Campus | B5 7mm×30行 40页

Digital System Design

Campus®

B5 点线本 | 7mm×30行 | 40页

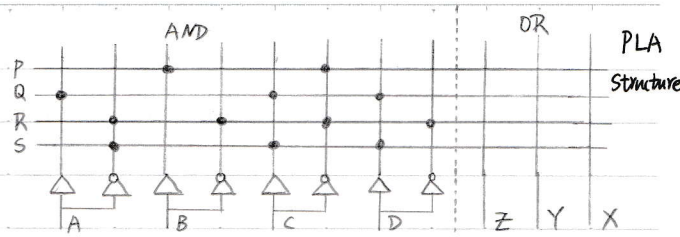
KOKUYO

No.

Date

Introduction

IC : Integrated circuit
ASIC : Application-specific integrated circuit
PLDs : Programmable logic devices
PLAs : Programmable logic arrays



Design flow

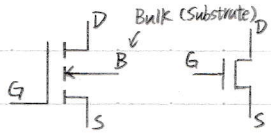
① Write a specification
② Partition the design into smaller parts and write a specification for each part (if necessary)
③ From the specification → draw state machine chart (each state, input conditions, change of state, output)
④ Minimize the number of states
⑤ Assign Boolean variables to represent each state
⑥ Derive the next state and output logic
⑦ Optimize the next state and output logic to minimize the number of gates needed.
⑧ Choose a suitable placement for the gates in terms of which gates share integrated circuits and in terms of where each integrated circuit is placed on the printed circuit board.
⑨ Design the routing between the integrated circuits.

EDA : Electronic design automation
CAD : Computer aided design
RTL : Register transfer level

Dominant technology : ↙ uses FETs
CMOS : Complementary metal oxide semiconductor

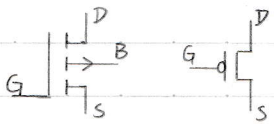
In digital

NMOS



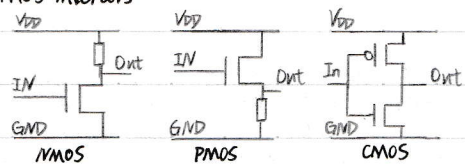
The bulk is always connected to the GND (logic 0)

PMOS

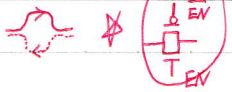


The bulk is always connected to the positive power supply (logic 1)

CMOS inverters

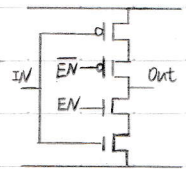


Transmission Gate

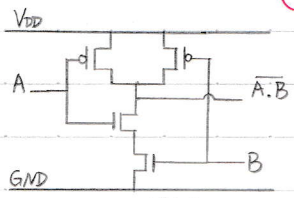


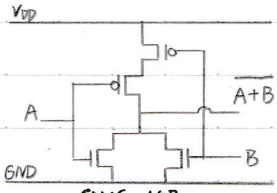
不用代控制开关是因为它需要衬底供电(供电或接地)

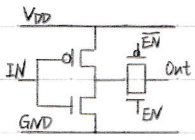
Three state buffer



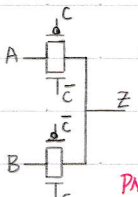
EN=1
Normally
EN=0
Neither of the two inner transistor is conducting and the output floats



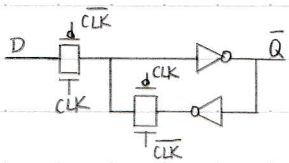


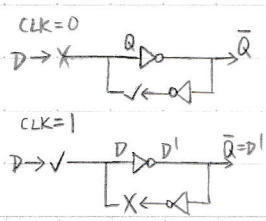


EN=1
Both transistors conduct
EN=0
Both transistors are open circuit



Multiplexer





D-latch

PMOS passes a strong '1' but poor '0'
NMOS passes a strong '0' but poor '1'

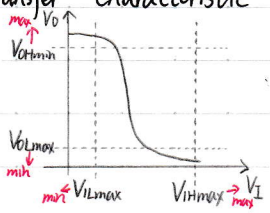
No.

Date

Transfer characteristic of CMOS inverter

输入逻辑0的最大值，使输出跳变，一般大于输出逻辑值0

输出逻辑1的最小值，一般大于输入逻辑1的电压



$$\begin{cases} NM_L = V_{ILmax} - V_{OLmax} \\ NM_H = V_{OHmin} - V_{IHmin} \end{cases}$$

The noise margin specifies how much noise, from electrical interface, can be added to a signal before a logic value is misinterpreted.

E.g: The maximum voltage that a gate will generate to represent a logic '0' is 0.75V
Any voltage up to 1.05V is, however, recognized as a logic '0'
Therefore there is a 'spare' 0.3V, any noise added to a logic 0 within this band will be accepted

Fan-out

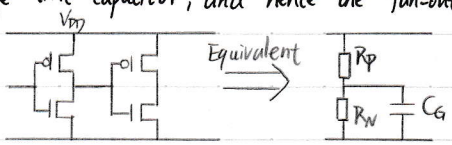
The fan-out of a gate is the number of other gates that it can drive

[TTL] the input to a gate is resistive e.g: 74ALS

$$\begin{cases} \text{Logic 1 fan-out} = \dfrac{I_{OHmax}}{I_{IHmax}} = \dfrac{400\mu A}{20\mu A} = 20 \\ \text{Logic 0 fan-out} = \dfrac{I_{OLmax}}{I_{ILmax}} = \dfrac{8mA}{100\mu A} = 80 \end{cases} \Rightarrow \text{choose the smaller value}$$

[CMOS] no dc input current while gate and substrate form a capacitor

CMOS gates draw almost no DC input current because there is no DC path between the gate of a transistor and the drain, source or substrate of the transistor.
Different effect: the gate and substrate of a CMOS gate ~~form~~ form a capacitor, it takes a finite time to change that capacitor, and hence the fan-out is determined by how fast the circuit is required to switch



$$V_O = V_{DD} e^{-\frac{t}{R_N C_G}} \quad (\text{Assume } R_N \gg R_P)$$

$t = 2.5ns$ (If $V_{DD} = 2.5V$, $R_N = 100\Omega$, $C_G = 100\mu F$)

If two inverters are driven, the capacitive load doubles, so the switching time doubles.

Although a CMOS gate can drive an almost unlimited number of other gates at a fixed logical level, the fan-out is limited by the speed required of the circuit.

- Xilinx.com
- Actel.com

Combinational logic design

Rules of Boolean algebra

① Commutativity $A+B=B+A$ $A\cdot B=B\cdot A$

② Associativity $A+(B+C)=(A+B)+C$ $A\cdot(B\cdot C)=(A\cdot B)\cdot C$

③ Distributivity $A\cdot(B+C)=A\cdot B+A\cdot C$

De Morgan's law

$\overline{(A\cdot B)}=\bar{A}+\bar{B}$ $\overline{(A+B)}=\bar{A}\cdot\bar{B}$

Shannon's expansion theorem (将任意布尔函数表达为：变中任何一个变量乘以一个子函数，加上这个变量乘以另一个子函数)

$F(A,B,C,D,\dots)=A\cdot F(1,B,C,D\dots)+\bar{A}\cdot F(0,B,C,D,\dots)$

$=(A+F(0,B,C,D,\dots))\cdot(\bar{A}+F(1,B,C,D,\dots))$

$F=AB+C=A\cdot(B+C)+\bar{A}\cdot C$

$=AB+AC+\bar{A}C=AB+C$

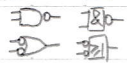
Logic gates

AND:  

OR: 

NOT: 

XOR:  

NAND: 

NOR: 

A	B	C	Z	
0	0	0	1	m_0
0	0	1	1	m_1
0	1	0	0	M_2
0	1	1	0	M_3
1	0	0	0	M_4
1	0	1	1	m_5
1	1	0	0	M_6
1	1	1	1	m_7

A minterm is a Boolean AND function containing one instance (function is true)
A maxterm is a Boolean OR function with one instance (function is false)

$\Rightarrow Z=m_0+m_1+m_5+m_7=M_2+M_3+M_4+M_6$

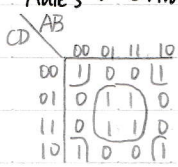
$=\bar{A}\bar{B}\bar{C}+\bar{A}\bar{B}C+A\bar{B}C+ABC=$ ~~$A\bar{B}C+A\bar{B}\bar{C}+\bar{A}\bar{B}C+\bar{A}\bar{B}C$~~

$=(A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+B+C)(\bar{A}+\bar{B}+C)$

Logic minimization

[Karnaugh maps]

Rules: ① Circle the largest possible groups $(1,2,4,\dots,2^n)$
② Avoid circles inside circles



$\Rightarrow Z=B\cdot D+\bar{B}\cdot\bar{D}$

Karnaugh maps is a graphical method, effectively with six or fewer V

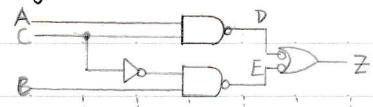
No.

Date

Timing

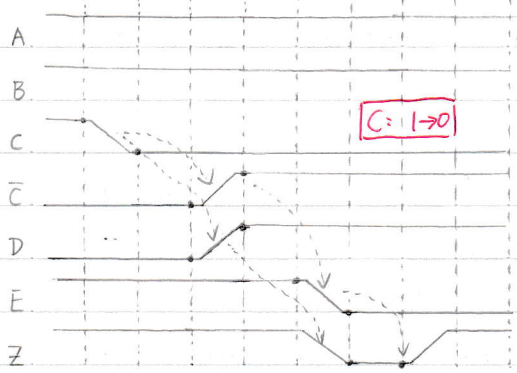
All gates have a finite delay between a change at an input and the change at an input. If gates are used, therefore, different paths may exist in the network, with different types.

E.g: $Z = A.C + B.\bar{C}$



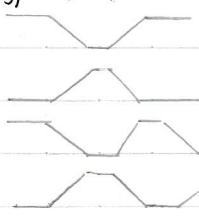
Let A=1 and B=1

⇒ Z=1, irrespective of the value of C



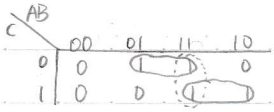
← This Z changes from 1 to 0 and back to 1, known as hazard

Types of hazard



	Occur in	
Static 1	AND-OR	NAND-NAND
Static 0	OR-AND	NOR-NOR
Dynamic 0	3 or more unequal signal paths	
Dynamic 1		

Static hazards can be avoided by designing with redundant logic



⇒ $Z = A.C + A.B + B.\bar{C}$

$A.B$: {A=1, B=1} ⇒ Z=1 ← hold the value, no glitch

Number codes

[Integers]

Two's complement: $-b : 2^n - b$ (b is a binary number; represented using n bits)

E.g: -6_{10}: $10000_2 - 0110_2 = 1010_2$

The same result is obtained by inverting all bits and adding 1

$6_{10} \Rightarrow 0110_2 \Rightarrow 1001_2 \Rightarrow 1001_2 + 1_2 \Rightarrow 1010_2$ ✓

[Floating point numbers] ← allows a much wider range of accuracy

$(-1)^S + 1.m \times 2^e$ (↑ 二进制)

Typically 32 bits: 1 ⇒ sign bit (S) (符号位) 8 ⇒ exponent (e) (阶码) 23 ⇒ mantissa (m) (小数部分 尾数)

E.g: 123.456 ①二进制→ $1111011.01110100101111001_2$ (24位) ②右移 (右移了6位) → $1.11101101110100101111001_2$

阶码要用移码表示 (符号位相反) 6_{10} 移码→ 1000101_2 (可以直接 ? + 127 转二进制)

(符号位取反的补码)

用补码表示阶码时，当阶码无穷小，产生下溢的时候，阶码变成了0，浮点数的值变为1

而实际上这个数字无限接近于0（~~六~~）需要取出"-0"作为机器零

浮点数的阶码表示指数大小，有正有负，对每个阶码都加上一个正的常数（称偏移常数），使能表示的所有阶码都为整数

就变成了偏移了的阶码 也就是移码，浮点数小数点的实际位置由移码减去偏移常数决定

[Gray codes]

The transition from $0111_2(7)$ to $1000_2(8)$ causes three bits to change, it may be undesirable that several bits should change at once as bits change may not occur at exactly the same time.

000
001
011
010
110
111
101
100

A gray code is one in which only one bit changes at a time.

[Parity bits] 校验位

Use a parity bit to tell that whether there are an even number of ones in the data.

A single error can be corrected by using a two-dimensional parity scheme in which every ninth word is itself a set of parity bits. (row parity + column parity)

No.

Date

Combinational logic using VHDL gate models

Combinational logic is stateless : changes in inputs are immediately reflected by changes in inputs. VHDL : Very-high-speed Hardware Description Language.

Entities and architectures

```
entity And2 is                                  ← Describe a block box, inputs and output with types
    port (x, y : in BIT; z : out BIT);          ← Specify connections between entity and the outside world
end entity And2;                                                                  { in/out
                                                                                  { inout ← Bidirectional

architecture ex1 of And2 is                     ← Describe function and/or structure
begin
    z <= x and y;
end architecture ex1;
```

Identifiers, spaces and comments

VHDL is not case-sensitive, VHDL description : '--'

Each VHDL should include a header :

```
-- Design unit :
-- File name :
-- Description :
-- Limitation :
-- System    : VHDL'93
-- Author    : Klein Kwang

-- Revision  : Version 1.0
```

Netlists

For function : $Z = \bar{A}.B + A.C$

```
entity comb_function is
    port (a, b, c : in BIT; z : out BIT);
end entity comb_function;

architecture expression of comb_function is
begin
    z <= (not a and b) or (a and c);
end architecture expression;
```

```
architecture netlist2 of comb_function is
    component And2 is
        port (x, y : in BIT; z : out BIT);
    end component And2;
    component Or2 is
        port (x, y : in BIT; z : out BIT);
    end component Or2;  ...
    signal p, q, r : BIT;
begin
    g1 : Not-1 port map (a, p);
    g2 : And-2 port map (p, b, q); ...
end architecture netlist2;
```

Component Declaration

[Manual]

```
entity Or2 is
    port (x, y : in BIT; z : out BIT);
end entity Or2;

entity Not1 is
    port (x : in BIT; z : out BIT);
end entity Not1;
```

```
architecture exp1 of Or2 is
begin
    z <= x or y;
end architecture exp1;
architecture exp1 of Not1 is
begin
    z <= not x;
end architecture exp1;
```

Identify signals to connect gates →

Create instance →

```
architecture netlist of comb_function is
    signal p, q, r : BIT;
begin                                   ↗ refers to the current working library
    g1 : entity WORK.Not1 (exp1) port map (a, p);
    g2 : entity WORK.And2 (exp1) port map (p, b, q);
    g3 : entity WORK.And2 (exp1) port map (a, c, r);
    g4 : entity WORK.Or2 (exp1) port map (q, r, z);
end architecture netlist;
```

↳ refers, in each case, to the architecture of each gate model

The order of signal is important → We can assign signal in different order :

```
g2 : entity WORK.And2 (ex1) port map (z => q, x => p, y => b);
```

Direct Instantiation Style — usually preferred for simple netlists.

Signal assignments

Inertial delay: z <= x after 4 ns (默认的延时 (不会将传输/保持时间低于4ns的pulse))

Transport delay: z <= transport x after 4 ns (用于模拟连线延迟)

z <= x and y after 5 ns ✓

z <= transport not ((x and y) or (a and b)) after 8 ns ✓

Multiple changes are allowed in response to a single change:

z <= x after 4 ns, not x after 8 ns;

↑ 改整个时间轴

Generics

z <= x and y after 5 ns; defines exactly delay for an AND gate. We could define different delay using parameter

```
entity And2 is
  generic (delay: DELAY_LENGTH);
  port (x,y: in BIT; z: out BIT);
end entity And2;

architecture ex2 of And2 is
begin
  z <= x and y after delay;
end architecture ex2;
```

a value is passed →

```
g2: entity WORK.And2(ex2) generic map (5ns)
                          port map (p,b,q);
```

A component declaration must also include the generic declaration
The nonpositional form can also be used.

It can be useful to specify a default value for a generic

generic (delay: DELAY_LENGTH := 5 ns); ← A different value can override the default if it is explicitly stated.

If a compnent declaration is used, the default value can be specified in the entity declaration or the component declaration or in both. If different default values are specified in each declaration, the value in the component declaration will be used.

If no default value is given and the generic map part is omitted, delay would be undefined X

If the component declaration does not include the generic part definition, the default value will automatically be used and the generic map part of the component instantiation must be omitted

Finally, the default value of a generic will be used if the reserved word open is used:

```
g2: entity WORK.And2(ex2) generic map (open)
                          port map (p,b,q)
```

No.

Date

Constant and open ports

There may be occasions on which not all the inputs or outputs of a component are needed.

E.g 'universal' gate : invert=0 → AND , invert=1 → OR

```
entity universal is
    port (x, y, invert : in BIT ; a, o : out BIT);
end entity universal ;

architecture univ of universal is
begin
    a <= (y and (x xor invert) or (invert and not y);
    o <= (not x and (y xor invert)) or (x and not invert);
end architecture univ;
```

Use this gate as an AND gate :

```
u0 : entity WORK.universal (univ) port map (x, y, '0', a, open);
```

Outputs can be left ~~connection~~ connected but inputs may be left open only if a default value has been specified.

Test benches

```
entity TestAnd2 is
end entity TestAnd2 ;
```

```
architecture io of TestAnd2 is
    signal a, b, c : BIT ;
begin
    g1 : entity WORK.AND2(ex2) port map (x=> , y=> , z=>c);
        a <= '0', '1' after 100 ns;
        b <= '0', '1' after 150 ns;
end architecture io ;
```

Configuration

The architecture name was explicitly stated, but it can be omitted if an entity has only one archit

```
g1 : entity WORK.Not1(ex1) port map (p,q);
g1 : entity WORK.Not1 port map (p,q);
```

Suppose that we wish to have more control over exactly which architecture to use. With direct instantiation, there is no difficulty.

With alternative style, however, there needs to be an explicit statement – the configuration specification

```
architecture alternate of TestAnd2 is
    component A2 is
        port (x, y : BIT ; z : out BIT) ;
    end component A2;
    for all : A2 use entity WORK.And2(ex2);
    signal a,b,c : BIT ;
begin
    g1 : A2 port map ( x=>a, y=> b , z=>c) ;
end architecture alternate ;
```

For complex models, with several levels of hierarchy, it is often more appropriate to use a configuration unit.

```
configuration Tester1 of TestAnd2 is
  for io
    for g1 : And2
      use entity WORK.And2(ex1);
    end for;
  end for;
end configuration Tester1;
```

A configuration declaration for the original testbench

The complete model consists of:
① entity and architecture of the And2 gate
② entity and architecture of the testbench
③ configuration

↑ This style requires one configuration for the entire design

Another way to write configuration: (board-socket-chip 基座 芯片 analogy) ← configuration is used to map between arbitrary internal and external names

```
architecture remapped of TestAnd2 is
  component MyAnd2 is
    generic (dly : DELAY_LENGTH);
    port (in1, in2 : BIT; out1 : BIT);
  end component MyAnd2;
  signal a, b, c : BIT;
begin
  g1 : MyAnd2 generic map (6 ns) port map (a, b, c);
end architecture remapped;

configuration Tester2 of TestAnd2 is
  for remapped
    for g1 : MyAnd2
      use entity WORK.And2(ex2)
        generic map (delay => dly);
        port map (x => in1, y => in2, z => out1);
    end for;
  end for;
end configuration Tester2;
```

A different style of configuration has one configuration for each per entity

```
configuration And2Con of And2 is
  for ex1
  end for;
end configuration And2Con;
```

↑ This selects the architecture for an entity

For TestBench ⇒

```
configuration Tester3 of TestAnd2 is
  for remapped
    for g1 : MyAnd2
      use configuration And2Con of WORK.And2Con;
    end for;
  end for;
end configuration Tester3;
```

This approach requires a great number of configuration units, but each unit is simplier.

No.

Date

Combinational building blocks

Three state buffers 高阻抗状态相当于隔断状态，对下一级无任何影响

[Multi-valued logic]

Use switches to disconnect a signal from a wire, as being in a 'high-impedance' state.

Type BIT is no longer adequate to represent logical signal value {'0' '1'

```
signal a, b, c : tri ;  -- tri is a new type
```

We need an AND operator described by the truth table

AND	0	1	Z
0	0	0	0
1	0	1	1
Z	0	1	1

In VHDL, functions and operators can be overloaded

Rewrite a new AND operator to take two operands of type tri

```
function "and" (Left, Right : tri) return tri is
  type tri_array is array (tri, tri) of tri ;
  constant and_table : tri_array := (('0', '0', '0'),
                                     ('0', '1', 'Z'),
                                     ('0', 'Z', 'Z'));
begin
  return and_table (Left, Right);
end function "and";
```

Fail if input BIT and TRI, as the operator has not been defined.

[Standard logic package]

'U'	Uninitialized
'X'	Forcing unknown
'0'	Forcing 0
'1'	Forcing 1
'Z'	High impedence
'W'	Weak unknown
'L'	Weak '0'
'H'	Weak '1'
'-'	Don't care

The standard logic type is defined by:

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

If we write a model using signals of type BIT and/or std_ulogic, we must ensure that two models do not attempt to put a value onto the same signal.

std_logic allows more than one output to be connected to the same signal

```
subtype std_logic is resolved std_ulogic ;
```

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

```
library IEEE;
use IEEE.std_logic_1164.all;
```

No.

Date

[When ... else statement]

```
library IEEE;
use IEEE.std_logic_1164.all;

entity three_state is
    port (a, enable : in std_logic;
          z : out std_logic);
end entity three_state;
```

```
architecture when_else of three_state is
begin
    z <= a when enable = '1' else 'z';
end architecture when_else;
```

→ `z <= a after 4 ns when enable = '1' else 'z';`

Decoders

[2 to 4 decoder]

Vectors can be described using an array of Boolean signals:

```
type std_logic_vector is array (NATURAL range <>) of std_logic;
```

(NATURAL: Predefined subtype with integer values from 0 to max; range <>: Undefined range)

```
entity decoder is
    port (a: in std_logic_vector (1 downto 0);
          z: out std_logic_vector (3 downto 0));
end entity decoder;

architecture when_else of decoder is
begin
    z <= "0001" when a = "00" else
         "0010" when a = "01" else
         "0100" when a = "10" else
         "1000" when a = "11" else
         "XXXX"; -- Can be omitted
end architecture when_else;
```

unaffected; z; 置換

[With ... select statement]

```
architecture with_select of decoder is
begin
    with a select
        z <= "0001" when "00",
             "0010" when "01",
             "0100" when "10",
             "1000" when "11",
             "XXXX" when others;
end architecture with_select;
```

If ~~one~~ more than one pattern should give the same output, the pattern can be listed!

eg.

```
...
"0000000" when "1010" | "1011" | "1100",
...
```

No.

Date

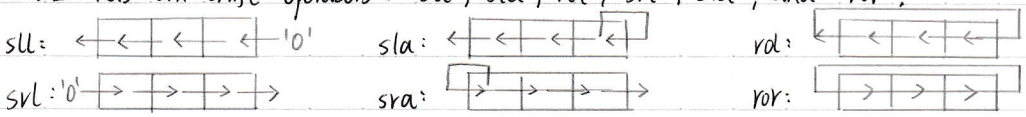
[n to 2^n decoder - shift operators]

```
library IEEE;
use IEEE.std_logic_1164.all;

entity decoder is
    generic (n: positive);          --POSITIVE is a predefined subtype of INTEGER (1 to MAX)
    port (a: in std_logic_vector (n-1 downto 0);
          z: out std_logic_vector (2**n-1 downto 0));     --"**" is the power operator
end entity decoder;
```

As the output value is always "00...01" rotated left by the number of places given by a, we can declare a vector of length 2^n with all bits set to '0' other than bit '0' with a constant declaration:

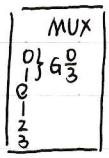
```
constant z_out : std_logic_vector (2**n-1 downto 0) := (0=> '1', others => '0');
```

VHDL has six shift operators: sll, sla, rol, srl, sra, and ror.

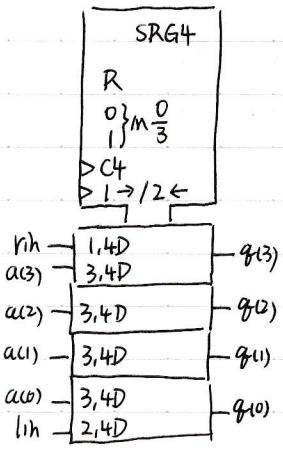


G means select

1 is signal name



0/3 means 0-3



M: mode control (0 to 3)

Mode 1 : shift right

Mode 2 : shift left

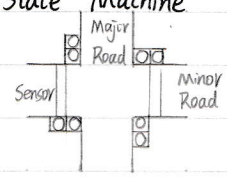
1,4D - in mode 1 D type behaviour when control signal 4 is asserted

Engineer wants it to work, but scientist wants it to be the best

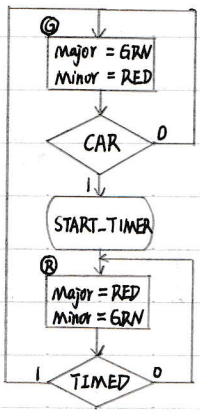
No.

Date

State Machine

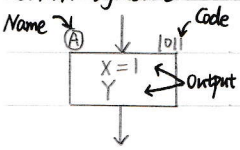


Two states:
- Major Road Red, Minor Road Green
- Major Road Green, Minor Road Red



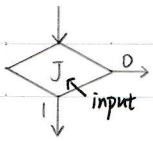
ASM Chart (Algorithm state machine)

[ASM Symbol]



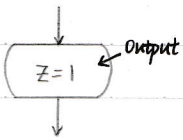
State

※ Output asserted for ONE CLACK CYCLE



Decision

Evaluated during cycle of state



Conditional output

Follows decision

Output asserted for one cycle

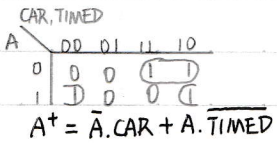
[Synthesis from ASM]

Present State	CAR, TIMED 00	01	11	10
G	G,0	G,0	R,1	R,1
R	R,0	G,0	G,0	R,0

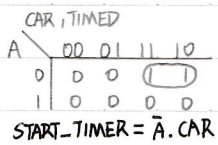
Next state, START_TIMER

State Assignment

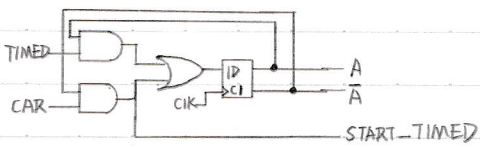
A	CAR, TIMED 00	01	11	10
0	0,0	0,0	1,1	1,1
1	1,0	0,0	0,0	1,0



$A^+ = \bar{A}.CAR + A.\overline{TIMED}$



START_TIMER = $\bar{A}$.CAR



SystemVerilog

initial → execute once and stop

[Adder]

Dataflow SystemVerilog

```
module full_adder (output logic sum, co,
                   input logic a, b, ci);

always_comb
begin
  sum = a^b^ci;
  co  = (a&b) | (a&ci) | (b&ci);
end

end_module
```

Structural SystemVerilog

```
module full_adder (output logic sum, co,
                   input logic a, b, ci);

logic i, j, k, l;

xor g0 (i, a, b);
xor g1 (sum, i, ci);
and g2 (j, a, b);
and g3 (k, a, ci);
and g4 (l, b, ci);
or  g5 (co, j, k, l);

endmodule
```

No.

Date

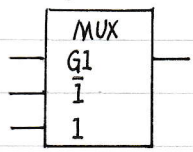
ModelSim : force clk 0 0, 1 10 -r 20
set clk 0

Arrays : logic [3:1] carry ; A vector value can be written as an integer, 4'b0101, '0 (repeated value)

[4 bit adder]

```
module four_bit_adder (output logic [3:0] sum, output logic co,
                       input logic [3:0] a, b, input logic ci);
logic [3:1] carry ;

full_adder fa0 ( sum[0], carry[1], a[0], b[0], ci );
full_adder fa1 ( sum[1], carry[2], a[1], b[1], carry[1] );
full_adder fa2 ( sum[2], carry[3], a[2], b[2], carry[2]);
full_adder fa3 (sum[3], co, a[3], b[3], carry[3] );

endmodule
```

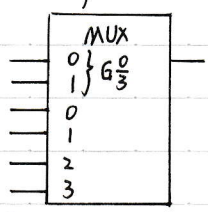
[Multiplexer]



G: Select

```
module mux2 (output logic y,
             input logic a, b, s);
always_comb
  if (s)
    y=b;
  else
    y=a;
endmodule
```

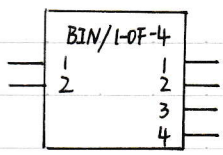
[4 input MUX]



0/3 : 0-3

```
module mux (input logic a, b, c, d, s1, s0
            output logic y );
always_comb
  if(s0)
    if(s1)
      y=d
    else
      y=c;
  else
    if (s1)
      y=b;
    else
      y=a;
endmodule
```

[2-4 decoder]




```
module decoder (input logic [1:0] a,
                output logic [3:0] y);
always_comb
  case (a)
    0: y=1;
    1: y=2;
    2: y=4;
    3: y=8;
    default: y='x ;
  endcase
endmodule
```

[Testbench] { tools → options
assignment → setting → simulation

```
module test_decoder;
logic [1:0] a ;
logic [3:0] y ;
decoder dd (.*); // only works if internal signals have the same names

initial
begin
  #10ns a=0;
  #10ns a=1;
  #10ns a=2;
  #10ns a=3;
end
endmodule
```

```
initial
begin
  clock=10;
  forever #5ns clock=~clock;
end

always
begin
  #5ns clock='0;
  #5ns clock='1;
end
```

SystemVerilog parameters

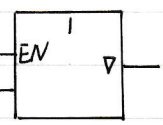
```
module decoder2
  #(parameter N=3)
  (input logic [N-1:0] a,
   output logic [(1<<N)-1:0] y);

  always_comb
    y='b1 <<a ;
endmodule
```

could add delay → assign y=(s==0)?a:1'bz; assign y=(s==1)?b:1'bz; assign y=(s==2)?c:1'bz; assign y=(s==3)?d:1'bz; endmodule

If s=2'b0x, ?: evaluates both branches in case of x
Thus ↳ return 1'b x [Three-State Buffer]

from the tristate mux

```
module three_state_mux
  (output wire y,
   input logic a,b,c,d,
   input logic [1:0] s);
```




```
module three_state (input logic a, enable
                    output logic y);
always_comb
  y = enable ? a : 1'bz;
end module
```

[logic vs Wire]

logic is a variable, but no signal strength or signal resolution

If two signals drive the same physical hardware, use "wire" ← wire has 16 strengths (e.g. strong1, weak1, highz1 ...)

The only occasion when it is permissible to connect outputs of two or more gates together

case → parallel hardware

unique will try to not make it parallel

No.

Date

unique checks for overlapping branches

```
always_comb
begin
    valid = 1'b1;  // default value
    unique casez (a) // treat z or ? as don't care
        4'b1???: y = 2'b11;
        4'b01??: y = 2'b10;
        4'b001?: y = 2'b01;
        4'b0001: y = 2'b00;
        4'b0000: begin
                    y = 2'b00;
                    valid = 1'b0;
                 end
    endcase
end
```

This will check whether the case is unique

`===` (or `!==`) treats x or z as don't care

Unknows `==` (or `!=`) operator can return 0, 1, or X

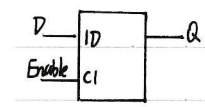
[Priority Encoder]

Input				Output		
A_3	A_2	A_1	A_0	Y_1	Y_0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	–	0	1	1
0	1	–	–	1	0	1
1	–	–	–	1	1	1

```
module encoder (output logic [1:0] y,
                output logic valid,
                input logic [3:0] a);

endmodule
```

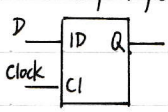
Sequential Building Blocks

[D-latch]




```
module dlatch (output logic q,
               input logic d, en);
always_latch  ← indicate a latch
    if (en)
        q <= d;  ← use "<=" for sequential logic
endmodule
```

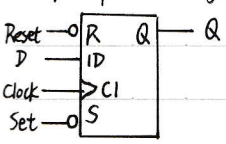
[D-Flip-flop]




```
module dff (output logic q,
            input logic d, clk);
always_ff @ (posedge clk)  ← indicate a flip-flop
    q <= d;                ↖ activation list, sensitive to clock edge
endmodule
```

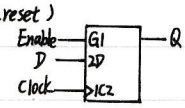
No dependency on clock

[Flip-flop with asynchronous set/reset]




```
module dffr (output logic q,
             input logic d, clk, n_reset);
always_ff @ (posedge clk, negedge n_reset)
    if (!n_reset)
        q <= 1'b0;
    else
        q <= d;
endmodule
```

[Flip-flop with clock enable]




```
module dffe (output logic q,
             input logic d, clk, enable);
always_ff @ (posedge clk)
    if (enable)
        q <= d;
endmodule
```

[n-bit Register]

```
module dffn #(parameter N=8)
             (output logic [N-1:0] q,
              input logic [N-1:0] d,
              input clk, n_reset, enable);
always_ff @ (posedge clk, negedge n_reset)
    if (!n_reset)
        q <= 10;
    else
        q <= d;
endmodule
```

[Universal Shift Register]

Mode Control → S_1	S_0	Action
0	0	Hold
0	1	Shift Right ← lin
1	0	Shift Left rin →
1	1	Parallel Load

```
modul use #(parameter N=8)
           (output logic [N-1:0] q,
            input logic lin, rin, clk, n_reset,
            input logic [N-1:0] a,
            input logic [1:0] s);
always_ff @ (posedge clk, negedge n_reset)
    if (!n_reset)
        q <= 10;
    else                          { } 合并位 operator
        case (s)
            2'b01: q <= a;
            2'b10: q <= {q[N-2:0], lin};
            2'b01: q <= {rin, q[N-1:1]};
            default;
        endcase
endmodule
```

asynchronous reset

[Binary Counter]

```
module counter #(parameter = 8)
                (output logic [N-1:0] q,
                 input logic clk, n_reset);
always_ff @ (posedge clk, negedge n_reset)
    if (!n_reset)
        q <= 10;
    else
        q <= q+1;
endmodule
```

and

3:4D → load data

mode 3

clock

No.

Date

[SystemVerilog Model of State Machine]

§ Version 1

```
enum {G,R} present_state, next_state;
always_ff @(posedge clock, negedge n_reset)
begin: SEQ
   if (!n_reset)
      present_state <= G;
   else
      present_state <= next_state;
end

always_comb
begin: COM
   start_timer = '0;
   minor_green = '0;
   major_green = '0;
   next_state = present_state;
case (present_state)
   G: begin
      major_green = '1';
      if (car)
      begin
         start_timer = '1;
         next_state = R;
      end
      end

   R: begin
      minor_green = '1;
      if (timed)
         next_state = G;
      end
endcase
```

```
module traffic (output logic start_timer, major_green, minor_green,
                input logic clock, n_reset, timed, car);
```

§ Version 2

```
enum {G,R} state;
always_ff @(posedge clock,
            negedge n_reset)
begin: SEQ
   if (!n_reset)
      state <= G;
   else
      case (state)
         G: if (car)
               state <= R;
         R: if (timed)
               state <= G;
      endcase
   end

always_comb
begin: OP
   start_timer = '0;
   minor_green = '0;
   major_green = '0;
   case (state)
      G: begin
         major_green = '1;
         if (car)
            start_timer = '1;
         end
      R:
         minor_green = '1;
   endcase
end
endmodule
```

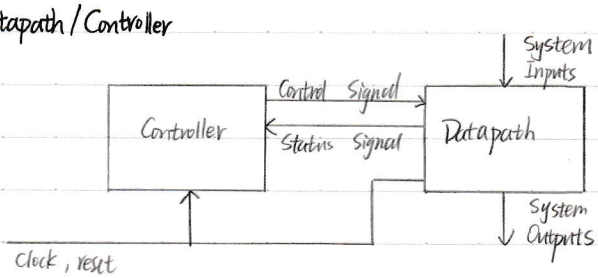
Assert Statement

assert <condition> Message will be printed if condition is not true

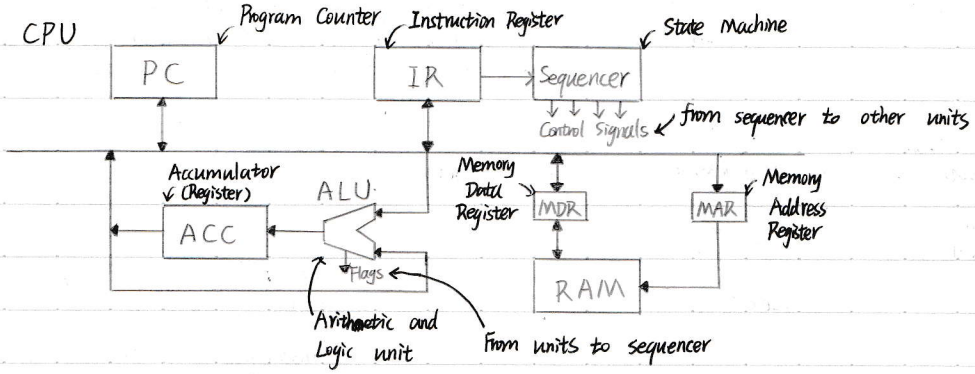
Displaying signals

$display ("%2t ns a = %b, y = %b", $time, a, y);

Datapath / Controller



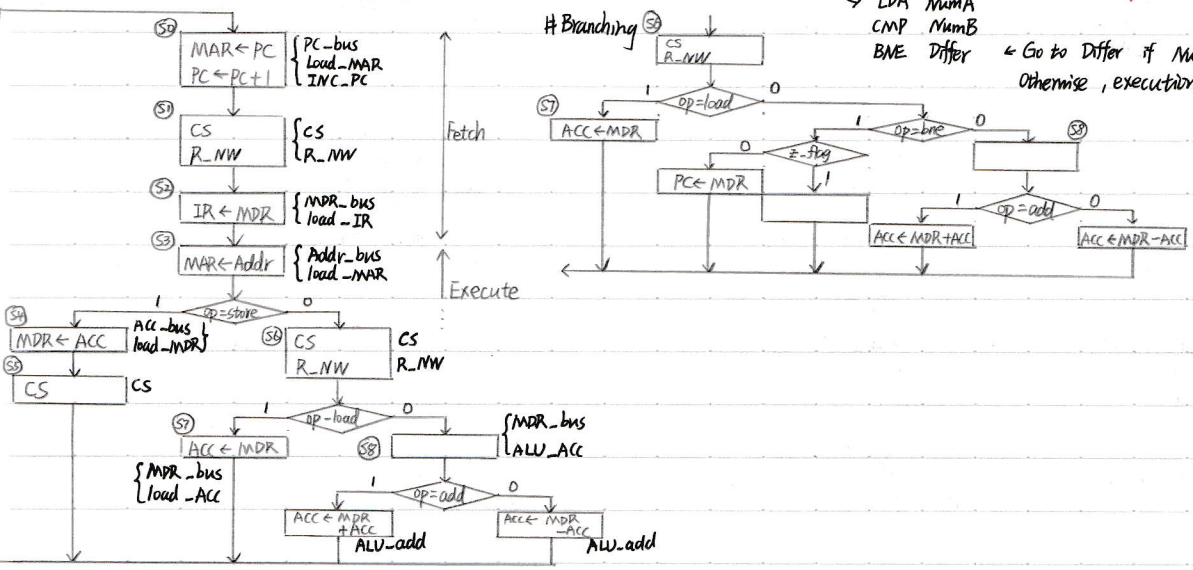
Simple CPU



[Control Signals]

① ACC_bus:	Drive bus with contents of ACC. (Enable three state output)
② load_ACC:	Load ACC from bus
③ PC_bus	Drive bus with contents of PC
④ load_IR	Load IR from bus
⑤ load_MAR	Load MAR from bus
⑥ MDR_bus	Drive bus with contents of MDR
⑦ load_MDR	Load MDR from bus
⑧ ALU_ACC	Load ACC with result from ACC
⑨ INC_PC	Increment PC, and save the result in PC
⑩ Addr_bus	Drive bus with operand part of instruction held in IR
⑪ CS	Chip select. Use contents of MAR to set up memory address.
⑫ R_NW	Read, Not Write. When false, contents of MDR are stored
⑬ ALU_add	Performed an add operation in the ALU.
⑭ ALU_sub	Perform a substract operation in the ALU

BNE: Branch if not equal
↳ LDA NumA
CMP NumB
BNE Differ ← Go to Differ if NumA ≠ NumB
Otherwise, execution continues



No.

Date

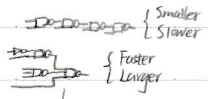
```
module sequencer #(parameter OP_W=3)(input logic clock, n_reset, z_flag, input logic [OP_W-1:0] op,
                                     output logic ACC_bus, load_ACC, PC_bus, load_PC, load_IR, load_MAR, CS, R_W,
                                     MDR_bus, load_MDR, ALU_ACC, ALU_add, ALU_sub, INC_PC, Addr_bus);
`include "opcodes.v"
enum {S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10} state;
```

```
`define LOAD  3'b000
`define STORE 3'b001
`define ADD   3'b010
`define SUB   3'b011
`define BNE   3'b100
```

```
always_comb
begin: comb
    ACC_bus = '0;
    load_ACC = '0;
    PC_bus = '0;
    load_PC = '0;
    load_IR = '0;
    load_MAR = '0;
    MDR_bus = '0;
    load_MDR = '0;
    ALU_ACC = '0;
    ALU_add = '0;
    ALU_sub = '0;
    INC_PC = '0;
    Addr_bus = '0;
    CS = '0;
    R_W = '0;
  case (state)
    S0: begin
        PC_bus = '1;
        load_MAR = '1;
        INC_PC = '1;
        load_PC = '1;
      end
    S1: begin
        CS = '1;
        R_W = '1;
      end
    S2: begin
        MDR_bus = '1;
        load_IR = '1;
      end
    S3: begin
        Addr_bus = '1;
        load_MAR = '1;
      end
    S4: begin
        ACC_bus = '1;
        load_MDR = '1;
      end
    S5: CS = '1;
    S6: begin
        CS = '1;
        R_W = '1;
      end
    S7: begin
        MDR_bus = '1;
        load_ACC = '1;
      end
```

```
always_ff @ (posedge clock, negedge n_reset)
begin: seq
  if (~n_reset)
      state <= S0;
  else
    case (state)
      S0: state <= S1;
      S1: state <= S2;
      S2: state <= S3;
      S3: if (op = `STORE)
            state <= S4;
          else
            state <= S6;
      S4: state <= S5;
      S5: state <= S0;
      S6: if (op = `BNE) LOAD
            state <= S7;
          else if (op = `BNE)
            if (~z_flag)
              state <= S9;
            else
              state <= S10;
          else
            state <= S8;
      S7: state <= S0;
      S8: state <= S0;
      S9: state <= S0;
      S10: state <= S0;
    endcase
end
```

```
    S8: begin
        MDR_bus = '1;
        ALU_ACC = '1;
        load_ACC = '1;
        if (op == `ADD)
          ALU_add = '1;
        else if (op == `SUB)
          ALU_sub = '1;
        end
    S9: begin
        MDR_bus = '1;
        load_PC = '1;
        end
    S10: ;
  endcase
  end
endmodule
```



Synthesis Considerations

{ Area, Delay
{ Clock speed

① Constraints (约束条件)
② Coding for synthesis
③ Hierarchical Designs
④ Technology Dependence
⑤ Design for Reliability

Area Constraints

Can define overall objective of synthesis to be area minimisation

define_sharing full_adder on

The compiler would attempt to share resources

State Assignment

One Hot assignment - One flip-flop per state

Good for fpga (lots of flip-flops)

State Encoding

Synplify [综合器] uses one-hot encoding by default

To force sequential encoding:

```
typedef enum {S0,S1,S2} state_type;
(* syn_encoding = "sequential" *) state_type state;
```

Timing Constraints

Clock Frequency is 20MHz → Clock period is 50ns

Delay through ff is 1ns

External Input and Output delays can be specified:

define_input_delay In1 10.0

⇒ The maximum delay through the input logic is 39ns

50 - 10 - 1 = 39

Design for Reliability

```
unique case (present_state)
A: begin
    ready = '1;
    if (twenty)
      next_state = D;
    ...
I: begin
    ret = '1;
    next_state = A;
    end
default: next_state = A;
endcase
```

When state goes to an unused state because of noise, could return to a known state or the system could switch between the unused states (known as "parasitic" state ma

模块接线　　已创建的模块　模块端口　　模块端口

e.g. counter C C.osc_clk (clock), .clock (slow_clock));

No.

Date

Testing Digital Systems

The need for testing

- Real time systems may have manufacturing defects { ① Short circuits ② Missing components ③ Damaged components
- Need to know if a system (board, IC, whole system) has a defect (and there doesn't work). Don't want to sell bad systems, Don't want to reject good systems
  ↳ Need for testing is economic

Two approach

Functional testing - does system work correctly ? ← can be simply a long and difficult task
Structural testing - does system contain a fault ?

Fault Models

- What defects occur?
- How can they be modelled ?
- Why do models imply ?

PCB Defects

- Break in connections
  - bad etching (腐蝕)
  - stress
- Short Circuits
  - Solder flow (焊锡流)
- Bad solder joints (焊点)

IC Defects

- Open circuit
  - Electromigration (电迁移)
  - Current overstress
  - corrosion
- Short Circuit
- "Latch-up" - transient currents
- Short Circuit
- Incorrect Transistor Action
  - Silicon or oxide defects
  - Mask misalignment
  - Impurities
  - Gamma radiation
- Data Corruption { Alpha particles (α粒子) EMI

How do defects manifest themselves ?

- Static failure (50%)
  - shorts, breaks etc
- Dynamic failure (49%)
  - out of spec components
  - timing failures
- Intermittent failures (1%)
  - environmental

Fault Model

- Physical defect manifests itself as a logical fault
  - Applies only to digital circuits
  - No analogue fault modeling
- Stuck Fault Model
  - Many physical defects can be modelled as a circuit node being :
    - Stuck at 1 (s-a-1)
    - Stuck at 0 (s-a-0)

Single-Stuck Fault Model

- Assumption
  - The fault directly affects only one node
  - The node is stuck at either 0 or 1

Validity of Single-Stuck Fault Model ?

- Multiple faults do occur
- Can multiple faults mask each other?
- Number of faults might rise with complexity
- Can all defects be modelled with stuck model method?
- The model appears to be valid most of the time
- Almost all test pattern generation relies on this model
- Multiple faults are found with test patterns for single faults

Fault-oriented Test Pattern Generation

- Prepare a fault list (e.g. all nodes stuck-at 0 & stuck-at 1)

repeat
  write a test
  check fault cover (one test may cover > 1 fault)
  (delete covered faults from list)
until fault cover target is reached

Test Generation

1. Test pattern generation (writing a test) may be random or optimised. Ideally we want a minimum number of tests (cheaper to apply)
2. One test may cover more than one fault, often faults are indistinguishable
3. If we simply want a pass/fail test, we can remove faults from further consideration, once we have found a test for a fault. If we want to diagnose a fault (for subsequent repair) we probably want to find all tests for a fault to deduce where the fault occurs
4. Fault cover target may be less than 100%. The higher the cover, the greater the number of tests and hence the cost of applying the test

Testability

How testable is a system ?

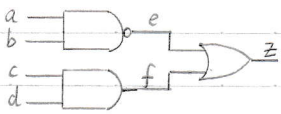
a. Controllability - can we control all the nodes to establish if there is a fault ?

b. Observability - can we observe and distinguish between a faulty node and a correct node ?

No.

Date

Sensitive Path Algorithm



- 7 nodes, ∴ 14 stuck faults:
  - a/0, a/1, b/0, b/1, ⋯, z/0, z/1
  - "a stuck-at-0" etc

To test for a/0, need to set a to 1 (fault-free condition)
Need to observe existance or otherwise of fault at z
If b is 0, e is 1 irrespective of a, ∴ b must be 1

Similarly if f is 1, z is 1, irrespective of e, ∴ f must be 0
Thus we are establishing a sensitive path from a to z
To force f to 0, either c or d or both must be 0

This can be expressed as 1101/0

If the fault a/0 exists, e is 1, z is 1. If the fault does not exist, e is 0, z is 0
We can conclude from this that a test for a/0 is: a=1, b=1, c=0, d=1 for which the fault-free output is z=0,
Other tests are 1110/0 and 1100/0. Therefore, there is more than one test for the fault a/0.

To test for e/1, requires that f=0 to make e visible at z. ∴ c or d or both must be 0
To make e=0 requires that a=b=1, So a test for e/1 is 1101/0.

~~Unstable~~ Untestable FAULT ARE ALWAYS DUE TO REDUNDANCY

Algorithm

① Select a fault
② Setup input to force the node to a fixed value
③ Set up input to transmit node value to an output
④ Check for consistency
⑤ Check for coverage of other faults.

a	b	c	d	e	f	z	
1	1	0	1	0	0	0	a/0, z/1, e/1, f/1, b/0 c/1
0	1	0	1	1	0	1	a/1 z/0 e/0
1	0	0	1	1	0	1	b/1 z/0 e/0
1	1	1	1	0	1	1	c/0 z/0 f/0 d/0
1	1	1	0	0	0	0	d/1

Testing Sequential Circuits

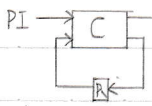
Testing combinational circuits is relatively easy, provided, there is no redundancy in the circuit.

Number of test vectors << 2^no. of inputs

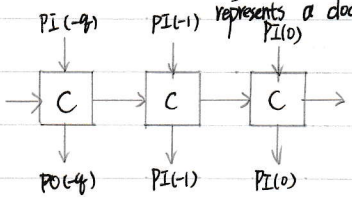
Testing sequential circuits is ~~differe~~ difficult because circuits have states. May require long sequences of inputs to reach states. Some ~~&~~ faults may be unstable, because certain states cannot be reached.
N.B. All sequential circuits MUST have a set or reset to initialise all flip-flops or testing is nearly impossible.

Sequential ATPG

Combinational ATPG uses sensitive path algorithm. i.e. sensitise a path from a potential fault site to a PO (or in the case of SISO, a pseudo PO - a scan register input).
Can model a sequential circuit.



As an iterative array, each element represents a clock cycle.




```
For each SSL SSF in C
    r=1 ; q=0;
    repeat
        build model with r+q time frames;
        ignore POs in 1st r+q-1 time frames;
        generate test CD algorithm) for att fault as if all time frames
        represent a combinational circuit. PIs can be set up in all time frames;
        only POs in last time frame observed.
    if Success the return Success!;
    else increase increment r or q;
    until r+q = fmax;
return failure!;
```

Design For Test

Ad Hoc Design for Test Method

① Controllability – ability to control logic value of an internal node from PI
② Observability – ability to observe logic value of internal node at PO

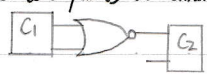
Testability can be enhanced by:

{ ad hoc design guildelines
{ structured design methodology

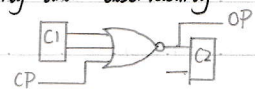
Things to watch:
① Redundant Logic – undetectable faults
② Asynchronous sequential systems
– difficult to synchronize with tester
– if necessary, confine to independent blocks
③ Monostables
– difficult to control

Improve Test Access

Use test points to enhance controllability and observability



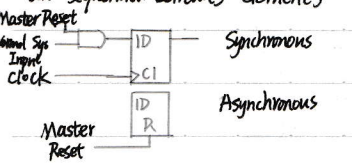
Original Circuit



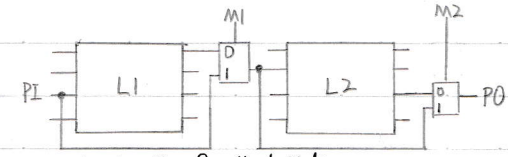
Circuit modified to provide CP and OP

Initialisation

Initialisation must be provided for all sequential circuits elements



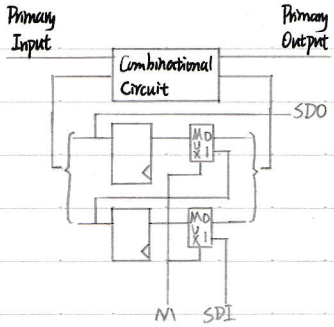
Multiplexers



M1M2 = 00 : Operational Mode
M1M2 = 01 : Test L1 – observe N at PO
M1M2 = 10 : Test L2 – observe Control N from PI

SISO

{ SDI : Serial Data In
{ SDO : Serial Data Out



- Modify all flip-flops to include MUX
- Make every state controllable and observable
- Sequential test problem reduce to combinational test

SISO Test Generation

- ~~Generation~~ Generate tests for the combinational part using the sensitive path algorithm (or a variant) as before.
- For each test, need to distinguish between inputs that are applied through the scan path, and those that are applied through the primary inputs.
- Same with outputs

Running a test

① Set M=1 and test the flip-flops as a shift register. If a sequence of 1s and 0s is fed to SDI, we would expect the same sequence to emerge from SDO delayed by the number of clock cycles equal to the length of the shift register (n). A useful test sequence would be 00110... which tests all transitions and whether the flip-flops are stable

② For each combinational test
a) Set M=1 to set the state of the flip-flops after n clock cycles by shifting a pattern in through SDI
b) Set M=0. Set up the primary inputs. Collect the values of the primary outputs. Apply 1 clock cycle to load the state outputs into the flip-flops
c) Set M=1 to shift the flip-flop contents to SDO after n-1 clock cycles.

Costs of enhanced testability

- Extra I/O pins
- Includes interface etc
- Extra components (MUXs), extra wiring
- Degradation of performance because of extra gates in signal paths
- More things to go wrong

But the circuit will be easier to test
Inserting a scan path is easy
Ad hoc methods are difficult to automate

Scan path can be inserted before or after layout. Ordering is therefor arbitrary.
Test vectors need to be sorted – a job for a computer

Reducing the Cost

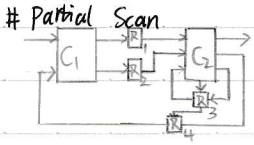
- Share SDI with PI (MUX)
- Share SDO with PO
- Do all flip-flops need to be in the scan path
- Build MUXes into next state logic

$S^+ = \bar{S}.T + S.\bar{T}.\bar{A}$
$T^+ = \bar{S}.\bar{T}.A + S.T.\bar{A}$

Becomes
$S^+ = (\bar{S}.T + S.\bar{T}.\bar{A}).\bar{M} + T.M$
$T^+ = (S.T + \bar{S}.T.\bar{A}).\bar{M} + SDI.M$

- Reduces delay

Partial Scan



Don't put every flip-flop in the scan path

C1/R1/R2/C2 form a balanced structure
Similarly C2/R4/R1 are balanced

We can make R3 and R4 part of the scan path (or R_1, R_2, R_3)

No.

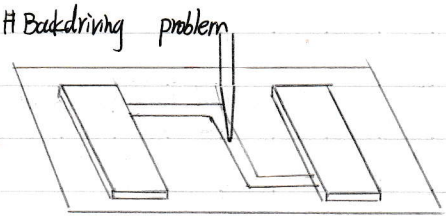
Date

Boundary Scan (JTAG)

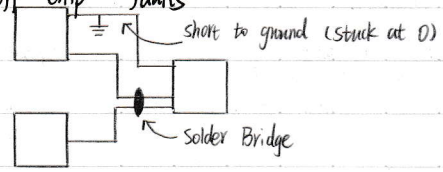
Boundary Scan

- IEEE 1149.1 Also known as JTAG
- Not possible to test mounted ICs (the pins may be connected together);
- PCBs now often have more than 20 layers of metal, so deep layers cannot be reached;
- Density of components on a PCB is increasing. Multi-chip modules (MCMs) take the chip/board concept further and have unpacked integrated circuits mounted directly on a silicon substrate.

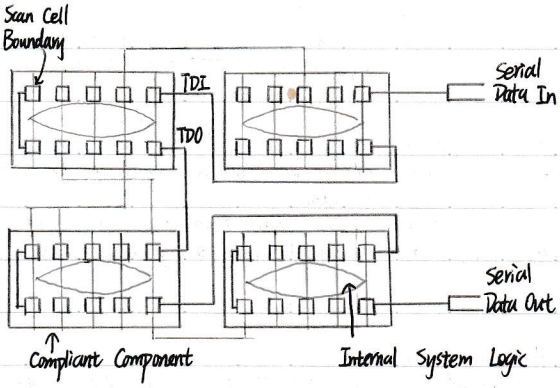
Backdriving problem



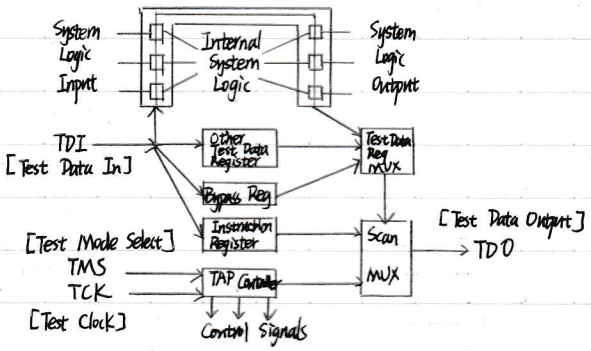
Off chip faults



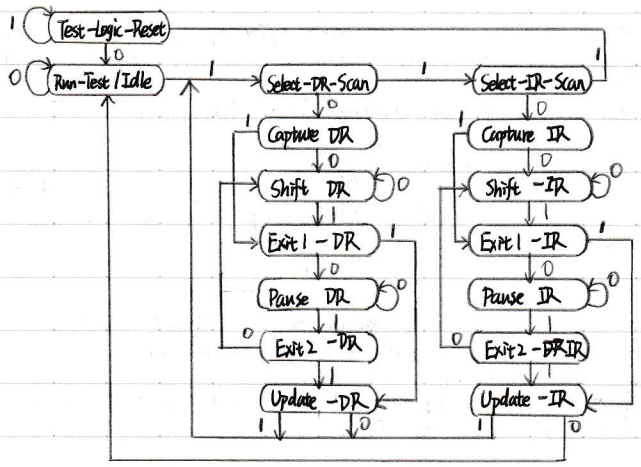
Principle of JTAG



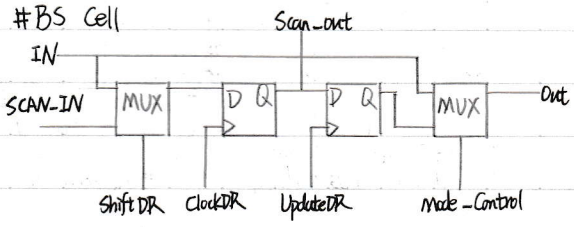
Test Architecture



TAP Controller



BS Cell



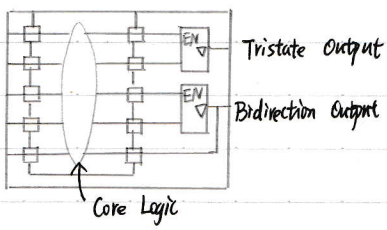
Modes of Operation

1. Normal mode
   - Normal system data flows from IN to OUT.
2. Scan mode
   - ShiftDR selects the SCAN_IN input, clockDR clocks the scan path
   - ShiftDR is derived from the similarly named state in the TAP controller
   - ClockDR is asserted when the TAP controller is in state Capture-DR or Shift-DR
   - (Hence, the Boundary Scan architecture is not ~~truely~~ truly synchronous)
3. Capture mode (捕获) of the system
   - ShiftDR selects the IN input, data is clocked into the scan path register with ClockDR to take a snapshot
4. Update mode
   - After a capture or scan, data from the left flip-flop is sent OUT by applying 1 clock edge to UpdateDR
   - Again, the clock signal comes from the TAP controller when it is in state Update-DR. The TAP controller then enters the Run Test state and MODE_CONTROL is set as appropriate according to the instruction held in the instruction register.

Instructions

- EXTEST (Mandatory) 外测试
  - This instruction performs a test of the system, external to the core logic of particular devices.
  - Data is sent from the output boundary scan cells of one device, through the pads and pins of that device, along the interconnect wiring, through the pins and pads of a second device and into the input boundary scan cells of that second device
  - Hence a complete test of the interconnect from one IC core to another is performed.
- SAMPLE/PRELOAD (Mandatory) 预加载
  - This instruction is executed before and after the EXTEST and INTEST instructions to set up pin outputs and to capture pin inputs.
- BYPASS (Mandatory)
  - This instruction selects the Bypass register, to shorten the scan path.
- RUNBIST (Optional) 自测试
  - Runs a built-in self-test on a component
- IDCODE, USERCODE (Optional) device). The code is put into the scan path
  - These instructions return the identification of the device (and the user identification for a programmable logic
- INTEST 内部测试 (optional)
  - This instruction uses the boundary scan register to test the internal circuitry of an IC
  - Although ~~th~~ such a test would normally be performed before a component is mounted on a PCB, it might be desirable to check that the process of soldering the component onto the board has not damaged it
  - Note that the internal logic is disconnected from the pins, so if pins have been connected together on the board, that will have no effect on the standard test.
- CONFIGURE (Optional)
  - An SRAM-based FPGA needs to be configured each time ~~a~~ power is applied.
  - The configuration of the FPGA is held in registers. These registers can be linked to the TAP interface.
  - This clearly saves pins as the configuration and test interfaces are shared.

Tristates and Bidirectionals.



No.

Date

Built-in self-test

Why Built-in Test

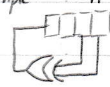
- Economic justification
  - Simplifies test equipment
  - Simplifies TPG
  - Allow easy field test
- Increases user confidence

↙ Built in self test ...

How to generate test vectors

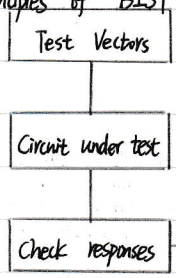
- Store pre-generated vectors in ROM
  - Possibly a very large number
  - Problem with sequential logic
- Exhaustive test
  - Use binary counter to generate all test vectors
  - Separate combinational logic from registers

Example: n=4



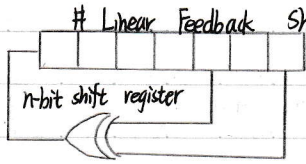
	Q1	Q0
1	1	1
2	0	1
3	0	0
4	1	0
5	1	1
6	0	1
7	0	0
8	1	0
9	1	1
10	0	1
11	1	0
12	0	1
13	1	0
14	1	1
15	1	1
16	1	1

Principles of BIST



Go/No Go
(Perhaps diagnostics)

Linear Feedback Shift Register (LFSR)



Generate test patterns on chip

$2^n - 1$ patterns, if feedback chosen correctly

Easy to build, 1-3 XOR gates

Pseudo-random sequence – can have implications for power dissipation. If n is large (>32), time taken for sequence may be unacceptable. All 0s state excluded.

LFSR - Feedback connections

- n=2 D1 = Q1 XOR Q0
- n=4 D3 = Q1 XOR Q0
- n=8 D7 = Q3 XOR Q0
- n=16 D15 = Q5 XOR Q4 XOR Q3 XOR Q0
- n=24 D23 = Q7 XOR Q2 XOR Q1 XOR Q0

Not more than 24 feedback connections are needed to generate $2^n - 1$ patterns

For n=24, >250,000 possible feedback connections

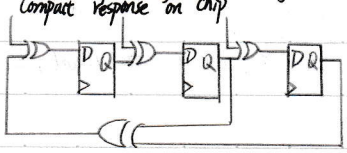
Circuit can be modified to generate all 0s state (2^n patterns)

Check Responses

- Look-up table of correct responses
  - potentially very large
- Signature Analysis 簽名分析
  - Data compression

Multiple Input Signature Register

Compact response on chip



Probability of a fault in the CUT giving the fault-free signature → 2^{-n} (aliasing)

Easy to build

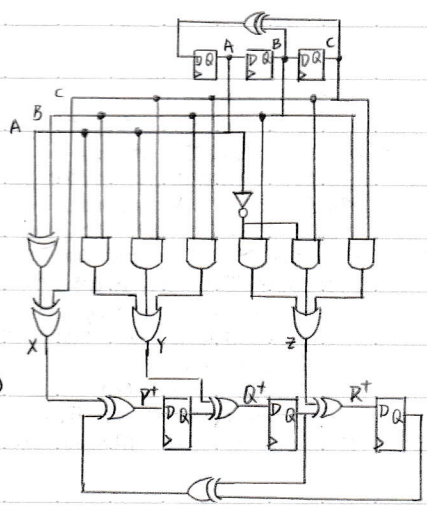
BIST Example

This circuit consists of:
- a 3 stage LFSR
- a 3 stage MISR
- a circuit under test

$$\begin{cases} X = A \text{ XOR } B \text{ XOR } C \\ Y = A.B + A.C + B.C \\ Z = A'.B + A'.C + B.C \end{cases}$$

LFSR:
$$\begin{cases} A^+ = B \text{ XOR } C \\ B^+ = A \\ C^+ = B \end{cases}$$

MISR:
$$\begin{cases} P^+ = X \text{ XOR } (Q \text{ XOR } R) \\ Q^+ = Y \text{ XOR } P \\ R^+ = Z \text{ XOR } Q \end{cases}$$



Fault Free Simulation

$P^+ = Q + R$
$Q^+ = \cancel{R}\ P$
$R^+ = Z + Q$

Both the LFSR and MISR are initialised to the 111 state. By simulation, the sequence of states is found to be:

LFSR a b c	CUT x y z	MISR P Q R	z	P Q R
1 1 1	1 1 1	1 1 1	1	1 1 1
0 1 1	0 1 1	1 0 0	1	0 1 0
0 0 1	1 0 1	0 0 1	1	1 0 0
1 0 0	1 0 0	0 0 1	0	0 1 1
0 1 0	1 0 1	0 0 0	1	0 0 1
1 0 1	0 1 0	1 0 1	0	1 0 1
1 1 0	0 1 0	1 0 0	0	1 1 0
1 1 1	1 1 1	0 0 0	1	1 1 1

↑ signature of fault-free circuit

Simulation of a fault

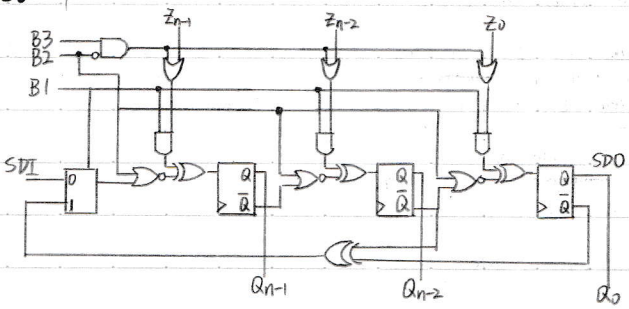
Suppose a is s-a-0

a b c	x y z		
1 1 1	0 1 1	1 1 1	1
0 1 1	0 1 1	0 0 0	2
0 0 1	1 0 1	0 1 1	3
1 0 0	0 0 0	1 0 0	4
0 1 0	1 0 1	0 1 0	5
1 0 1	1 0 1	0 0 0	6
1 1 0	1 0 1	1 0 1	7
1 1 1	0 1 1	0 1 1	8

↑ Signature of faulty circuit

[Note] If a is s-a-1, the signature 000 is also generated. This is an example of aliasing

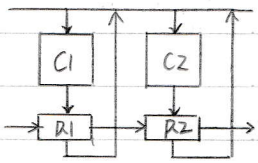
#BIBO



Combines LFSR and MISR into one

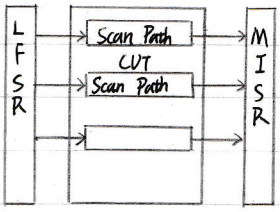
Typically 5 modes: ① normal
② scan
③ synchronous reset
④ MISR
⑤ LFSR

#BIBO Example



- To test C1, R2 is LFSR, R1 is MISR
- To test C2, R1 is LFSR, R2 is MISR
- Scan signatures in R1, R2 to some comparator
- NB 2 test sessions, each of 2^{n-1} cycles + scan + decision

#STUMPS



A 32-bit LFSR would take about 2s to go through the entire sequence at 2 GHz

STUMPS avoids this problem using a single LSFR and MISR to generate patterns for all scan paths

Technique Technique adopted by Logic Vision

Disadvantage - same input to each scan path, delayed by a cycle each time

No.

Date

System Verilog Simulation

SystemVerilog Simulation

- Semantics (meaning) of SystemVerilog is defined by the simulation model (语义学)
- Want to write SystemVerilog that synthesises correctly and that behaves in the same way before and after synthesis
- Difficult simulators may produce different behaviour with poorly written model

Simulation Time

- The simulator models time - this is called simulation time.
- Simulation time is an integral multiple (整数倍) of the resolution limit
- The simulator cannot measure time delays less than the resolution limit.
- For gate-level simulations the resolution limit may be quite fine, possibly down to 1ps.
- For RTL (Register Transfer Level) simulators, there is no need to specify a fine resolution since we are only interested in clock-cycle by clock-cycle behaviour and then transfer functions are described with zero delay or unit delay
- In this case, a resolution limit of 1ns is usually used.

Verilog Simulation

- A new event may be one of five types. The LRM talks of a stratified (分层的) event queue, with five regions as follows:
  ① Active events. These occur at the present time and can be processed in any order.
  ② Inactive events. These occur at the present time and are processed after all the active events have been processed.
  ③ Nonblocking assign update events
  ④ Monitor events
  ⑤ Futher events

Event Regions

- Prepned - sample stable values for later checking
- Active - blocking assignments
- Inactive - zero delay assignments here
- NBA - nonblocking assignments
- Observe - Evaluate assertions
- Reactive - Execute programs in testbenches
- Postponed - $ store and $ monitor print routines

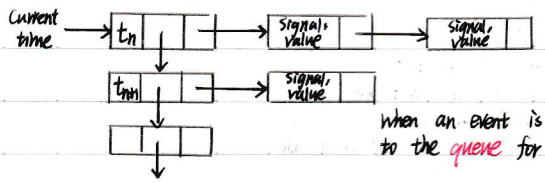
Event Driven Simulation

- The basis of SystemVerilog simulation is event processing. SystemVerilog ~~simulation~~ simulators are event driven simulators.
- There are 3 essential concepts to event-driven simulation:
  - simulation time
  - simulation regions
  - ~~simulation~~ event processing
- The VHDL model is different

Simulation Cycle

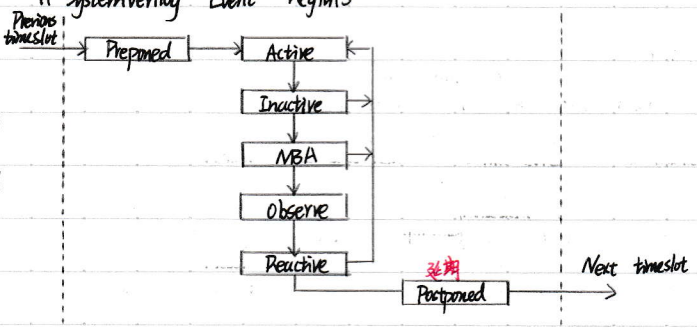
- Process Execution
  - each process is recalculated only if its sensitivity list has an event
  - this creates a new value for each target signal and a time at which the signal gets the value. This value/time pair is a new event and is scheduled for the current or a future time.

Event Queue



when an event is created it is added to the queue for the scheduled time

SystemVerilog Event Regions



Simulation Cycle

```
while (there are events) {
    if (no active events)
        if (there are events in the other lists (in order))
            activate all events in the next list;
        else
            advance T to the next event time;
    E = any active event;
    if (E is an update event) {
        update the modified event object;
        add evaluation events for sensitive processes to event queue;
    }
    else {
        evaluate the process;
        add update events to the event queue;
    }
}
```

Reactive events are generated by assignments in program blocks.

A program block is like a module, but
- Can only be instantiated (実体化) in a testbench
- Can only contain initial procedural blocks

Introduced in SystemVerilog to remove testbench problems

Every Events may be processed from the active event list in any order (or to think of it in another way, events can be added to the events list in any order)

This is the fundamental cause cause of non-determinism in Verilog

We can be sure of only two things:
- Statements between begin and end will be executed in the order stated
- Nonblocking assignments will be performed after active events.

Everthing else is indetermine.

Moreover, the execution of a procedural block can be ~~introduced~~ interrupted to allow another procedural blak to be executed.

Therefore The skill in writing Verilog code is therefore to ensure that this indeterminism does not matter.

If the code is badly written, a race condition is likely to result – that is a situation where the procedure writing a value and the procedure reading that value are racing each other.

Depending which completes first, either the original or the updated value may be read.

The list of active events is one of the lists, Inactive, Nonblocking, Reactive, that has been created during some previous simulation cycle, together with any (active) events that are generated during the current cycle.

Nonblocking assign update events are created by nonblocking assignments (<=)

The evaluation of the right hand side of all nonblocking assignments is always done before any nonblocking assign updates are done.

This is important as it allows sequential systems to be modelled correctly.

Inactive events are those events that are due to occur at the current time but that have been explicitly delayed,

In practice, this can be done with a zero delay (#0)

A zero delay doesn't represent real hardware (nor a useful testbench ■ construct). Therefore you are simply trying to fool the simulator.

Unless you know exactly what you are doing, it will probably fool you

Race 1

```
assign p = q;
initial
  begin
    q = 1;
    # q = 0;
    $ display (p);
  end
```

Because the execution of the initial procedure can be interleaved (交互作用) with the assign statement, the value of p that is displayed could be 1 or 0. Either is "correct" and different simulators may give different result

Race 2

```
always @ (posedge clock)
    b = c;
always @ (posedge clock)
    a = b;
```

This is supposed to model two flip-flops connected in series.

If the procedures are evaluated in the order written, at a rising edge the value of c will be copied to b.

That same value is copied to a. This is clearly not the ~~intent~~ intended behaviour.

If the procedures are evaluated in the opposite order, the correct behaviour is modelled.

Avoiding Races.

Don't assign to the same variable from two or more procedures variables. Not only is contention liable, but multiple assignment can cause ambiguous behaviour

Part of the problem is the keyword "initial". Procedures defined with the initial keyword are executed once: they are not initialised

Use nonblocking assignments for modulling sequential logic.

The example above, evaluates correctly if the two assignments are made nonblocking, irrespective of the order of evaluation.

This is because nonblocking assignments are evaluated last and cannot influence each other.

Use blocking assignments to evaluate combinational logic.

Assignments made in combinational logic models take immediate effect. The use of nonblocking assignments would often be confusing (and wrong)

Do not mix blocking and nonblocking assignments in the same procedure.

Don't use zero delays (#0). They are not necessary and will cause confusion.

No.

Date

Different ways to model delays.

- Left-hand side (LHS) of blocking assignment:
  `#5 a=b;` Wait then do assignment. Used for generating waveforms in a testbench
- Righ-hand side (RHS) of blocking assignment;
  `a=#5 b;` Pure delay, e.g. a transmission line
- LHS of nonblocking assignments:
  `#5 a<=b;` No real different difference to first form
- RHS of nonblocking assignments:
  `a <= #5;` Pure delay. Useful to model delays in flip-flops.
- LHS of continuous assignments:
  `assign #5 a=b;` Inertial delay in combinational logic

{ Can get weired behaviour if you use the wrong form
Can get differences between simulators
Can specify zero delays - but don't

wide is suppressed.
Inertial delay means that any transition is delayed by 5 units AND any pulse less than 5 units

VHDL Simulation

- VHDL uses a different terminology to Verilog
  - transaction generation
  - event processing
- Additionally it has a different mechanism for 0 delays:
  - delta time

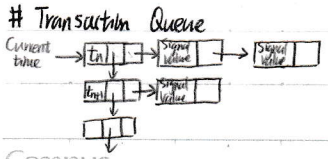
VHDL Simulation Cycle

Simulation alternates between two modes:

① Statement Execution
- each statement is recalculated only if its sensitivity list has an event
- this creates a new value for each target signal and a time at which the signal gets the value. This value/time pair is called a transaction and is added to a transaction queue.
- target signals are not updated during statement execution.

② Event processing
- transactions for the current time are taken off the queue.
- if a transaction represents a change in the value of the signal it becomes an event, causing the signal to be updated with new value
- Signals are updated only at the next wait statement.
  - A process with a sensitivity list has an implicit wait at the end of the process.
  - Don't mix sensitivity lists and wait statement - must have one or the other.

Transaction Queue

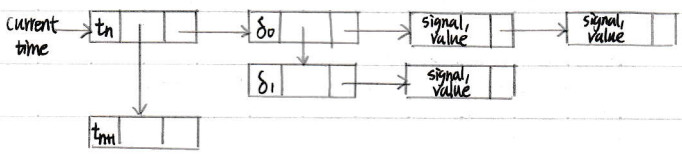


Zero-Delay Transaction

- It is also possible to have a zero-delay assignment
  `a <= '0';`
- The signal is still not updated immediately
- But, a transaction still needs to be scheduled.
  - If the transaction were scheduled at the current time:
    - It could be put at the start of the transaction queue
    - It could be put at the end of the transaction queue (easier to program)
  - Therefore the exact behaviour would depend on the simulator (not good - but see Verilog!)

Delta Time

- Zero-delay transactions are a problem - how ~~many~~ do you handle a transaction for the current time?
- Answer: don't! Schedule them for one delta time later.
  - a delta time is infinitesimally small



Each event generation phase consumes one delta time-step transactions

SystemVerilog Assertions

Verification

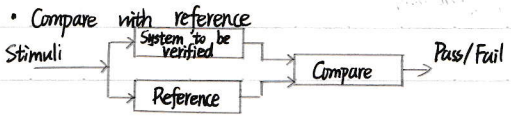
- Simulation is the main method for verifying designs
- Inspecting waveform is difficult and errorprone (容易出错的)
- The principle of Assertions Based verification is:
  - Describe what you expect to see, and when you expect to see it
  - The simulator will tell you if this doesn't happen

Writing data

- Verilog has simple output tasks
  - $display - writes data
  - $store - writes data after all events have been processed
  - $monitor - continuously monitors variables
- Use formatting commands like C
  - $display ("Output = %b", a);

Condition

- Compare with reference



- Reference could be behavioural (simulation) model
- ABV combines reference and comparsion.

SystemVerilog Assertions

- Introduction - not a full tutorial
- SVAs can be used with VHDL/Verilog models and testbenches.

EXAMPLE - Lift Controller

Toy lift - Three floors
- Sensors - top, bottom, middle_plus, middle_minus
- Call buttons
- Indicator lamps

- All signals active high
- Rising edge of clock is active (posedge in verilog)
- Direction - 1 is up, 0 is down
- Seven states

Invariant

- An invariant is a property that is always true.

```
always @*
  assert (! top || !bottom)        top & bottom
    else $ error ("At top and bottom !");
```

- @* is a default sensitivity list
- assertion is evaluated whenever top or bottom changes
- Assertion that one or both of top and buttom is true
  - i.e. that both ~~become~~ cannot be 1 at the same time
- Is this really useful?
  - Better to synchronise with clock!

What is a Property?

- A fact about your design
- In the HDL code or in a separate file
- Properties are used to write
  - Simulation checkers
  - Constrains defining legal Simulation inputs
  - Assertions to be proved by static formal property checkers
  - Assumptions made by static formal tools
  - Functional coverage points

Property

```
property Not In 2 Places;
  ( !top || !bottom );
endproperty
```

Assert property at clock edge

↓

```
assert property (
  @(posedge clock) NotIn2places);
```

Temporal Properties

- So far, no better than Verilog
- Want to write properties that hold over time:

```
property (@(posedge clk) a |=> b);
```

b is true in the cycle after a is true

- Or we can write sequences:

```
a ##1 b ##1 c |=> d
```

##1 means one clock cycle

- a ##[1:4] b one to four clock cycles

Temporal Property

If the lift is at the buttom and the call1 buttom is pressed, we expect that the lift will go up in the next clock cycle

```
property Call1;
 ( bottom && call1 |=> enable && direction);
end-property
```

- |=> is non-overlapping implication

No.

Date

Complex Properties

- So, it's possible to write complex properties like
  - "If a is true for 3 cycles, b will be true for at least 2 cycles, not more than 2 cycles later"
- In order words, this is the reference model against which we check the behaviour of the system.
- As noted, this reference model can be used in simulations and by model checkers.

Clocking

- All assertions are tested on rising edge of the clock.
  - We can make this a default condition

```
default clocking clock_block
  @(posedge clock);
endclocking
```

- Hence

```
assert property (Call1);
```

Non-overlapping
~~Nonblocking~~ occur next CLK
Overlapping
~~Blocking~~ occur within CLK

Coverage

- A Verilog simulator can measure how often each line is executed (if ever)
  - known as coverage
  - Built into ModelSim
- Can ~~often~~ also measure property coverage
  - How often is each property tested in a simulation?
  - Gives us a measure of how good our simulation is.
- How many times is the property checked?

```
cover property (Call1);
```

- Just because an assertion does not fail, doesn't mean that it's ever been tested!
- If (bottom && call1) is never true, the assertion passes vacuously. (空自由)

Overlapping Implication

If the ground floor indicator is lit, the lift should be going down

```
property Indicator0;
  (indicator 0 |-> !direction);
endproperty
```

- |-> overlapping implication — same clock cycle.

Sequences

If the top indicator is 0 and then goes to 1, the lift should be at the top in the next clock cycle.

```
property GetTo2;
  (indicator2 ##1 !indicator 2 |=> top);
end property
```

- ## refers to the clocking method
- Could use an overlapping implication here

Liveness

```
property Eventually2;
  (button && call2 |-> ## [1:$] top);
end property
```

- [1:$] means a range of 1 to infinite number of clock cycles.
- If the lift is at the bottom and the top call button is pressed, the lift goes to the top floor eventually

Liveness & Safety

- "Liveness" means that good things happen eventually.
- "Safety" means that bad things never happen
- Liveness properties are not a good idea
  - What does eventually mean?
  - Properties can pass vacuously, so how can we know it's failed?

States

```
property StateChange;
  (state == floor 0) && call 1 |=> (state == upto1);
endproperty
```

- Mapping states is a little convoluted, but possible

How do we know the property is correct?

- We don't
- The properties have to be written and debugged at the same time as the HDL model
- But not by the same engineer.

No.

Date

System Verilog Assertions Ext

Assertions are primarily used to validate the behaviour of a design ("Is it working correctly"). They may also be used to provide functional coverage information for a design ("How good is the design."). Assertion can be checked dynamically by simulating or statically by a separate property check tool - i.e. a formal verification tool that provides proves whether or not a design meets its specification. Such tools may require certain assumption about design's behaviour to be specified.

In systemVerilog there are two kinds of assertions : immediate (assert) and concurrent (assert property). Coverage statements (cover property) are concurrent and have the same syntax (语法) as concurrent assertions, as do assume property statements.
Another similar statement - expect - is used in testbenches; it is a procedural statement that checks that some specified activity occurs. The three types of concurrent assertion statement and the expect statement make use of sequences and properties that describe the design's temporal behaviour - i.e. behaviour over time, as defined by one or more clocks.

Immediate Assertions

Immediate assertions are procedural statements and are mainly used in simulation. An assertion is basically a statement that something must be true, similar to if statement. The difference is that an if statement does not assert that an expression is true, it simply checks that it is true, e.g:

```
if (A==B)     // Simply checks if A equals B
assert (A==B) // Asserts that A equals B, if not, an error is generated.
```

If the condition expression of the immediate assert evaluates to X, Z or 0, then the assertion fails and the ~~simulation~~ simulator writes an error message.

An immediate assertion may include a pass statement and/or a fail statement. In our example the pass statement is omitted, so no action is taken when assert expression is true. If the pass statement exists:

```
assert (A==B) $display ("OK. A equals B");
```

it is executed immediately after the evaluation of the assert expression. The statement associated with an else is called a fail statement and is executed if the assertion fails:

```
assert (A==B) $display ("OK. A equals B");
    else $error ("It's gone wrong");
```

Note that you can omit the pass statement and still have a fail statement

```
assert (A==B) else $error ("It's gone wrong");
```

The failure of an assertion has a severity associated with it. There are three severity system tasks that can be included in the fail statement to specify a severity level: $fatal, $error (the default severity) and $warning. In addition, the system task $info indicates that the assertion failure carries no specific severity.
Here are some examples:

```
ReadCheck:  assert (data === correct_data)
              else $error ("memory read error");
Igt10:      assert (I>10)
              else $warning ("I is less than or equal to 10");
```

The pass and fail statements can be any legal SystemVerilog procedural statement. They can be used, for example, to write out a message, set an error flag, increment a count of errors, or signal a failure to another part of the testbench.

```
AeqB:       assert (a===b)
              else begin error_count++; $error ("A should equal B"); end
```

Concurrent Assertions

The behaviour of a design may be specified using statements similar to these: "The Read and Write signals should never be asserted together." "A Request should be followed by an Acknowledge occuring no more than two clocks after the Request is asserted."
Concurrent assertions are used to check behaviour such as this. There are statements that assert that specified properties must be true.

E.g. `assert property (!(Read && Write));` assert property that the expression Read && Write is never true at any point during simulation.

Properties are built using sequences. For example,

`assert property (@(posedge clock) Req |-> ##[1:2] Ack);` where Req is a simple sequence (just a boolean expression) and ##[1:2] Ack is a more complex sequence expression, meaning that Ack is true on the next clock, or on the one following (or both). |-> is the implication operator, so this assertion checks that whenever Req is asserted, Ack must be asserted on the next clock, or the following clock.

Concurrent assertions like these are checked throughout simulation. They usually appear outside any initial or always blocks in modules, interfaces and programs. Concurrent assertions may also be used as statements in initial or always blocks. A concurrent assertion in an initial block is only tested on the first clk.

The first assertion example above does not contain a clock. Therefore it is checked at every point in the simulation. The second assertion is only checked when a rising clock edge has occured; the values of Req and Ack are sampled on the rising edge of Clock.

C=a→b

a	b	c
0	0	1
0	1	1
1	0	0
1	1	1

No.

Date

Implication

The implication construct (|->) allows a user to monitor sequences based on satisfying some criteria (有准), e.g. attach a precondition to a sequence and evaluate the sequence only if the condition is successful. The left-hand side operand of the implication is called the antecedent (前提) sequence expression, while the right-hand side is called the consequent (結論) sequence expression.

If there is no match of the antecedent sequence expression, implication succeeds vacuously by returning true. If there is a match, for each successful match of the antecedent sequence expression, the consequent sequence expression is separately evaluated, beginning at the end point of the match. There are two forms of implication: ① ~~overlapping~~ overlapped |-> ② non-overlapped |=> (overlap 重疊)

For the overlapped implication, if there is a match for the antecedent sequence expression, then the first element of the consequent sequence expression is evaluated on the same clock tick,

```
s1 |-> s2 ;
```

In the example above, if the sequence s1 matches, then sequence s2 must also match. If sequence s1 does not match, the result is true

For non-overlapped implication, this first element of the consequent sequence expression is evaluated on the next clock tick

```
s1 |=> s2;
```

The expression above is basically equivalent to:

```
`define true 1
s1 ##1 `true |-> s2;
```

Where `true is a boolean expression, used for visual clarity, that always evaluates to true.

Properties and Sequences

In these examples, we have been using, the properties being asserted are specified in the ass assert property statements themselves. Properties may also be declared separately, for example:

```
property not_read_and_write ;
    not (Read && Write) ;
endproperty assert property (not_read_and_write) ;
```

Complex properties are often built using sequence. Sequences, too, may be declared separately:

```
~~assert~~ sequence request          sequence acknowledge        property handshake ;                    assert property (handshake)
    Req ;                              ##[1:2] Ack ;               @(posedge Clock) request |-> acknowledge ;
endsequence                        endsequence                 endproperty
```

Assertion Clocking

Concurrent assertions (assert property and cover property statements) use a generalised model of a clock and are only evaluated when a clock tick occurs. (In fact the values of the variables in the property are sampled right at the end of the previous time step.) Everything in between clock ticks is ignored. This model of execution corresponds to the way a RTL description of a design is interpreted after synthesis.

A clock tick is an atomic moment in time and a clock ticks only once at any simulation time. The clock can actually be a single signal, a gated clock (e.g. (clk && GatingSig)) or other more complex expression). When monitoring asynchronous signals, a simulation time step corresponds to a clock tick.

The clock for a property can be specified in several ways:

① Explicitly specified in a sequence:

```
sequence s ;
  @(posedge clk) a ##1 b;
endsequence
property p
    a |-> s;
endproperty
assert property (p);
```

② Explicitly specified in the property

```
property p;
  @(posedge clk) a ##1 b;
endproperty
assert ~~(p);~~ property (p);
```

③ Explicitly specified in the concurrent assertion:

```
assert property (@(posedge clk) a ##1 b);
```

④ Inferred from a procedural block

```
property P ;
  a ##1 b ;
endproperty
always @(posedge clk) assert property (p) ;
```

⑤ From a default clock

```
default clock cb;
```

⑥ From a clocking block

```
clocking cb @(posedge clk);
    property p;
      a ##1 b;
    endproperty
endclocking
assert property (cb.p);
```

Handling Asynchronous Reset

In the following example, the disable iff 用 clause allows an asynchronous reset to be specified.

```
property p1;
  @(posedge clk) disable iff (Reset) not b ##1 c;
endproperty
assert property (p1);
```

The not negates the result of the sequence following it. So, this assertion means that if Reset becomes true at any time during the evaluation of the sequence, then the attempt for p1 is a success. Otherwise, the sequence b ##1 c must never evaluate to true

Sequence

A sequence is a list of boolean in a linear order of increasing time. The sequence is true over time if boolean expressions are true at the specific clock ticks. The expressions used in sequences are interpreted in the same way as the condition of a procedural if statement

The ## operator delays execution by the specified number of clocking events, or clock cycles.

```
a ##1 b      // a must be true on the current clock tick and b on the next clock tick
a ##N b      // check b on the Nth clock tick after a
a ##[1:4] b  // a must be true on the current clock tick and b on some clock tick between the fist and fourth after the current clock tick
```

The * operator is used to specify a consecutive repetition of the left-hand side operand

```
a##1b[*3] ##1c  // Equiv. to a ##1 b ##1 b ##1 c
(a ##2b)[*2]    // Equiv. to (a##2b ##1 a ##2b)
(a##2b)[*1:3]   // Equiv. to (a##2b) or (a##2b ##a ##2b) or (a##2b ##1 a ##2b ##1 a ##2b)
```

The $ operator can be used to extend a time window to a finite, but unbounded range.

```
a##b[*1:$] ##1 c  // E.g. a b b b b c
```

The [→ or goto repitition operator specifies a non-consecutive sequence.

```
a##1b[→1:3]##1c  // Eg. a ! b b b ! b ! b c c
```

This means a is followed by any number of clocks where c is fulse, and b is true between 1 and three times, the last time being the clock before c is true.

The [= or non-consecutive repitition operator is similar to goto repetition, but the expression (b in this example expression) need not be true in the clock cycle before c is true.

```
a##1b[=1:3] c  // E.g. a !b b b !b !b b !b !b c
```

No.

Date

SystemVerilog Simulation Extra

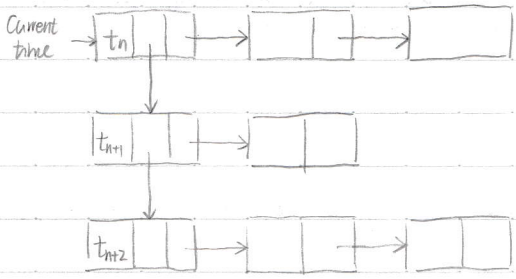
Event-driven Simulation

The objective of event-driven simulation is to minimize the work done by the simulator. Therefore, the state of the circuit is only evaluated when a change occurs in the circuit.

It is possible to predict when the output of an element might change because we know that such a change can only occur after an input changes. If we only monitor the inputs to element, we can only know that an output might change; the logic function of the element determines whether or not a change actually occurs. As we also know the delays through the element, we know when the output might change. Thus an element only needs to be evaluated when it is known that its output might change but not otherwise. Nevertheless, even by predicting a possible changes, it is only necessary to re-evaluate elements when the possible change occur.

By following the possible events through the circuits we can minimize the computation done by the simulator. Only elements that change need to be evaluated.

The delays through elements are defined in terms of integer times. The units of time might be nanoseconds or picoseconds. As the time is incremented in discrete intervals, it is likely that, for any reasonably large circuit, more than one element will be evaluated at any one time. Equally, there may be times at which no elements are due for evaluation. This implies a form of time step control. As each element is evaluated, any change in its output will cause inputs to further elements to change and hence the outputs of those elements may subsequently change. Clearly, it is necessary to maintain a list of which signals change and when. An event is therefore a new value of a signal, together with the time at which the signal changes. If the event list is ordered in time, it should be easy to add new events at the correct place in the further.



A suitable data structure for the event list

When an event is predicted, it is added to the list of events at the predicted time. When an event is processed it is removed from the list. When all the events at a particular time have been processed, that time can be removed.

Zero-width spike

An event is only scheduled if the new value is different from the value that has previously been scheduled for that signal. If two or more events occur on input signals to an element, more than one event may be scheduled for an output signal. It is thus important to know that the new value is not merely different from the present value but also from a value that might be already be scheduled to be set in the future. This algorithm therefore has a disadvantage as it stands because an element is evaluated whenever an event occurs at an input. It is quite possible that two events might be scheduled for the same gate at the same time. This could lead to a zero-width spike (尖峰脉冲) being scheduled one delay later. Even worse, if the delay for rising and falling output differ, the presence or absence of an output pulse would depend on the order in which the input events were processed.

If zero-width pulses are to be suppressed, they can be considered as a special case of the inertial delay model.

Include a PULSE CANCELLATION if a pulse is less than the permitted minimum width

- If an event is predicted at a time less than the inertial delay after the previous event for that node, this new event is not set and the previous event is also removed.

SystemVerilog simulation

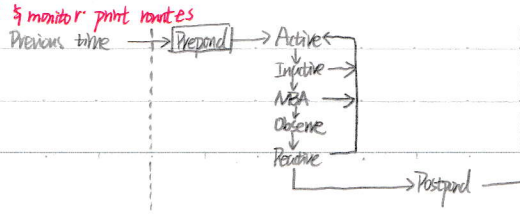
The systemverilog simulation model is based on upon the selective trace algorithm. The LRM (Language Reference Manual) describes a stratified event queue in which the event list is divided into a number of regions.

① Preponed – sample stable values for later checking
② Active – blocking assignments
③ Inactive – zero delay assignments
④ NBA – Nonblocking assignments
⑤ Observe – Evaluate assertions
⑥ Reactive – Execute programs in testbenches → Can create new active events at the current simulation time
⑦ Postpond – $strobe and $monitor print routines

ALLOW Sequential systems to be modeled correctly

any non-blocking assign updates are done
The evaluation of the right hand side of all nonblocking assignment is always done before

Events being explicitly delayed



No.

Date

Reusable

Assertion

```
property Bill;
  @(posedge clock) ready && (twenty || ten) |=> bill;
endproperty
```

Symbol |=> is a non-overlapping implication. In other words, the condition on the left implies that in the next clock cycle the condition on the right is expected to become true.

Symbol |-> is a overlapping implication. The condition on the left implies that the condition on the right is true in the same clock cycle.

```
property BillNotDispense;
  bill |-> !dispense;
endproperty
```

If the vending machine is waiting for bill, it is not dispensing a ticket at the same time

Vacuous Pass

The implication operator means that property fails if bill is true and dispense is also true.
The property passes if bill is true and dispense is false
The property also passes if bill is false. This is known as a vacuous pass. In other words, the property passes because it is never really tested.

Potential source of false optimism → Just because none of the assertions used to verify a design fails, it does not follow that the design has been fully tested. Every assertion might have passed vacuously.

↳ One way to check the properties are fully tested is to use a cover statement

⇒ cover property (BillNotDispense)

After a simulation, the number of times the property was checked will be reported, together with the number of passes.

```
property BillDispenseReady;
  bill ##[1:3] dispense |=> ready;
endproperty
```

The symbol ## is used to indicate clock cycles

##[1:3] means "between 1 and 3 clock cycles"

```
property Eventually;
  ready && twenty |-> ##[1:$] dispense;
endproperty
```

##[1:$] means one and infinity clock cycles later. This is known as a liveness property. In general, liveness properties should be avoided. "Eventually" can be a very long time and therefore might never fail. The property can also pass vacuously. So in practice, this property tells us nothing, while appearing to say a lot.

```
property StateAtoD
  (State==A) && twenty |=> (state==D);
endproperty
```

No.

Date

Default parameter

with ↗ Put that into synthesis and see the structure

If we don't have parameter, you cannot synthesize it.

See what Quartus does to it

Use default

Setup time violation

Design for Reliability one-hot scheme

① put a default line there to go back a normal state

unique check whether all states are covered

② Synplify (Directive in Quartus)

unique case (~)
+ default:

Synthesis Tool use to design the actual circuit

cost (duble the size, more complicated)

EXTRA

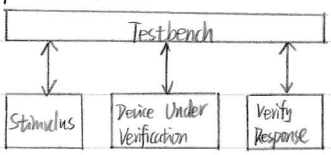
Add more flip-flops doing Hamming code

No.

Date

Modular testbench example

Simple example to illustrate modularity



Top level

```
module testbooth;

parameter N=16;

logic signed [N-1:0] ain, bin;
logic signed [2*N-1:0] qout;
logic clk, load, n_reset;
logic ready;

clock_gen #(10.0, 10.0) c0(.*);
stimulus s0(.*);
booth #(N, N, 2*N) b0(.*);
verify v0(.*);

endmodule
```

Clock generator

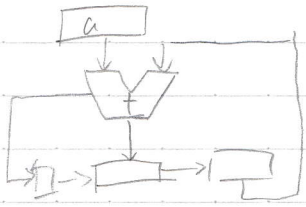
```
module clock_gen #(parameter ClockFreq_MHz=100.0, ResetWidth=10.0)
                  (output logic clk, n_reset);

timeunit 1ns;
timeprecision 100ps;

parameter ClockHigh = (500.0)/ClockFreq_MHz;

initial
  begin
    n_reset = 1;
    clk = 0;
# ResetWidth n_reset = 0;
# ResetWidth n_reset = 1;
forever # ClockHigh clk = ~clk;
  end
endmodule
```

Stimulus

```
program stimulus #(parameter NA=16, NB=16)
                  (output logic signed [NA-1:0] ain,
                   output logic signed [NB-1:0] bin,
                   output logic load);
initial
  begin
    #30ns ain = 12; bin = 410;
    #10ns load = 1;
    #10ns load = 0;
  end
endprogram
```

Verify Assertions

```
module verify #(parameter AL=8, BL=8, QL=AL+BL)
              (input logic signed [QL-1:0] qout,
               input logic ready,
               input logic signed [AL-1:0] ain,
               input logic signed [BL-1:0] bin,
               input logic clk, load);

default clocking clock_clock
@(posedge clk);
endclocking

property LoadReady;
  load |=> ##AL ready;
endproperty

property Product;
  load |=> ##AL (qout == ain*bin);
endproperty

assert property (Load)
else $error("%d * %d gives %d", ain, bin, qout);

endmodule
```



a, b two numbers 8bit × 8bit

Shift bit of b is 0 → Done dd

always-comb

{Cin, Sin} <= a+b

always-ff

~~{Cin, Sin}~~

{C, S, b} <= {Cin, Sin, b[W-2:1]};

A## S<=(A+S)/2

No.

Date

Review

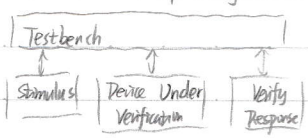
[Q] Explain the difference between synchronous and asynchronous inputs to a sequential system.

Synchronous inputs will affect a sequential system outputs only when a clock edge comes (usually a rising edge). In other words, the inputs are synchronized with the clock.
Asynchronous inputs are independent of the clock, they are dominant over clk and will affect a sequential system outputs immediately.

(Pros) Purposes: High speeds can be achived, as the data path is independent of reset signal
The circuit can be reset with or without a clock present
No work around is required for logic synthesis
The reset signal does not need to be long enough in order to be captured at rising clock edge.
(Cons): Reset line is sensitive to glitches
May have metastability issues

[Q] The circuit might ever enter an unused state because of a set-up time violation. How can this be avoided?
① Use unique case to cover all unused states and use default to set the next state when the circuit goes to a unused state.
③ self-correcting circuit ⑤ Add more flip-flops doing Hamming code.

[Q] Layered Testbench



[Q] Benefits of using Assertion Based Verification
① Providing internal test point in the design
② Simplifying the diagnosis and detection of bugs by localizing the occurence of a suspected bug due to an assertion monitor which is then checked
③ Allowing designers to verify the same assertions using both simulation and formal verification
④ Increasing the observability when simulation is used.
⑤ Increasing both controllability and the observability when formal verification is used.
Assertions help to detect more functional bugs, detect them earlier in the process and detect them closer to their original case
Formulating and writing assertions can give the designer a better understanding of the design and hence uncover bugs in the specification or else avoid introducing bugs into the design in the first place
High verification efficiency

[Q] Principle of Assertion-based verification

P Describe what you expect to see, and when you expect to see it. The simulator will tell you if this doesn't happen.
W An assertion is an expression that is if it is false, indicates an error. Assertions are used for debugging by catching can't happen errors. Within hardware description language designs, an assertion is a conditional statement that checks for specific behaviour and displays a message if it occurs. Assertions are generally used as monitors looking for bad behaviour, but may be used to create an alert for desired behaviour as well.
For our purposes, an assertion is a statement about a specific functional characteristic or property that is expected to hold for a design.

[Q] Principle of Scan-in, Scan-out (SISO)

SISO makes the state variables directly accessible by connecting all the state registers as a shift register. The shifter register has a mode control input M. when M=0, it's in operational mode. when M=1, it's in scan mode, the flip-flop performs a shift register with the input to the shift register being the scan data in (SDI) pin and the output being the scan data out (SDO) pin.

[Q] What features distinguish a SystemVerilog "testbench" from other SystemVerilog modules

① A testbench has no inputs or outputs.
② As testbench is never synthesized, we can use the entire SystemVerilog language
③ Can use initial assignment which indicates a produce that is executed once

CPLL: Channel PLL

PMA: Physical medium attachment

Verilog (|7) 归约或 bit 进行 or

No.

Date

Primitive 是针对具器件特征开发的一有列常用便块的名字，可以将其直接一个库函数

UG578 Notes

The GTY Transceivers support line rate from 500Mb/s to 32.75Gb/s in UltraScale + FPGAs.

Each GTY transceiver channel in a Quad has six clock inputs available: ① Two clock local clock reference clock pin pairs, GER GTREFCLK0/1 ②Two reference clock pin pairs from the Quads above, GTSOUTHREFCLK0/1 ③…from Quads below, GTNORTHREFCLK0 or GTNORTHREFCLK1

TXSYSCLKSEL: Selects the PLL reference clock source to drive the TXOUTCLK

TXPLLCLKSEL: Selects the PLL to drive the TX datapath { 00: CPLL / 10: QPLL1 / 11: QPLL0

QPLL0/1 Nominal VCO Operating Range { QPLL0 9.8-16.375 / QPLL1 8.0-13

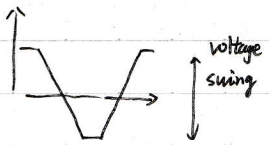
$f_{PLLCLKout} = f_{PLLCLKin} \times \frac{N.FractionalPart}{M \times QPLL\_CLKOUTRATE}$ $f_{LineRate} = \frac{f_{PLLCLKout} \times 2}{D}$ $FractionalPart = \frac{SDMDATA}{2^{SDMWIDTH}}$

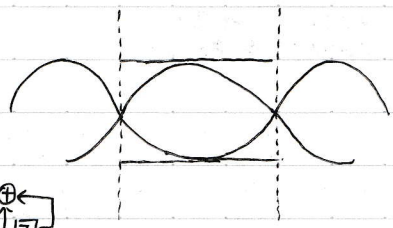
PCS PSA & PMA

The physical coding sublayer (PCS) is a networking protocol sublayer, it resides at the top of the physical layer (PHY), and provides an interface between the physical medium attachment (PMA) sublayer and the media independent

DRP Dynamical Reconfiguration Port

IBERT: Integrated Bit Error Ratio Test





PRBS

PRBS7 = x^7+x^6+1

PRBS9 = x^9+x^5+1

PRBS11 = $x^{11}+x^9+1$

PRBS15 = $x^{15}+x^{14}+1$

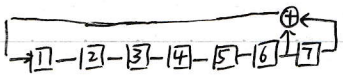
PRBS20 = $x^{20}+x^3+1$

PRBS23 = $x^{23}+x^{18}+1$

PRBS31 = $x^{31}+x^{28}+1$

↑ poly_length ↖ poly_tap

e.g: PRBS:



2^7-1: 127-bit long

[1] [2] [8]

reg [1:8] prbs = {8'b10101010};

Xilinx Code: reg [1:POLY_LENGTH] prbs_reg;

check_mode == 0?

Stimulus: data_in = 10

Checking: data_in = rx data

[64:1] prbs_msb

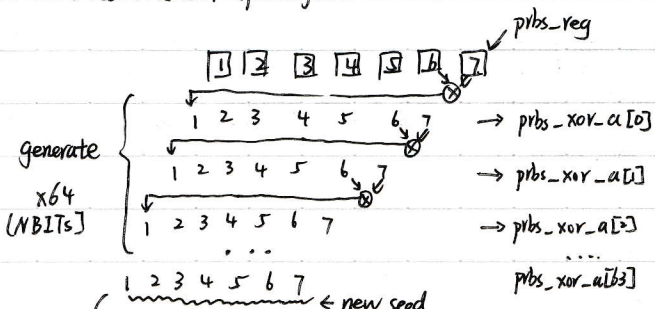
prbs[0] = prbs_reg{1'b1}

xor_a[0] = prbs[0][6] ^ prbs[0][7]

(>>1) prbs[1] = {prbs_msb[1+1], prbs[0][1:POLY_LENGTH-1]}

输入R LFSR的 Seed 为 {1'b1}，NBITS 设置每周期产生的 PRBS 的 bit 数，并通过 DATA_OUT 输出 NBITS PRBS bits.

每周期后，Seed 会随更新（prbs_reg 的值）



check mode 下配产生的 prbs bits

RX 接收的 prbs

⊗ ⇒ Data_in ⇒ xor_b 作为 Data_out 输出

产生噪输出噪

非 check mode 下，data_in = 0

check mode 下，产生误码位并上层通过判断误码位是否有在 | (|check)

由于 PRBS 仅有 2^7-1 bits 的长度，fpga 中 prbs 生成器每周期生成 64-bit，因此 seed 并不会每2个周期重复出现

Original seed: (7F)

After 2 clk: 7F → (3F)

4: 7F → 3F → (1F)

6 clk: 7F → 3F → 1F → 0F

No.

Date

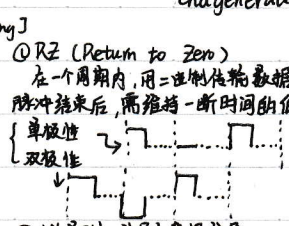
[Verilog Generate]

```
genvar I;
generate for (I=0 ; I<NBITS; I=I+1)
begin : g1
   ...
end
endgenerate
```

[Coding]

① RZ (Return to Zero)

在一个周期内，用二进制传输数据，在数据位脉冲结束后，需维持一断时间的低电平



同时传递时钟信号和数据信号

归零码高占用一部分带宽，效率低

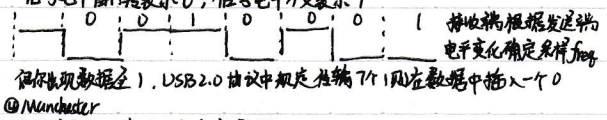
② NRZ (Non Return Zero)

最常见，HIGH-1，LOW-0 e.g. UART

③ NRZI (Non Return Zero Inverted)

既能传输时钟信号，又能尽量不损失系统带宽

信号电平翻转表示0，信号电平不变表示1

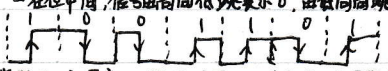


Bit Stuffing ⇒ 保存收到数据全1，USB2.0协议中规定传输7个1则在数据中插入一个0

④ Manchester

利用信号的跳变方向来决定数据

一在位中间，信号由高向低跳表示0，由低向高跳表示1



NRZ和NRZI都没有自同步特性，但是有一些特殊技巧，比如发送一个同步头(010101)的方波，让接受者通过这个同步头计算着发送者的频率，再利用这个频率采样之后的频率

[8b/10b]

#DC bias: the DC bias, DC component or DC coefficient is the mean value of the waveform waveform

If the mean amplitude is zero, there is no DC offset. A waveform without a DC component is known as DC-balanced or DC-free

#8b/10b encoding: In telecommunication, 8b/10b is a line code that maps 8-bit words to 10-bit symbols to achieve DC-balance and bounded disparity and yet provide enough state changes to allow reasonable clock recovery. This means that the difference between the counts of ones and zeros in a string of at least 20 bits is no more than two, and there are not more than five ones or zeros in a row. This helps to reduce the demand for the lower bandwidth limit of the channel necessary to transfer the signal.

[Phase-Locked Loop]

锁相环是一个相位反馈自动控制系统

① 压控振荡器的输出经过采集并分频

② 和参考信号同时输入鉴相器

③ 鉴相器通过比较上述两个信号的频率差，输出一个直流脉冲电压

④ 控制 VCO，使它的频率改变

⑤ 经过一个很短的时间，VCO输出稳定于某一期望值.

#Dynamic Reconfigurable Port

The DRP allows the dynamic change of parameters of the GTV primitives

The DRP interface: ① Address bus (DRPADDR)

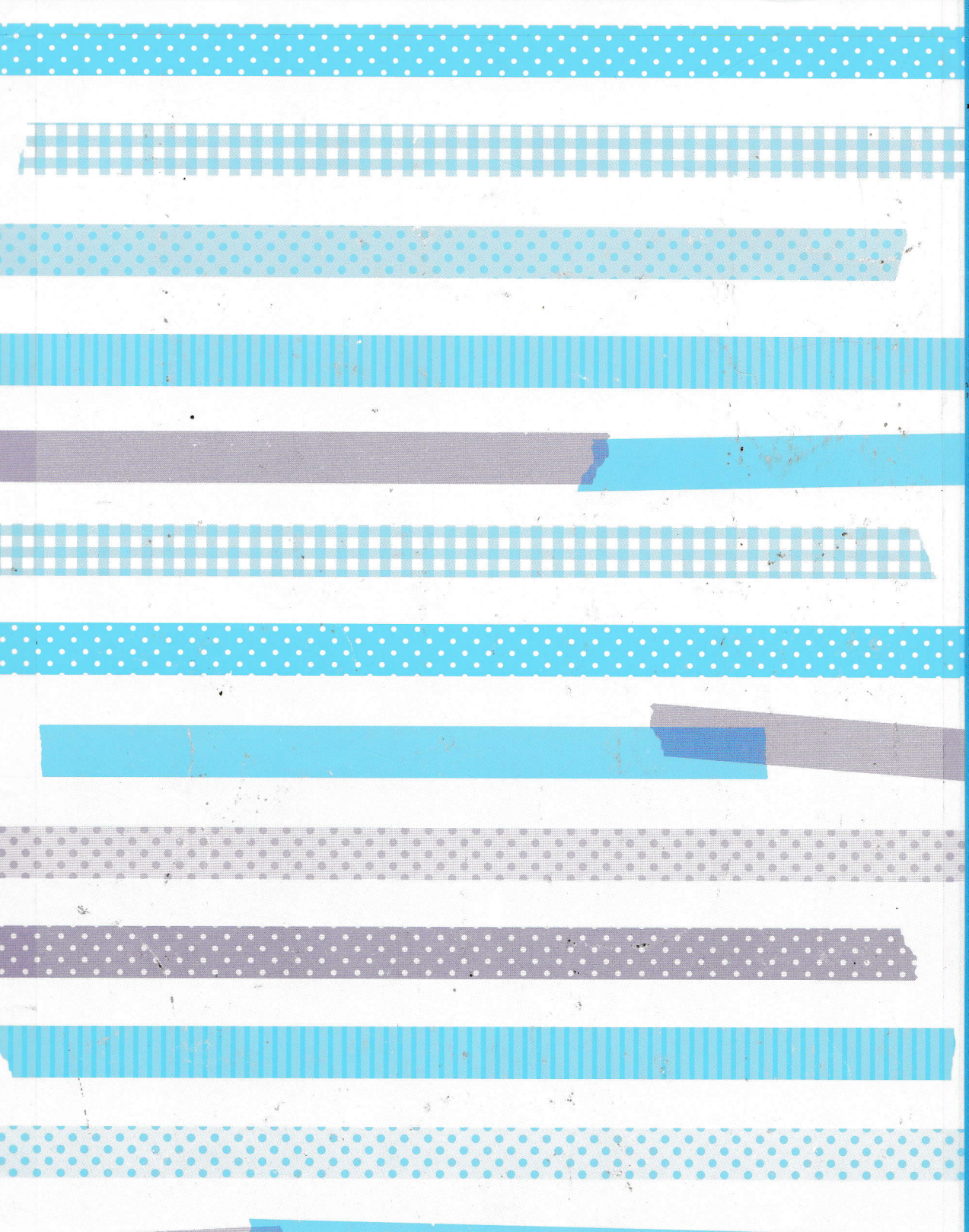
② Separated data buses for {reading (DRPDO) / writing (DRPDI)}

③ Enable signal (DRPEN)

④ Read/write signal (DRPWE) implement read/write operations.

⑤ Ready/valid signal (DRPRDY) indicate operation completion/availability of the data

I	DRPADDR [9:0]	DRP Address Bus
I	DRPCLK	DRP Interface Clock
I	DRPEN	DRP Enable signal (0: No read or write operation performed)
I	DRPDI [15:0]	Data bus for writing configuration data from the interconnect logic resources to the transceiver
O	DRPRDY	Indicates operation is complete for write operations and data is valid for read operations
O	DRPDO [15:0]	Data bus for reading config data from the GTV transceiver to the interconnect logic resources
I	DRPWE	DRP write enable (0= read operation when DRP is 1)



Campus 无线装订本
B5 40页
WCN-CNB1430
6 937748 309932
MADE IN CHINA

国誉商业(上海)有限公司 ·
http://www.kokuyo.cn/st/
上海市奉贤区人杰路128号
TEL：400-820-0798　FAX：021-3255-8508
产地：上海市　QB/T1438-2007合格
B5　252×179mm

● 采用日本进口纸张，纸质细滑，牢固不掉页。