. GPU: GPU的计算性能通常优于FPGA,但FPGA灵活度更高,功耗低于GPU. 在小分和新算法上, FPGA具有优势.

VS. ASIC(专用集成电路): FPGA提供了高灵活性和较短的上市时间. 在小批量生产时, FPGA较于ASIC有成本优势,因为ASIC有高昂的NRE(一次性工程成本)

日期: /

EDA:

设计方法:

① Top-down 设计: 从系统设计入手, 自顶向下进行功能划分和设计.

② IP核复用: IP (软核, 固核, 硬核)

SoC (片上系统) System on chip.

数字设计流程:

设计输入 → 功能仿真 → 综合 → 布局布线 → 时序仿真 → 编程到配置

将逻辑网表映射到具体可用器件中

1. 原理图输入
2. 硬件描述语言 (HDL)

发现错误, 设计修改直至正确为止

综合: 高层次 (HDL代码) → 低层次描述 (门级网表)

将器件互连关系以文件方式载到可编程器件中

加入实际的延迟信息, 对电路的时序性能进行验证

日期: /

Moore 输出只与当前状态有关 → 同步输出

Mealy 还与输入有关 → 异步输出

Mealy 比 Moore 输出要比输入领先一个时钟周期

ASCII: 0-31: 控制字符.

32-126: 可打印字符. space: 32

数字'0'→'9' 48-57

大写英文字母'A'→'Z' 65-90

小写英文字母'a'→'z' 97-122

JK触发器: $Q_{n+1} = J\overline{Q_n} + \overline{K}Q_n$.

RS: $Q_{n+1} = S + \overline{R}Q_n$

可综合的语句:

always. begin-end. assign/过程赋值(=, <=). if-else/case. for.

`define. `ifdef `else. `endif

日期: /

常见的编码方式为需要多少个状态寄存器位.

1. 顺序编码: N个状态. 需 $\log_2 N$ 位. 2→1, 5→3

2. 格雷码 (特殊的顺序编码) 需 $\log_2 N$ 位.

3. 约翰逊编码 (Johnson. 环形编码) N位.

4. 独立热码/一位热码 N位

二进制计数器:

N位 = 进制计数器 $0 \sim 2^N - 1$ 4位: $0 \sim 2^4 - 1$ 即 0~15

十进制计数器:

每计数到 9 $\xrightarrow{\text{重置到}}$ 0

function 不可包含时序控制 (initial 延迟事件 # 或@)

1次仿真时间内完成执行 但是task可以包含时序控制