一、FPGA 开发概述

•理解 FPGA 的优势和开发流程

•熟悉 Vivado 工作环境

•清楚 Behavioral Simulation 和 IO planning 的作用

二、基本语法

数字声明（Number Specification）的方法，位宽、进制的表示方法。

<位宽>'<进制> <数字> ，例：4b'0011，10'hA3

位宽缺省 32 位，进制缺省十进制(二进制 b，十进制 d，八进制 o，十六进制 h)

其中位宽指的是转成二进制后的宽度，例如 10'hA3 转成二进制后是

10'b00_1010_0011 (_不能在第一位)

其中下划线_是为了增强数字的可读性，可以放在除了第一位的任何一位。

这里十六进制数 3A 转成二进制数不满 10 位，所以最高的两位要补 0

如果是：6'hA3，转成二进制是 6'b10_0011，最高的两位溢出舍弃

Verilog操作符

类型	符号	功能说明	类型	符号	功能说明
算术运算符 （双目运算符）	+ − * / %	二进制加 二进制减 二进制乘 二进制除 求模	关系运算符 （双目运算符）	> < >= <= == !=	大于 小于 大于或等于 小于或等于 等于 不等于
位运算符 （除~是单目， 其余为双目运 算符）	~ & | ^ ~^ 或 ^~	按位取反 按位与 按位或 按位异或 按位同或	缩位运算符 （单目运算符）	& ~& | ~| ^ ^~ 或 ~^	缩位与 缩位与非 缩位或 缩位或非 缩位异或 缩位同或
逻辑运算符 （除!是单目， 其余为双目运 算符）	! && ||	逻辑非 逻辑与 逻辑或	移位运算符 （双目运算符）	>> <<	右移 左移

位拼接运算符{}

格式：

{信号 1 的某几位，信号 2 的某几位，...,...,信号 n 的某几位}

例：a=1，b=4'b1001，c=8'b11100110，d 为 6 位数，

d={a，b[1:0]，c[5:3]}= 6'b101100

变量的种类和声明方式

net 类型：主要是 wire 型 声明方式：wire < 位宽> 变量 1，变量 2，…

寄存器类型：主要是 reg 型 声明方式：reg < 位宽> 变量 1，变量 2，…

什么时候 reg 类型，什么时候 wire 类型

连续赋值语句: assign a = 1'b1; 这里的 a 必须是 wire 类型

过程赋值语句：b <= 1'b1; 这里的必须是 reg 类型
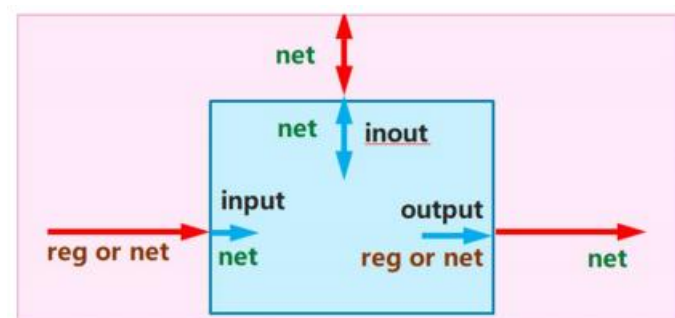
输入输出口缺省状态是 wire 类型

三、实例化调用 Instantiation

实例化调用时的语法格式：<module name> <instantiation name> port list

在一个模块中实例化引用另一个模块时的端口连接规则，什么情况下是 net 类型，什么情况下是 reg 类型，能够识别非法的端口连接和正确的端口连接。
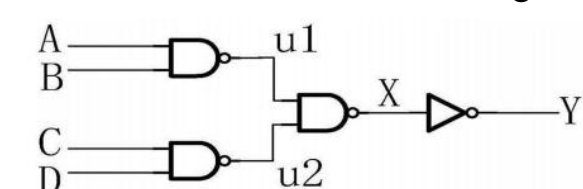
在 port list 中为端口连接外部信号时，有顺序连接方法与按名字连接的方法

四、组合逻辑电路设计

给定组合逻辑电路，能用三种建模方式来描述这个电路：

门级建模 Gate-level modeling：primitive (原语)

数据流建模 Dataflow Modeling: Continuous assignment（连续赋值语句 assign）

行为级建模 Behavioral Modeling: Procedural block （过程块 always，initial）

熟记常用的 primitive；

使用 primitive 建模时，要注意 port list 的顺序：输出口在前，输入口在后，例：

nand r1 (u1, A, B) ; //这里的 r1 是实例名，使用原语时可省略

五、时序逻辑电路设计

给出功能表，能根据功能表设计时序逻辑电路

如各种触发器 flip-flop（JK，D，T，…），counter（Johnson，BCD，…），shift register

注意 synchronous input 和 asynchronous input 的区别（依赖时钟，不依赖时钟），注意在 always 引导的过程块中如何表示这种区别。

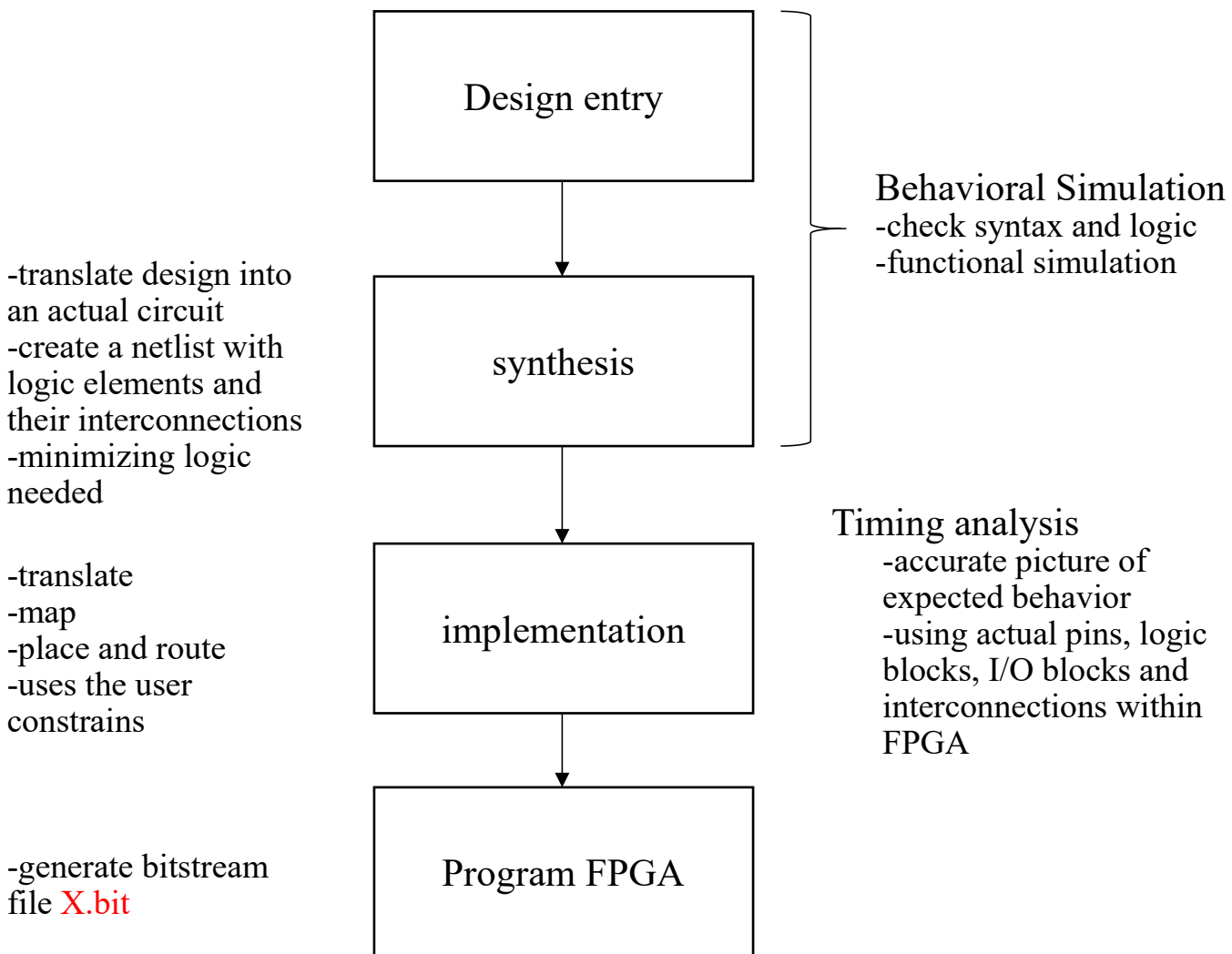能写 Testbench 验证所设计的 module

// Review

   --PLD→FPGA

   --use FPGAs because

      -speed up design process

      -reduced cost

      -increase design and operational efficiency

      -reuse/adapted/upgraded because of programmability

FPGA design flow   Verification

```
┌─────────────────────────┐
│                         │
│      Design entry       │
│                         │
└─────────────────────────┘
```
                                          Behavioral Simulation
                                          -check syntax and logic
-translate design into   -functional simulation
an actual circuit
-create a netlist with
logic elements and
their interconnections
```
┌─────────────────────────┐
│                         │
│       synthesis         │
│                         │
└─────────────────────────┘
```
-minimizing logic
needed
                                          Timing analysis
-translate   -accurate picture of
-map   expected behavior
-place and route   -using actual pins, logic
-uses the user   blocks, I/O blocks and
constrains   interconnections within
```
┌─────────────────────────┐
│                         │
│     implementation      │
│                         │
└─────────────────────────┘
```
                                             FPGA

-generate bitstream
file X.bit
```
┌─────────────────────────┐
│                         │
│      Program FPGA       │
│                         │
└─────────────────────────┘
```

C function

int main (   )
{

}

Verilog Module

module main (   );

endmodule
```
                          3
```

Verilog Syntax Classification

--Definitions
- -nets, registers, tasks, and functions
- //nets and registers are common in use
- //nets: wires, a way to transfer/transmit the value (logic 0/1)
- //registers: data storage/memory

--Declarative Code
- -continuous assignments
- -module (and primitive) instantiations
- //declarative code describe the functionality and structure of system
- //continuous assignments: to design combinational circuits- no storage; time-independent

--Procedural Code
- -initial and always blocks
- //to describe the functionality of both combinational and sequential circuits-memory (from registers)

Verilog nets

--Nets are used to connect model components together (like wires in a circuit)

net_type [range] [delay3] list_of_net_identifiers;
//[range]: [most significant code: most least code]
//[delay3]: propagation delay, non-synthesizable code, only for simulation
wire w1, w2, w3; //create 3 single bit wires named w1, w2, w3
wire [7:0] w4, w5; //create 2 8-bit wide wires named w4 and w5
wire [31:0] #5 _test; /*create 1 32-bit wide wire named _test that is simulated with 5 units of propagation delay*/
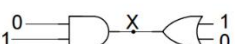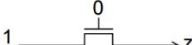**reg [63:0] mem;** //create 1 64-bit register named mem
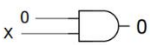
--Net types
wire, tri, tri0, tri1, wand, wor, triand, trior, supply0, supply

--Identifiers
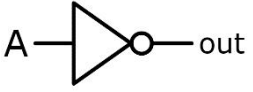Comprised of any number of letters, digits, "_", and "$"
First character must be alphabetic or "_"

Verilog Primitive Modeling
gate structure

AND		Y=A.B	

OR		Y=A+B	
NOT		Y=A'	
NAND		Y= (A.B)'	If at least one of the input is 0, then Y is '1'. Opposite to AND gate
NOR		Y= (A+B)'	If at least one of the input is 1, then Y is '0'. Opposite to OR gate
XOR		Y=A ⊕B =A'B+AB'	When both inputs are same, the output is '0'.
XNOR		Y=A⊙B =AB+A'B'	When both inputs are same, the output is '1'.

Vivado

• 2 modules within the file Example1.v
• top module: the module that will be synthesized
• Not top module: must make it top module in order to synthesize it (right-click and select the "set as top" option)

Verilog Primitive Modeling

--prim instance_name (out(s), in(s));

Vivado
2 modules within the file Example1.v
top module: the module will be synthesized
non-top module: must make it as the top module to synthesis it

3-input majority circuits

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

5

| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

X(SOP)=a'bc+ab'c+abc'

K-map:

c \ ab	0	1
00	0	0
01	0	1
11	1	1
10	0	1

X=ab+ac+bc

Testbench Using Verilog

Verilog testbench properties

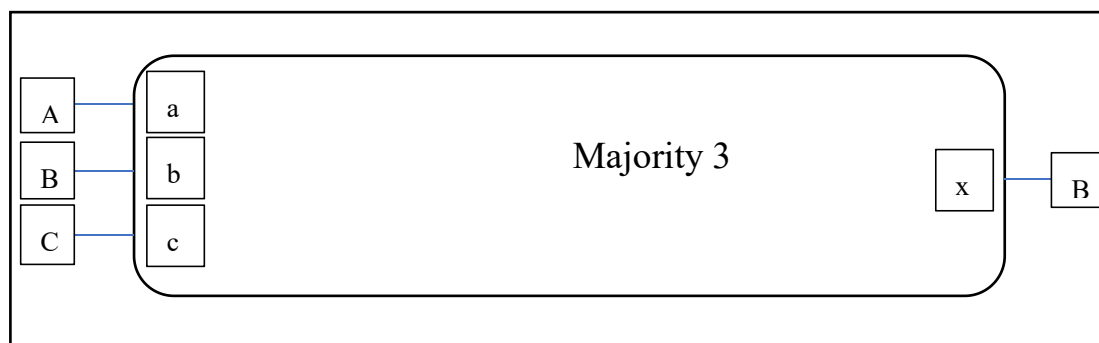--contain a teat module

    -no I/O ports

    -include registers to hold test values to input into the module being tested

    -include wires to monitor the output(s) of the module being tested

  •Synthesis successfully only means the code can be configured in hardware into the net list to fit into the FPGA, but Test can perform if it's functionally correct

Tester

//A, B, C: registers

    -used to provide input test values

//X: wire

    -used for monitoring

-to perform testing of module, update the values stored in the input registers and read the resulting output values on the output wires

Primitive Modeling: realize a digital circuit using logic gate primitives

    -require the circuit schematic

    -describing physical structure of combinational circuit

Dataflow Modeling: realize a digital circuit using Boolean expressions

    -require the Boolean expressions

-describing the functionality of complex combinational circuit

Verilog Numbers

number_of_bits'base_identifier digits

e.g.

4'd5 ➜ 0101

8'b101 ➜ 0000 0101

12'h58B_3 ➜ 0101 1011 0011 //'_' is no sense, just for readable

12'Hxa ➜ xxxxxxxx1010 // x means unknown

596 ➜ 1001010100 /*default base is decimal and numbers of the bits will match whatever is needed for the number*/

2'hE ➜ ~~11~~00 //11 would be truncated

wire [3:0] w1; /*How to set w1 to 5? Take the w1=1010 as example. Number on that base is decimal, and w1=1111110010(b). But w1 is only 4-bit, so w1=0010. */

/* w1=5; don't use this format, use w1=4'd5 or w1=4'b0101. */

Verilog Operators

Basic	OPERATION	DESCRIPTION	RESULT
Arithmetic	+ - * / %	Basic arithmetic	Multi-bit
Relational	> >= < <=	compare	One-bit

//% is modules, which is remainder from integer division.

// RESULT

e.g. 4'b1010+4'b0101→4'b1111

    //4'b1010 and 4'b0101 are operands, and 4'b1111 is result

    //multi-bit operands → multi-bit result

    //result size will match operand size in most cases excepted it's set.

//Relations

    //used in behavioral modeling

    //Relation's result is 1 bit and the T is 1, while F is 0.

Equality	OPERATION	DESCRIPTION	RESULT
Logical	== !=	Equality not including Z, X	One-bit
Case	=== !==	Equality including Z, X	One-bit

//Logical is like relational and checking for equality, and the result is 1 bit and the T is 1, while F is 0.

//Case will compare the operands bit by bit. If the bases are different, the result must be F.

Boolean	OPERATION	DESCRIPTION	RESULT
Logical	&& ‖ !	Simple logic	One-bit
Bit-wise	~ & | ^ ^~ ~^	Vector logic operation	One-bit
Reduction	& ~& |~| ^ ^~ ~^	Perform operation on all bits	One-bit

Bitwise operators and reduction operators are all represented by the same notation. The key to distinguishing these operators is the number of operands.

//&& is AND, ‖ is OR and ! is NOT.

    If an operand is not 0 it is equivalent to logic 1. If an operand is equal to 0, it is equivalent to logical 0. If any of its bits are x or z, it's the same thing as x.

//~ is NOT, & is AND, | is OR, ^ is XOR, and ~^or^~ is XNOR. They are for each operand bit.

//& is reduction AND, ~& is reduction NAND, | is reduction OR, ~| is reduction NOR, ^ is reduction XOR, and ^~or~^ is reduction XNOR.

Reduction uses a single operand and the result is for whole operand.

//E.G.

logical 4'b0110 && 4'b0 →0(F)

       4'b0110 ‖ 4'b0 → 1(T)

       4'b0110 && 4'b1000 → 1(T)

       //any non-zero value is True(1)

bitwise ~4'b0001 → 4'b1110

       4'b1001 & 4'b0111 → 4'b0001

       4'b1101 ^ 4'b0100 → 4'b1001

reduction &4'b1011 →1&0&1&1 →0(F)

       &4'b1111 → 1&1&1&1 →1(T)

       ~|4'b1001 → 1|~0|~0|~1 →0(F)

       ^4'b1001 → 1^0^0^1 →0(F)

Shift	OPERATION	DESCRIPTION	RESULT
Logical Right	>>n	Zero-fill Shift n places	Multi-bit
Logical Left	<<n	Zero-fill Shift n places	Multi-bit
Arithmetic Right	>>>n		Multi-bit
Arithmetic Left	<<<n		Multi-bit

//Logical shift's key is 0-fill shift n places.

shifting 4'b1001 <<1 → 4'b0010

       4'b1001 <<3 → 4'b1000

       4'b1101 >>1 → 4'b0110

Verilog Assignments

Continuous assignments

assign [delay3] list_of_net_assignments;

-Delay is propagation delay and for simulation only.

-Left-hand side of assignment must be a net.

-Continuous assign statements are not executed in source order.

E.G.

wire a, b, c;

① assign c=a/b;

② assign a=1'b1;

③ assign b=1'b0;

//Example: create a parity bit circuit(奇偶位电路)

TX(transmitter) → RX(receiver)

//8-bit data→provide error detection in the transmit and receive process using a parity bit

10101010

↓ 8-bit data

10101010_   //'_' is the parity bit

-set by TX and read by RX

-Even: set parity bit to a 1/0 so that the total number of 1s transmitted (data + parity bit) is even

-Odd: set parity bit to a 1/0 so that the total number of 1s transmitted (data + parity bit) is odd

//Back to the example, 101010101→Odd, 101010100→Even.

101010101//transmitted with odd parity

↓

001010101//receiver, count 4 1s which is even, therefore data is in error

//Traditional design process

1) Truth table with 8 inputs and parity bit outputs

2) Generate a Boolean expression

3) Minimize

data-in(8-bit)→ Even Parity →data-out(9-bit)

-use reduction XOR operator to reduce 8-bit data to an Even Parity bit

XOR: 10101010→1^0^1^0^1^0^1^0→0(even parity bit)

      11100000→…→1(even)

module Even Parity (data_in, data_out);

input [7:0] data_in;

output [8:0] data_out;

assign data_out [8:1] =data_in;

assign data_out[0] =^data_in; //store parity bit into LSB

endmodule

//Sometimes the parity bit is after the most significant bit, and sometimes it's before the least significant bit. It would be determined by the transmitter and receiver on how they place.

--EvenParity module testbench

EvenParity UUT(.data_in(d_in),.data_out(d_out));
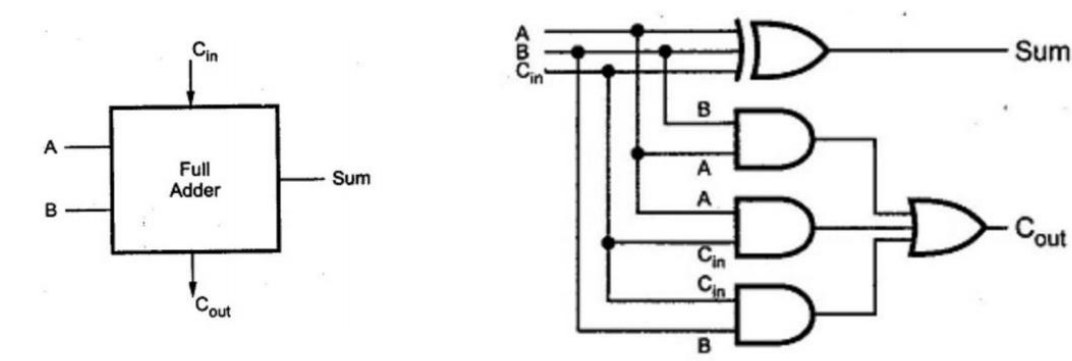
// **EvenParity** is the module being instantiated.

//**UUT** is the instance name.

//**.date_in/.date_out** are port names from the module EvenParity.

//**d_out/d_in** are net names from the testbench.

//Above are dot notation to directly connect the ports.

-Full Adder Example

A+B+Cin → Cout+Sum, if A and B are n-bit wide, Sum is also n-nit wide. The result is (n+1)-bit wide.

If Sum=A+B+Cin, only capture the least significant m-bit and store to Sum.

{Cout, S}=A+B+Cin; //capture n+1 bits and store to Cout and S
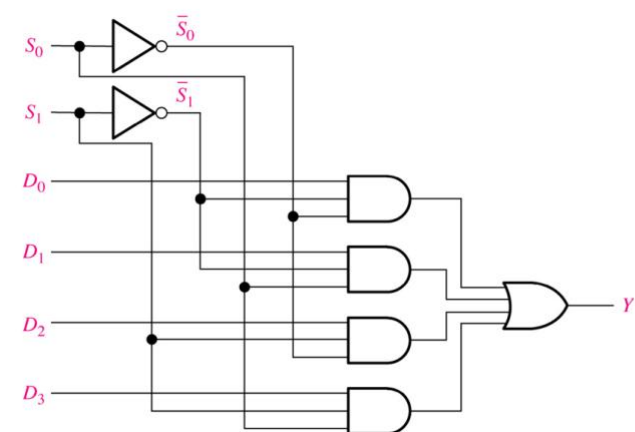
//{} is concatenation operator.

//Cout is msb and S is lsb

A	B	C_{in}	C_{out}	S
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	1	0
0	0	1	0	1
1	0	1	1	0
0	1	1	1	0
1	1	1	1	1

-4:1 Mux Example

S1	S0	Y
0	0	
0	1	
1	0	
1	1	

Y=S1'S0'D0+S1'S0D1+S1S0'D2+S1S0D3

```
//use bitwise operators
module Mux4_DF(S,D,Y);
input [3:0] D;
input [1:0] S;
output Y;


assign Y=(~S[1]) & (~S[0]) & (D[0]) |
         (~S[1]) & (S[0]) & (D[1]) |
         (S[1]) & (~S[0]) & (D[2]) |
         (S[1]) & (S[0]) & (D[3]);


endmodule
```

Behavioral Modeling

- Logic circuit described based on behavior of outputs with relation to the circuit inputs
  - Highest level of abstraction in Verilog modeling of logic circuits
  - Requires logic synthesis to create gate-level logic circuit

- Commonly used with sequential logic circuits

- Gate realization not needed prior to modeling

- just look at the behavior of the circuit

- highest level of abstraction

- Commonly used with sequential logic circuits

reg [range] list_of_register_identifiers;

//**reg**: reg types

//**[range]**: [msb:lsb]

-procedural blocks //assign values to registers

    -initial

    -always @ (sensitive/trigger list)

        -trigger on the change in level of a signal

            always @ (a)

            always @ clr

            -always blocks triggered when a or clr change values, 0→1 or 1→0

        -trigger on a specific edge of a signal

            always @ (posedge clk)
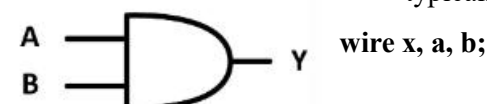
            always @ (negedge clk)

            - always blocks triggered @ rising edge of 'clk' or falling edge of 'rst'

E.G.

-level change triggered

        -typically used to model combinational logic

wire x, a, b;

assign x=a&b; //anytime the value of a or b changes, x is updated(continuously), the following is different

wire a, b;

reg x;

always (a, b) //anytime the value of a or b changes, the procedural block is executed(x is updated)
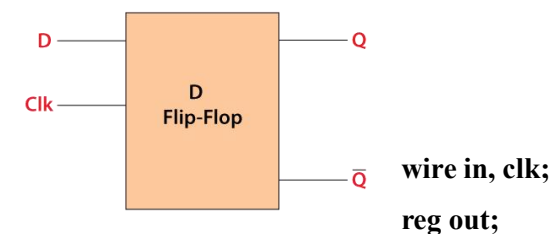
begin

    x=a&b;

end

-edge triggered

    -typically used to model sequential logic

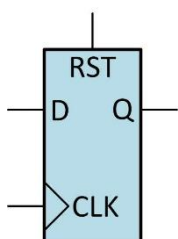**wire in, clk;**

reg out;

always @ (posedge clk) //state only updates on positive edge of clk signal, not when input changes

begin

    out=in;

end

//RST: asynchronous reset, -logic low makes Q low

//read-list:https://lambdageeks.com/d-flip-flop-circuit-working-truthtable-differences/

wire in, clk, rst;

reg out;

always @ (posedge clk, negedge rst)

begin

    out= D & rst;//rst fall, out=0

end

Verilog Procedural Blocks

Only registers can be assigned within a procedural block.

-always: loops for continuous execution

-initial: execute only once at time zero, and it's by synthesizer

//Multiple procedural statements within the body of a procedural block must be bracketed by **begin** and **end** keywords. But when the

begin – **end** bracketing encloses only one procedural statement it is not mandatory.

-**initial** Statements

initial begin

```
        clk = 0;
        reset = 0;
        req_0 = 0;
        req_1 = 0;
end
```

//**clk, reset, req_0,** and **req_1** are all registers.

//**initial** means at time 0, all values are 0.

- always assignment

·   An **always** block should have a sensitive list(trigger) or a delay associated with it

·   The sensitive list tells the always block when to execute the block of code

·   The @ symbol after **always** indicates that the block will be triggered "at" the condition in parenthesis

Procedural blocking assignments (=)

-not executed by FPGA, but by Vivado during synthesis into hardware

-A blocking assignment uses a reg data type on the left-hand side and an expression on the right-hand side of an equal sign
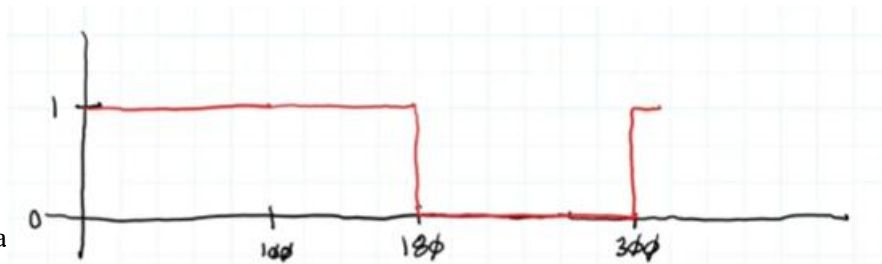
-May contain intra-assignment delay or event control

E.G. Blocking (=)

```
…
initial begin:
b=1;
#100; //delay
b=#80 0;
b=#120 1;
…
```


Procedural nonblocking a

-When flow reaches a non-blocking assignment, the right-hand side of the assignment is evaluated and will be scheduled for the left-hand side reg to take place when the intra-assignment control is satisfied

E.G. nonblocking

```
…
initial begin:
a<=1;
#100; //delay
a<=#80 0;
a<=#120 1;//start at sa
…
```


E.G.

reg A, B;

initial

begin

        A=0

                        end
                        always @ (posedge clk)

```verilog
        begin
            A=~A;
            B=A;
        end //A=1,B=1
        or

        reg A, B;
        initial
        begin
            A=0
end
always @ (posedge clk)
begin
    A<=~A;
    B<=A;
end //A=1,B=0
```
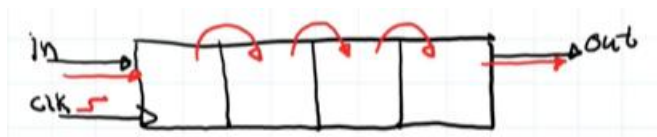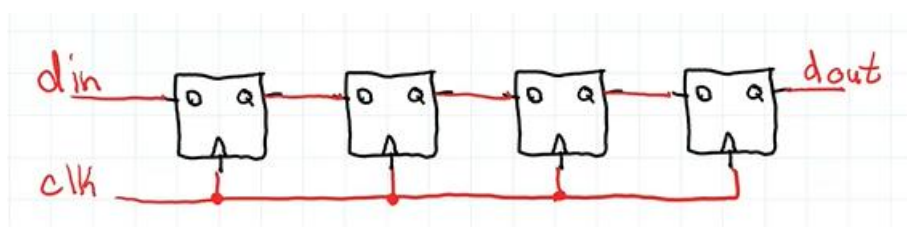
//5-bit serial-in serial-out shift register

NON-BLOCKING


```verilog
module SR4_NB(din, dout, clk); //4-bit shift register non-blocking
input din, clk;
output dout;

reg [3:0] Q;
always @ (posedge clk) //model the shifting within the shift register
begin
    Q [0] <=din;//serial inputs
    Q [1] <=Q [0];
    Q [2] <=Q [1];
    Q [3] <=Q [2];
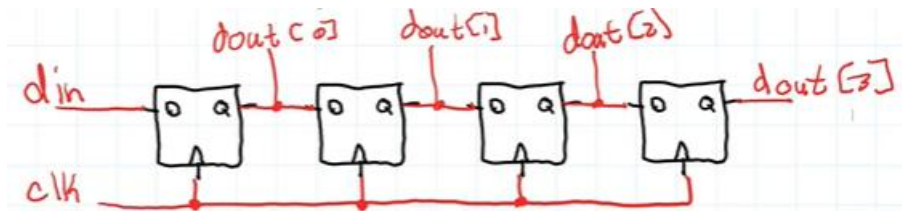```

**end**

**assign dout=Q[3];** //serial output updates when Q[3] value changes

**endmodule**

**module Tester();**//testbench to test the non-blocking SR4

**reg din, clk;**

**wire dout;**

**SR4_NB UUT (.din(din), .clk(clk), .dout(dout));**

**initial**

**begin**

**clk=1'b0;**

**din=1'b1;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**end**

**endmodule**



-show the [3:0] dout values

**module SR4_NB(din, dout, clk);** //4-bit shift register non-blocking

**input din, clk;**

**output [3:0] dout;**

**reg [3:0] Q;**

**always @ (posedge clk)** //model the shifting within the shift register

**begin**

    **Q[0]<=din;**//serial inputs

    **Q[1]<=Q[0];**

    **Q[2]<=Q[1];**

    **Q[3]<=Q[2];**

**end**

**assign dout=Q;** //serial output updates when Q[3] value changes

**endmodule**


//Tester

**module Tester();**//testbench to test the non-blocking SR4

**reg din, clk;**

**wire [3:0] dout;**

**SR4_NB UUT (.din(din), .clk(clk), .dout(dout));**

**initial**

**begin**

**clk=1'b0;**

**din=1'b1;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

**#5;**

**clk=~clk;**

```
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
end
endmodule
```


-initialize shift register

```
module SR4_NB(din, dout, clk); //4-bit shift register non-blocking
input din, clk;
output [3:0] dout;

reg [3:0] Q;
initial
begin
Q=4'b0; //initialize shift register to be clear
end
always @ (posedge clk) //model the shifting within the shift register
begin
    Q[0]<=din;//serial inputs
    Q[1]<=Q[0];
    Q[2]<=Q[1];
    Q[3]<=Q[2];
end
assign dout=Q; //serial output updates when Q[3] value changes
endmodule
```
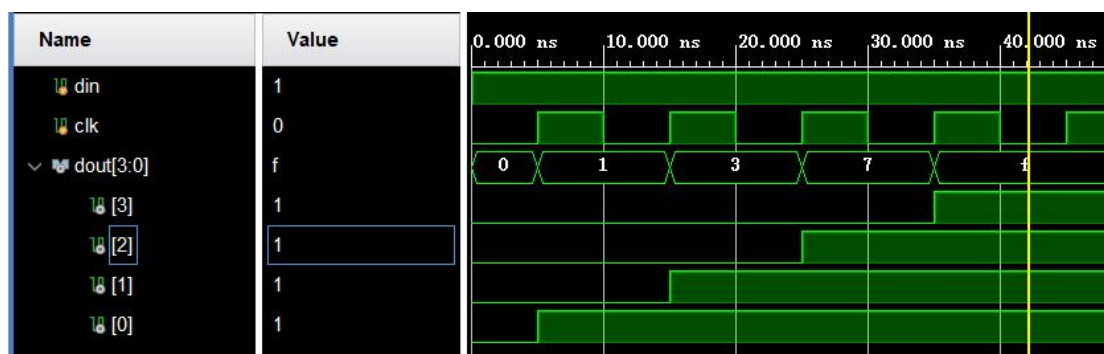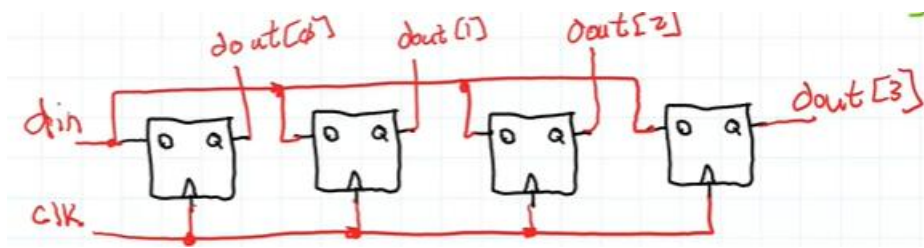
//Tester is as same as the second

Blocking


```verilog
module SR4_B(din, dout, clk); //4-bit shift register non-blocking
input din, clk;
output [3:0] dout;
reg [3:0] Q;
initial
begin
Q=4'b0; //initialize shift register to be clear
end
always @ (posedge clk) //model the shifting within the shift register
begin
    Q[0]=din;//serial inputs
    Q[1]=Q[0];
    Q[2]=Q[1];
    Q[3]=Q[2];
end
assign dout=Q; //serial output updates when Q[3] value changes
endmodule

//Tester
module Tester();//testbench to test the non-blocking SR4
reg din, clk;
wire [3:0] dout, dout2;
SR4_NB UUT (.din(din), .clk(clk), .dout(dout));
SR4_B UUT2 (.din(din), .clk(clk), .dout(dout2));
initial
begin
clk=1'b0;
```
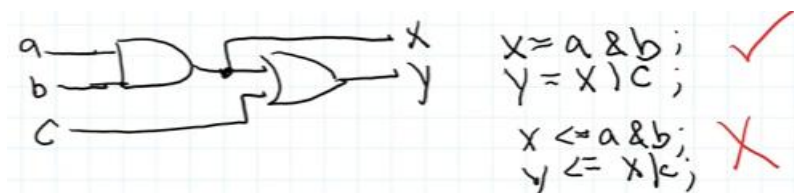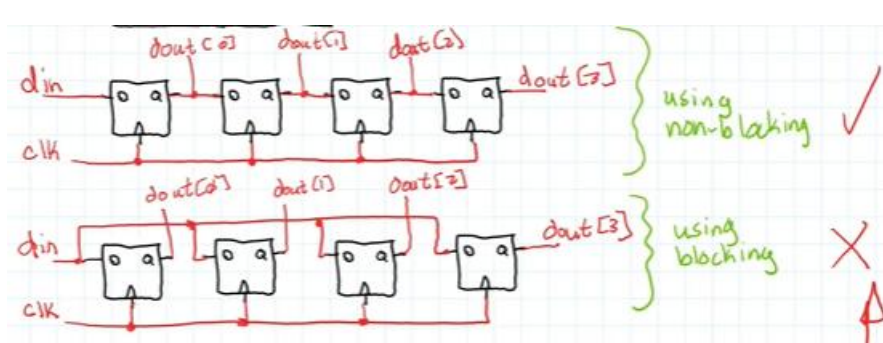
```verilog
din=1'b1;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
#5;
clk=~clk;
end
endmodule
```


--combinational circuits→blocking assignments

$$x = a \& b;$$
$$y = x \mid c;$$ ✓

$$x <= a \& b;$$
$$y <= x \mid c;$$ ✗

--sequential circuits→non-blocking assignments

19

-If several assignments appear at the same real time in a procedural body, the last assignment overrides all others.

-If program flow in two procedural bodies reaches assignments to the same reg at exactly the same time (multi-sources/race condition, which should be avoid), the outcome of the value assigned to the left-hand side of the assignment will not be known.

-use **always** to Test

module Tester();//testbench to test the non-blocking SR4

reg din, clk;

wire [3:0] dout, dout2;

SR4_NB UUT (.din(din), .clk(clk), .dout(dout));

SR4_B UUT2 (.din(din), .clk(clk), .dout(dout2));

initial

begin

    clk=1'b0;

    din=1'b1;

end

always//run continuously to generate clokck signal

begin

    clk=~clk;

    #5;

end

endmodule

-reduce the chance of race

```verilog
module Tester();//testbench to test the non-blocking SR4
reg din, clk;
wire [3:0] dout, dout2;
SR4_NB UUT (.din(din), .clk(clk), .dout(dout));
SR4_B UUT2 (.din(din), .clk(clk), .dout(dout2));
initial
begin
    clk=1'b0;
    din=1'b1;
end
always//run continuously to generate clokck signal
begin
    #5;
    clk=~clk;
end
endmodule
```
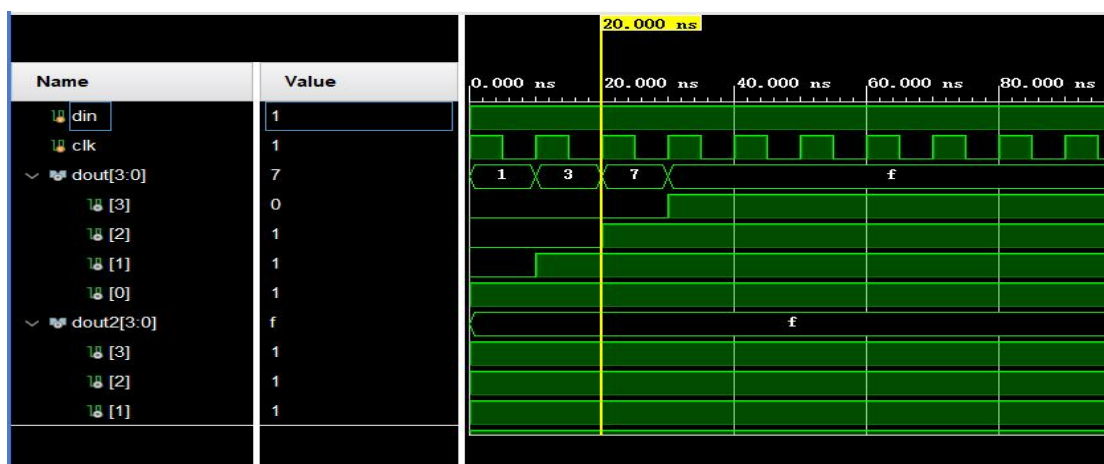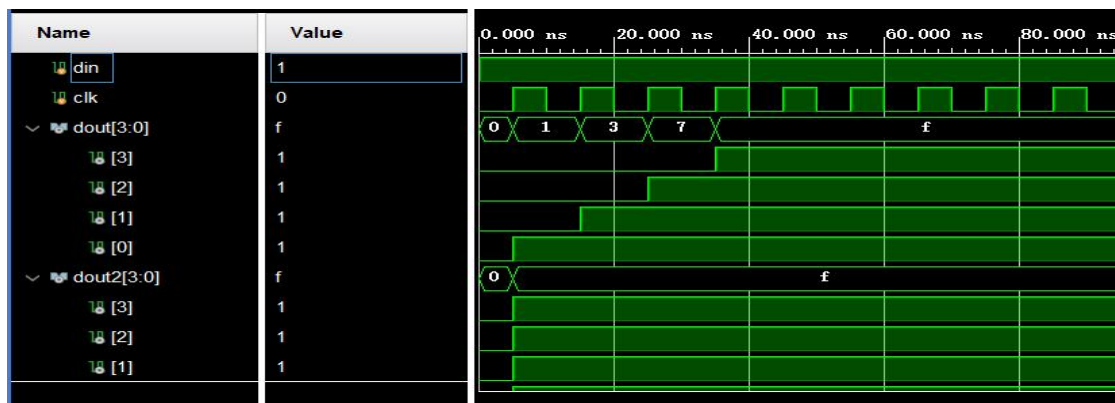

<u>Verilog Conditional Statements</u>

• Procedural conditional (**?**) operator
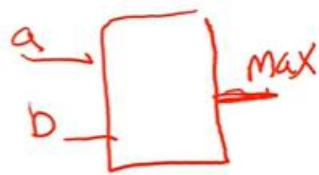
    -Compact single-way conditional expression

        test ? true_expression : false_expression

        e.g.

        wire [3:0] a, b, max;

        <u>**assign max=(a>b) ? a:b;**</u>

• Procedural **if – else – else if** statements

    -if part of statement only taken for conditions equal   ι equals 0, z,x

    -Use of begin-end bracketing is optional (if only 1 statement)

    -The else if statement is really a nested if statement within the original else statement

    -In coding combinational circuits, it is recommended that the last else of a group of nested if-else statements appears without an
if

//2-bit mux example

```verilog
    module mux2 (S, din, dout);
    input S;
    input [1:0] din;
    output dout;
    reg dout;          // or output reg dout;
always @ (S, din) //S and din are inputs
begin
    if (S==1'b0) //use == to justify equal or not
        begin
            dout=din[0];
        end
        else
        begin
            dout=din[1]; //use blocking assignments for combinational logic
        end
end
endmodule
```
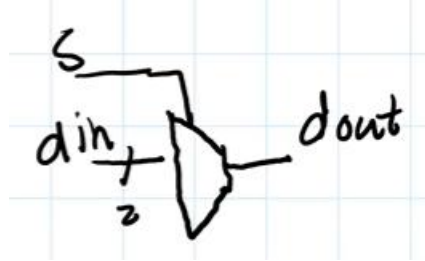

- Procedural case statement
  - Convenient multi-way selection statement
  - Compares an expression to a series of cases and executes the statement or statement group associated with the first matching case

```verilog
    case (expression)
        case1: statement;
        case2: statement;
        ...
        default: statement;
    endcase
```

//2-bit mux example

```verilog
    module mux2 (S, din, dout);
    input S;
    input [1:0] din;
    output reg dout;
    always @ (S, din)
    begin
            case(S)
                1'b0: dout=din [0];
                1'b1: dout=din [1];
                default: dout=0;
            endcase
    end
```
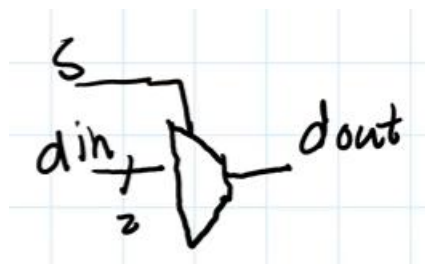
endmodule

Verilog Iterative Statements

Verilog Loops

    -use to replicate hardware //make copies of the hardware

    -control variable //often integer numbers

        1.initialed

        2.test

            -Verilog requires all loops to have a <u>fixed</u> number of iterations

        3.updated

            -Verilog does not have auto increment/decrement operators like C（++/--）

- Procedural **for** statement

    -Used for indexed looping

        for (init_assignment; constant expression; step)
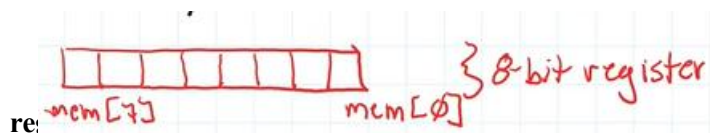
        begin

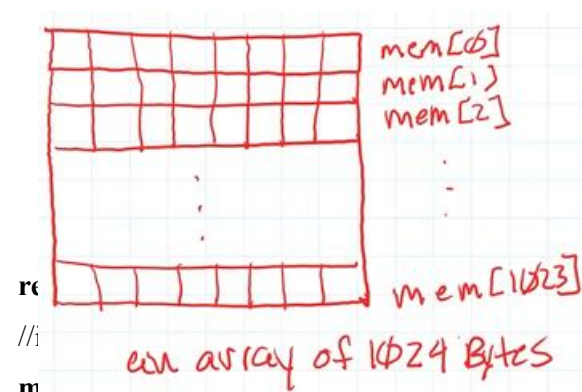   **··· // statements to repeat**

        end

        // **constant expression** is test and **step** is updated

    -Can be nested or combined with other procedural statements

 -use for loop to initialize a memory element

re

mem= 8'b0; //initialize reg to 0

re

//

m

mem[1] = 8'b0;

mem[2] = 8'b0;

···

mem[1023] = 8'b0;

integer cnt;

    initial

    begin

      for (cnt=0; cnt<1024; cnt=cnt+1)

      begin

        mem[cnt]=8'b0;

      end

- Procedural **while** statement

 -Executes as long as a conditional expression evaluates as true

    while (conditional_expression)

    begin

    ⋯ **// statements to repeat**

    end

e.g.

for (cnt=0; cnt<1024; cnt=cnt+1)

      begin

        mem[cnt]=8'b0;

      end

can be turned into

cnt=0; //initialize control variable
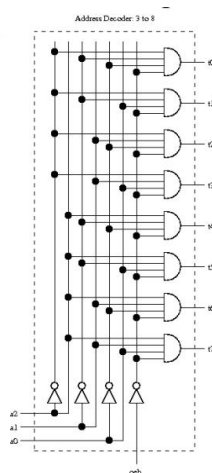
while(cnt<1024) //test control variable

begin

    mem[cnt]=8'b0;

    cnt=cnt+1; //update control variable

end

Verilog Behavioral Modeling Applications

– 3-to-8 address decoder

a0, a1, a2: binary address to memory location, allows to select which line of the memory to access

oeb: output to function as a decoding of the input when oeb=0 to   enable

//3-to-8 address decoder in Verilog

1) implement with case statement

```verilog
module decoder_38( in_data, oeb, out_data);
input wire [2:0] in_data;
input oeb;
output reg [7:0] out_data;
always @(in_data,oeb)
begi
    if(oeb==1'b0)
    begin
        case(in_data)
            3'b000: out_data=8'b0000_0001;
            3'b001: out_data=8'b0000_0010;
            3'b010: out_data=8'b0000_0100;
            3'b011: out_data=8'b0000_1000;
            3'b100: out_data=8'b0001_0000;
            3'b101: out_data=8'b0010_0000;
            3'b110: out_data=8'b0100_0000;
            3'b111: out_data=8'b1000_0000;
            default:out_data=8'b0000_0000;
        endcase
    end
    else
    begin
        out_data=8'b0000_0000;
    end
end
endmodule
```

2) implement with shift operation

```verilog
    000→8'b0000_0001
    010→8'b0000_0100
    8'b0000_0001<<010→8'b0000_0100
    module decoder_38( in_data, oeb, out_data);
    input wire [2:0] in_data;
    input oeb;
    output [7:0] out_data;
    //use continuous assignment
    assign out_data=(oeb==1'b0) ? (8'b0000_0001<<in_data):(8'b0000_0000);
    endmodule
```

3) implement with a for loop

-set **out_data [in_data] = 1'b1;**

//set each bit of out_data based upon in_data

```verilog
module decoder_38( in_data, oeb, out_data);
input wire [2:0] in_data;
input oeb;
output [7:0] out_data;
reg [7:0] out_data;
integer cnt;
always @ (in_data, oeb)
begin
    if (oeb ==1'b0)//output is enabled
    begin
        for(cnt=0; cnt<8; cnt=cnt+1) //set the value for each bit of the output
        begin
            if(cnt==in_data) //check if current bit location equals to the input value
            begin
                out_data[cnt]=1'b1;
            end
            else
            begin
                out_data[cnt]=1'b0;
            end
        end
    end
end
endmodule
```

Combinational logic circuit examples

– 4-bit comparator

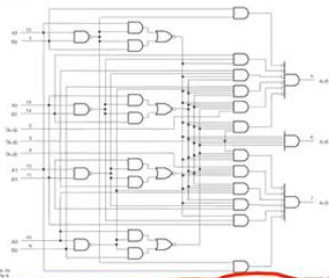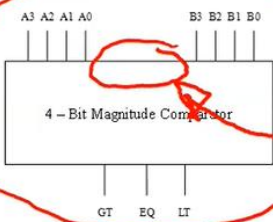| COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A3, B3	A2, B2	A1, B1	A0, B0	A > B	A < B	A = B	A > B	A < B	A = B
A3 > B3	X	X	X	X	X	X	H	L	L
A3 < B3	X	X	X	X	X	X	L	H	L
A3 = B3	A2 > B2	X	X	X	X	X	H	L	L
A3 = B3	A2 < B2	X	X	X	X	X	L	H	L
A3 = B2	A2 = B2	A1 > B1	X	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	X	X	X	L	H	L
A2 = B3	A2 = B2	A1 = B1	A0 > B0	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 < B0	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	L	L	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	H	L	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	X	X	H	L	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	L	L	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	L	H	H	L

//logic block diagram, the actual circuit, truth table

– 4-bit comparator

Order of operation

COMPARING INPUTS				CASCADING INPUTS			OUTPUTS		
A3, B3	A2, B2	A1, B1	A0, B0	A > B	A < B	A = B	A > B	A < B	A = B
A3 > B3	X	X	X	X	X	X	H	L	L
A3 < B3	X	X	X	X	X	X	L	H	L
A3 = B3	A2 > B2	X	X	X	X	X	H	L	L
A3 = B3	A2 < B2	X	X	X	X	X	L	H	L
A3 = B2	A2 = B2	A1 > B1	X	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 < B1	X	X	X	X	L	H	L
A2 = B3	A2 = B2	A1 = B1	A0 > B0	X	X	X	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 < B0	X	X	X	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	L	L	H	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	H	L	L	H	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	X	X	H	L	L	H
A3 = B3	A2 = B2	A1 = B1	A0 = B0	H	H	L	L	L	L
A3 = B3	A2 = B2	A1 = B1	A0 = B0	L	L	L	H	H	L

```verilog
module comp4 (A, B, iAltB, iAeqB, iAgtB, oAltB, oAeqB, oAgtB);
input [3:0] A,B;
input iAltB, iAeqB, iAgtB;
output reg oAltB, oAeqB, oAgtB;
always @ (A, B, iAltB, iAeqB, iAgtB)
begin
    //reset all output regs
    oAltB=1'b0;
    oAeqB=1'b0;
```

```verilog
        oAgtB=1'b0;
    if(A>B)
    begin
        oAgtB=1'b1;
    end
    else if (A<B)
    begin
        oAltB=1'b1;
    end
    else if (A==B)
    begin
        //check cascading input
        case({iAgtB, iAltB, iAeqB})
            3'b100: oAgtB=1'b1;
            3'b010: oAltB=1'b1;
            3'b001: oAeqB=1'b1;
            3'b000: begin
                        oAgtB=1'b1;
                        oAltB=1'b1;
                    end
            default:
                    begin
                        oAgtB=1'b0;
                        oAltB=1'b0;
                        oAeqB=1'b0;
                    end
        endcase
    end
end
endmodule
```

-Vivado Synthesis Waring

    -inferring latch for variables xxx register

    //latch: maintain values

        -caused when a registered is updated within a conditional expression inside a procedural block, but the register is not updated in all branches of the conditional

• Coding pitfalls

    -Assigning a variable within multiple procedure blocks

        //Race condition → non-deterministic operations

    -Forgetting to include all inputs in sensitivity list

        //Use of implicit sensitivity lists **always @ (*)**

-Warning: Failure to assign all controlled outputs on every

//iteration of a procedural block (inferred latch)

-Error: Failure to use a constant iteration count results

//in an unsynthesizable code (for / while)

Structural Modeling

- Digital system (logic circuit) described using instantiation of one or more design modules

  -Built-in primitives

  -Dataflow models

  -Behavioral models

- Focuses on interconnections between lower-level design modules
- Requires logic synthesis to create gate-level logic circuit
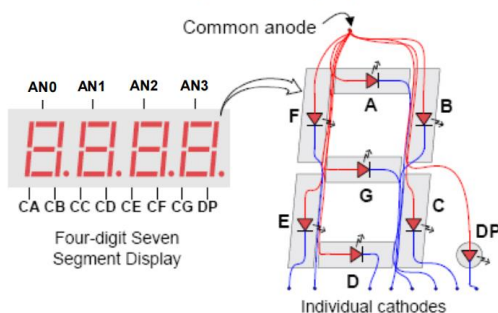
top module:

①instantiation

②interconnections

Module Instantiation

- A module may be composed of instances of other modules
- The full hierarchical path name of each module specifies its location in the hierarchy
- Only top level module is instantiated by its module definition

- Decoder Example
  - Develop a decoder to translate a 4-bit hexadecimal number to a seven segment display

Common anode

AN0 AN1 AN2 AN3

CA CB CC CD CE CF CG DP

Four-digit Seven
Segment Display

Individual cathodes

x	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
A	0	0	0	1	0	0	0
B	1	1	0	0	0	0	0
C	0	1	1	0	0	0	1
D	1	0	0	0	0	1	0
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	0

// Structural Modeling Example
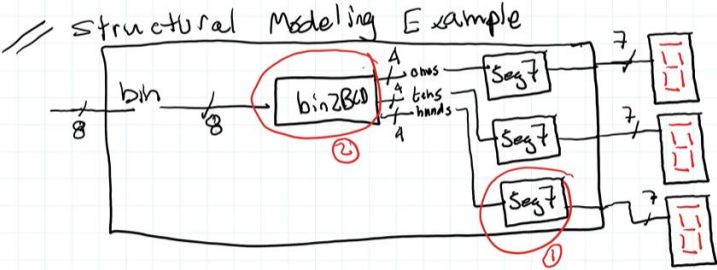
-Seg 7 module

BCD value LSB ... MSB

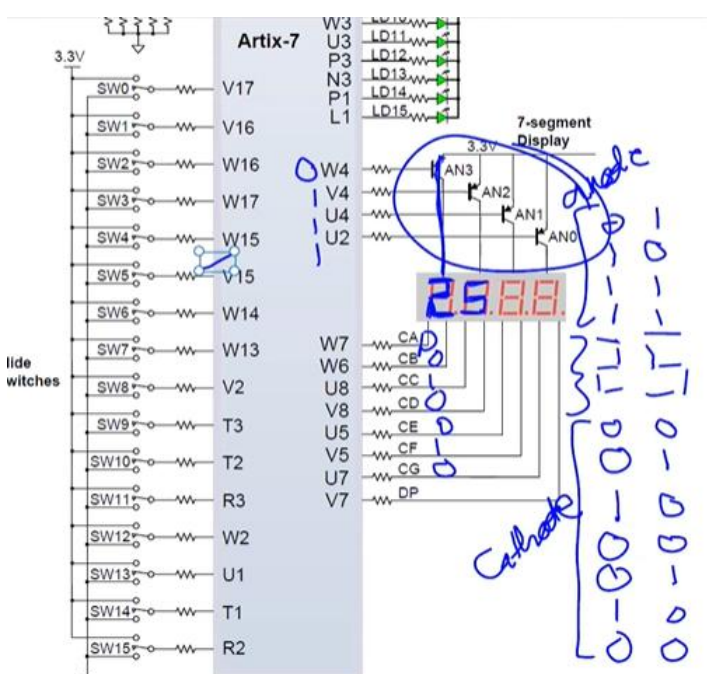| X	a	b	c	d	e	f	g
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
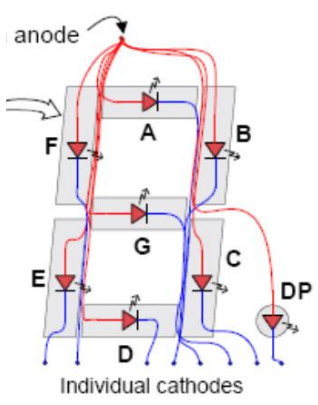
0=on, 1=off

-Bin2BCD module

-shift and add 3 methods

①shift binary value left 1 bit into BCD bits

②if any BCD digital is >=5, then add 3 to the digit

:repeat 8 times to get 8-bit input into BCD outputs

anode: 阴极，

  0=number on,

  1=number off

cathode: 阳极；

  0=segment on,

30

1=segment off

display value
is multiplexed
(same cathode value)

pos

2

bin

8

8

bin2BCD

2

A ones
A tens
hunds
A

Seg7

7 7

out 7

Seg7

7

Seg7

1

11111

anode value is
multiplexed to
select which
display is on for
current cathode
value

1110
1101
1011
0111

an

4

last display
is always off
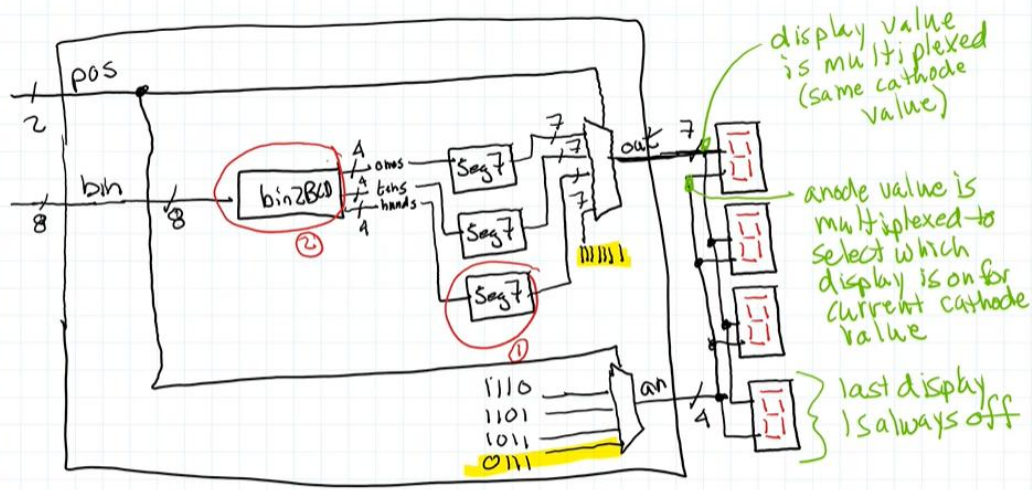
```verilog
module top(pos, bin, out, an);
input [1:0] pos;
input [7:0] bin;
output reg [6:0] out;
output reg [3:0] an;
wire [3:0] ones, tens, hunds;
wire [6:0] seg7ones, seg7tens, seg7hunds;
bin2BCD U1 (.bin(bin),   .ones(ones),   .tens(tens),   .hunds(hunds));
Seg7 U2 (.BCD(ones),.out(seg7ones));
Seg7 U3 (.BCD(tens),.out(seg7tens));
Seg7 U4 (.BCD(hunds),.out(seg7hunds));
always @ (pos,seg7ones, seg7tens, seg7hunds)
begin
    //implement mux for 7 segment display
    case(pos)
        2'b00://ones
            begin
                out=seg7ones;
                an=4'b1110;
            end
        2'b01://tens
            begin
                out=seg7tens;
                an=4'b1101;
            end
        2'b10://hunds
            begin
                out=seg7hunds;
                an=4'b1011;
```

```verilog
                    end
            2'b11://thousands
                begin
                    out=7'b1111111;
                    an=4'b0111;
                end
        endcase
end
endmodule


module bin2BCD(bin, ones, tens, hunds);
input [7:0] bin;
output [3:0] ones, tens, hunds;
reg [19:0] temp; //temporary register to hold 8-bit binary input and BCD output during shifting and add 3 process
integer cnt; //control variable for the for loop
always @ (bin)
begin
    temp={12'b0, bin}; //concatenate 12 bits of 0 with binary input and store to temp register
    //perform the shift and add 3 algorithm 8 times
    for (cnt=0; cnt<8; cnt=cnt+1)
    begin
        temp = temp << 1;//shift temp register to left by 1
        if(cnt<7)//first shifts
        begin
            //test each BCD output >=5
            if(temp[11:8]>=5)//if ones digit >=5
            begin
                temp[11:8]=temp[11:8]+3;
            end
            if(temp[15:12]>=5)//if tens digit >=5
            begin
                temp[15:12]=temp[15:12]+3;
            end
            if(temp[19:16]>=5)//if hunds digit >=5
            begin
                temp[19:16]=temp[19:16]+3;
            end
        end
    end
end
```

```verilog
assign ones=temp[11:8];
assign tens=temp[15:12];
assign hunds=temp[19:16];
endmodule


module Seg7 (BCD, out);
input [3:0] BCD;
output reg [6:0] out;
always @ (BCD)
begin
    case(BCD) //care about the lsb and msb
        4'b0000:out=7'b1000000;
        4'b0001:out=7'b1111001;
        4'b0010:out=7'b0100100;
        4'b0011:out=7'b0110000;
        4'b0100:out=7'b0011001;
        4'b0101:out=7'b0010010;
        4'b0110:out=7'b0000010;
        4'b0111:out=7'b1111000;
        4'b1000:out=7'b0000000;
        4'b1001:out=7'b0010000;
        default: out=7'b1111111; //turn off display if invalid number
        endcase
end
endmodule
```
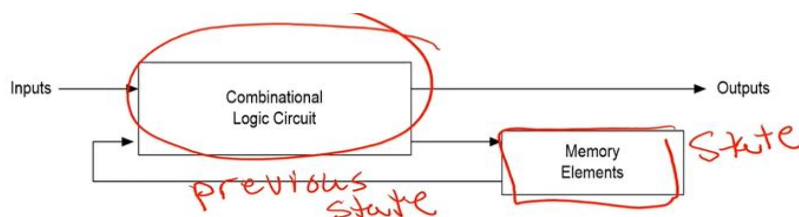
Sequential Circuit Applications

Sequential Circuits

- Contain both combinational logic and memory elements
    - Flip-flops
    - Shift registers
    - Counters

Flip Flops

- Asynchronous Flip-flop Inputs
    - -Inputs that affect the state of the flip-flop independently of the clock signal
    - -Preset (PRE)
        - Places the flip-flop into the SET state
    - -Clear (CLR)
        - Places the flip-clop into the RESET state
    - -Must be included within **always** block sensitivity list

Sequential Circuits Examples

D flip-flop with asynchronous PRE and CLR

PRE: set state =1. CLR: set state =0

```verilog
module DFF (clk, D, clr, pre, Q, Qnot);// clr, pre are active high
input clk, D, clr, pre;
output reg Q;
output Qnot;
always @ (posedge clk, posedge clr, posedge pre)
begin
    //first look at asynchronous inputs
    if(clr==1'b1)
    begin
        Q<=1'b0;
    end
    else if (pre==1'b1)
    begin
        Q<=1'b1;
    end
    else
    begin
        Q<=D;
    end
end
assign Qnot=~Q;
endmodule
```

Shift Registers

- Type
    - -Free-running
    - -Universal

- Behavior (on active clock edge)
    - If asynchronous clear is true,
        Register is loaded with zeros
    - If asynchronous load is true and clear is false, //parallel load
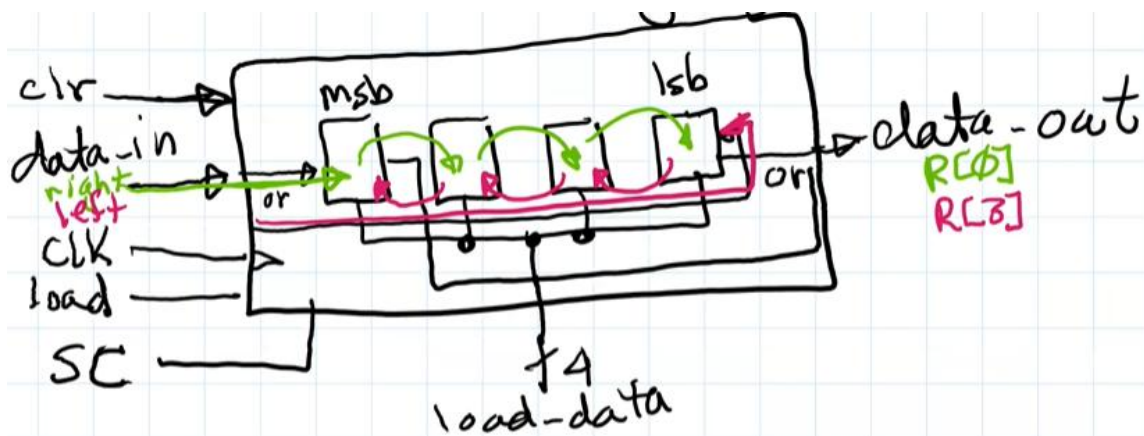   Register is loaded from the data inputs
    - If asynchronous load and clear are both false,
        Register contents are shifted according to value of Shift control
        //Shift control=1→shift right; Shift control=0→shift left
    - Else ignore asynchronous input values

Universal 4-bit shift register


```
module shiftreg (data_in, clk, clr, load, sc, load_data, data_out);
input data_in, clk, clr, load, sc;
input [3:0] load_data;
output data_out;
reg [3:0] R;
always @ (posedge clk, posedge clr, posedge load) //clr and load are asynchronous inputs
begin
    if (clr==1'b1)
    begin
        R<=0;
    end
    else if(load==1'b1)
    begin
        R<=load_data;
    end
    else
    begin
        if(sc==1'b1)//shift right
        begin
            R<={data_in, R[3:1]};
        end
```

```verilog
        else//shift left
        begin
            R<={R[2:0],data_in};
        end
    end
end
assign data_out=(sc==1'b1)?R[0]:R[3];
endmodule
```

-reuse modules with different sized entities
  -for 8-bit→replace 3 with 7 and 2 with 6
  -for 8-bit→replace 3 with 15 and 2 with 14
  //each change to the module, need to resynthesize code and implement into a new bit file

```verilog
module shiftreg (data_in, clk, clr, load, sc, load_data, data_out);
input data_in, clk, clr, load, sc;
input [3:0] load_data;
output data_out;
reg [3:0] R;
always @ (posedge clk, posedge clr, posedge load) //clr and load are asynchronous inputs
begin
    if (clr==1'b1)
    begin
        R<=0;
    end
    else if(load==1'b1)
    begin
        R<=load_data;
    end
    else
    begin
        if(sc==1'b1)//shift right
        begin
            R<={data_in, R[3:1]};
        end
        else//shift left
        begin
            R<={R[2:0],data_in};
        end
    end
end
```

```verilog
assign data_out=(sc==1'b1)?R[0]:R[3];
endmodule
```

   -single module that can be implement with the designed register width without being resynthesized

    -use parameters to set shift register width

     -create an n-bit register using a parameter named width

```verilog
module shiftreg (data_in, clk, clr, load, sc, load_data, data_out);
parameter width=4;//by default shift register is 4-bits wide
input data_in, clk, clr, load, sc;
input [(width-1):0] load_data;
output data_out;
reg [(width-1):0] R;
always @ (posedge clk, posedge clr, posedge load) //clr and load are asynchronous inputs
begin
    if (clr==1'b1)
    begin
        R<=0;
    end
    else if(load==1'b1)
    begin
        R<=load_data;
    end
    else
    begin
        if(sc==1'b1)//shift right
        begin
            R<={data_in, R[(width-1):1]};
        end
        else//shift left
        begin
            R<={R[(width-2):0],data_in};
        end
    end
end
assign data_out=(sc==1'b1)?R[0]:R[(width-1)];
endmodule
```

-creating an instance of a parameterized module//module top with 4-bit shift register

```verilog
module top(…);
…
wire din, clk, clr, ld, sc, dout;
```

```verilog
wire [3:0] data;
shiftreg U1 (.data_in(data_in), .clk(clk), .clr(clr), .load(ld), .sc(sc), ,load_data(data), .data_out(dout));
…
endmodule
```

-creating an instance of a parameterized module//module top with 32-bit shift register

```verilog
module top(…);
…
wire din, clk, clr, ld, sc, dout;
wire [31:0] data;
shiftreg #(.width(32)) U1 (.data_in(data_in), .clk(clk), .clr(clr), .load(ld), .sc(sc), ,load_data(data), .data_out(dout));
…
endmodule
```
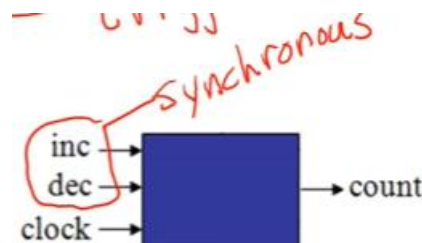
Counters

·   Counter types

   -Up/Down

   -Modulo N counter

   ·   BCD (Binary Coded Decimal)

   -Ring counter

Binary Up/Down Counters

• Behavior (on active clock edge//positive edge triggered)

   -If inc is true, //increment 增量

       • count = count +1 on clock edge

   -If dec is true,//decrement 减量

       • count = count –1 on clock edge

   -If both inc and dec are true,

       • count = count

   -If both inc and dec are false,

       • count = 0

-make a parasitized module that is 8-bit wide by default

Binary Up/Down Counter

```verilog
module UpDownCnt (clk, dec, inc, count);
parameter width=8;
input clk, inc, dec;
output reg [width-1:0] count;
always @ (posedge clk)
begin
    case({inc, dec})
```

```
            2'b00: count<=0;
            2'b01: count<=count-1;
            2'b10: count<=count+1;
            2'b11: count<=count; //is it necessary to use <= instead of =?
        endcase
    end
endmodule
```

Modulo N Counter

- Behavior (on active clock edge)
  - If count < N-1,
    - count = count + 1
  - If count == N-1
    - count = 0

```
//Module-N counter
module ModNCounter (clk, count, N);
parameter width;
input clk;
input [widh-1:0] N; //actually width of N is log2(N)
output reg [widh-1:0] count;
always @ (posedge clk)
begin
    if(count<N-1)
    begin
        count<=count+1;
    end
    else if (count==N-1)
    begin
        count<=0;//recycle count
    end
end
endmodule
```
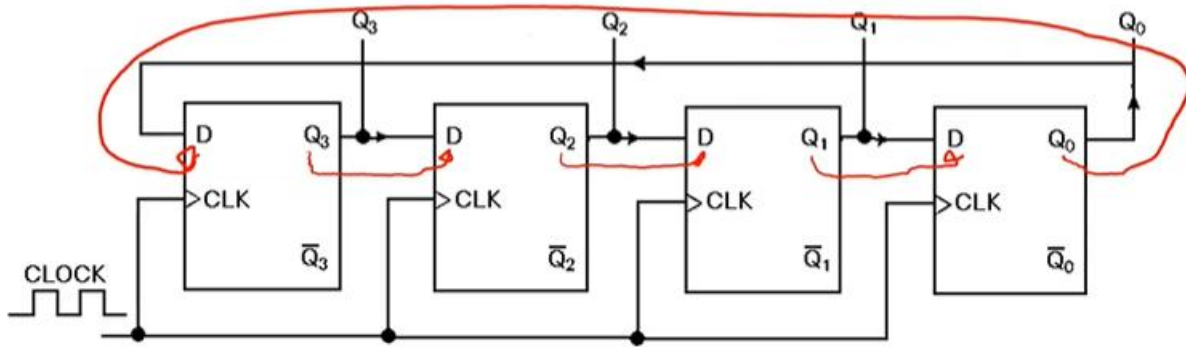
Ring Counter

- Behavior
  - Q[n-1:0] preset with pattern (ex: 0001)
  - On active edge of clock
    - Q[n-2:0] = Q[n-1:1]
    - Q[n-1] = Q[0]

```verilog
module ring(clk, pre, Q);
input clk, pre;
output reg [3:0] Q;
always @ (posedge clk, posedge pre)
begin
    if(pre==1'b1)
    begin
        Q<=4'b1110;
    end
    else
    begin
        Q[3]<=Q[0];
        Q[2:0]<=Q[3:1];//these two lines can be Q<={Q[0]:Q[3:1]};
    end
end
endmodule
```


asynchronous preset to initialize the ring counter
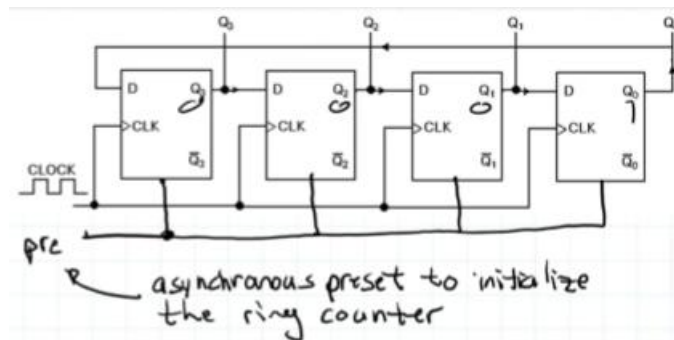
The Basys 3 board includes a single 100 MHz oscillator connected to pin W5 (W5 is a MRCC input on bank 34).
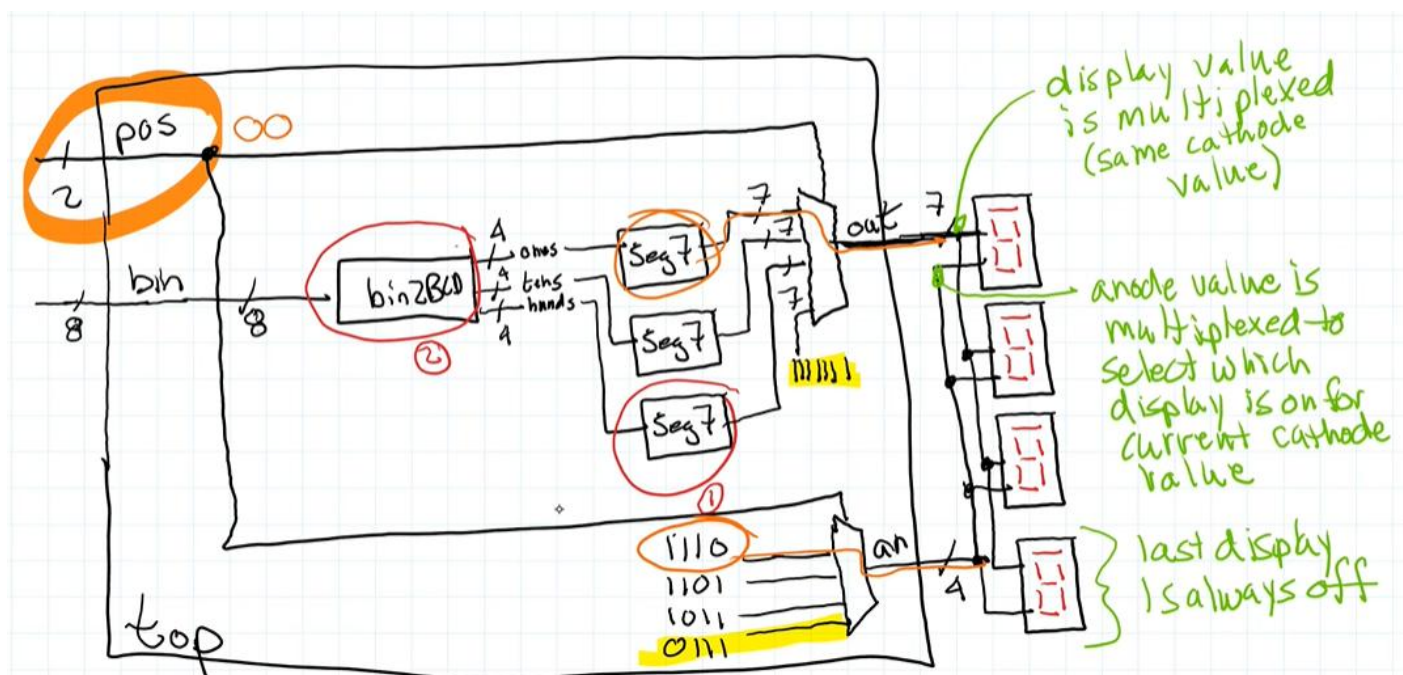
This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect.
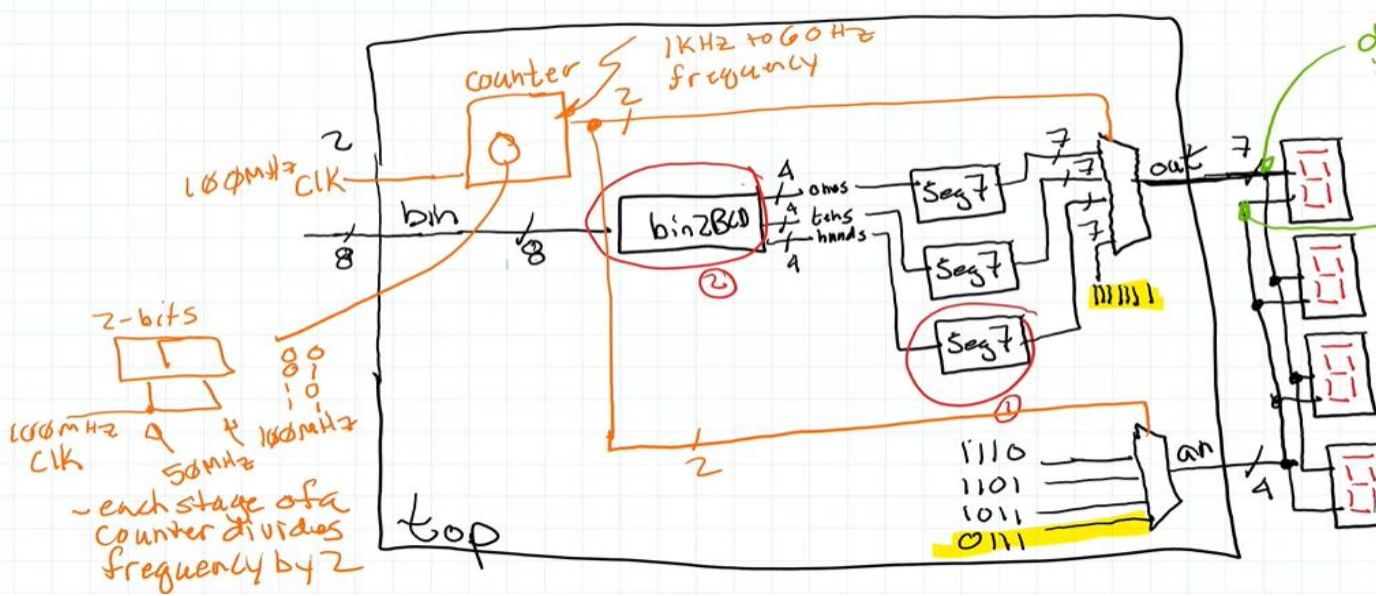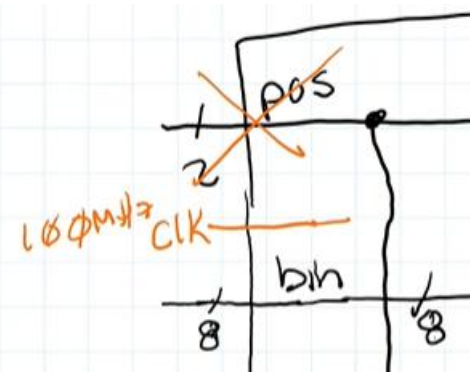
For each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of about 1 KHz to 60Hz.

CODE:

module Bin8ToBCD(clk, bin, out, an);

input clk;

input [7:0] bin;

output reg [6:0] out;

output reg [3:0] an;

reg [19:0] count; //20-bit counter to divide 100 MHz clock frequency for selecting 7-segment display

wire [1:0] pos;

wire [3:0] ones, tens, hunds;

wire [6:0] seg7ones, seg7tens, seg7hunds;

bin2BCD U1 (.bin(bin), .ones(ones), .tens(tens), .hunds(hunds));

Seg7 U2 (.BCD(ones),.out(seg7ones));

Seg7 U3 (.BCD(tens),.out(seg7tens));

Seg7 U4 (.BCD(hunds),.out(seg7hunds));

always @(posedge clk) //create a 20-bit free running binary counter

begin

```verilog
        count<=count+1;
end
assign pos=count[19:18]; //use upper 2-bit of counter for mux select input, too fast would not show clear and too slow would
blink
always @ (pos,seg7ones, seg7tens, seg7hunds)
begin
    //implement mux for 7 segment display
    case(pos)
        2'b00://ones
            begin
                out=seg7ones;
                an=4'b1110;
            end
        2'b01://tens
            begin
                out=seg7tens;
                an=4'b1101;
            end
        2'b10://hunds
            begin
                out=seg7hunds;
                an=4'b1011;
            end
        2'b11://thousands
            begin
                out=7'b1111111;
                an=4'b0111;
            end
    endcase
end
endmodule
module bin2BCD(bin, ones, tens, hunds);
input [7:0] bin;
output [3:0] ones, tens, hunds;
reg [19:0] temp; //temporary register to hold 8-bit binary input and BCD output during shifting and add 3 process
integer cnt; //control variable for the for loop
always @ (bin)
begin
    temp={12'b0, bin}; //concatenate 12 bits of 0 with binary input and store to temp register
    //perform the shift and add 3 algorithm 8 times
```

```verilog
        for (cnt=0; cnt<8; cnt=cnt+1)
        begin
            temp = temp << 1;//shift temp register to left by 1
            if(cnt<7)//first shifts
            begin
                //test each BCD output >=5
                if(temp[11:8]>=5)//if ones digit >=5
                begin
                    temp[11:8]=temp[11:8]+3;
                end
                if(temp[15:12]>=5)//if tens digit >=5
                begin
                    temp[15:12]=temp[15:12]+3;
                end
                if(temp[19:16]>=5)//if hunds digit >=5
                begin
                    temp[19:16]=temp[19:16]+3;
                end
            end
        end
end
assign ones=temp[11:8];
assign tens=temp[15:12];
assign hunds=temp[19:16];
endmodule
module Seg7 (BCD, out);
input [3:0] BCD;
output reg [6:0] out;
always @ (BCD)
begin
    case(BCD) //care about the lsb and msb
        4'b0000:out=7'b1000000;
        4'b0001:out=7'b1111001;
        4'b0010:out=7'b0100100;
        4'b0011:out=7'b0110000;
        4'b0100:out=7'b0011001;
        4'b0101:out=7'b0010010;
        4'b0110:out=7'b0000010;
        4'b0111:out=7'b1111000;
        4'b1000:out=7'b0000000;
```

4'b1001:out=7'b0010000;

default: out=7'b1111111; //turn off display if invalid number

endcase

end

endmodule

Linear Feedback Shift Register

• Behavior:

$$input = q0 \oplus q2$$
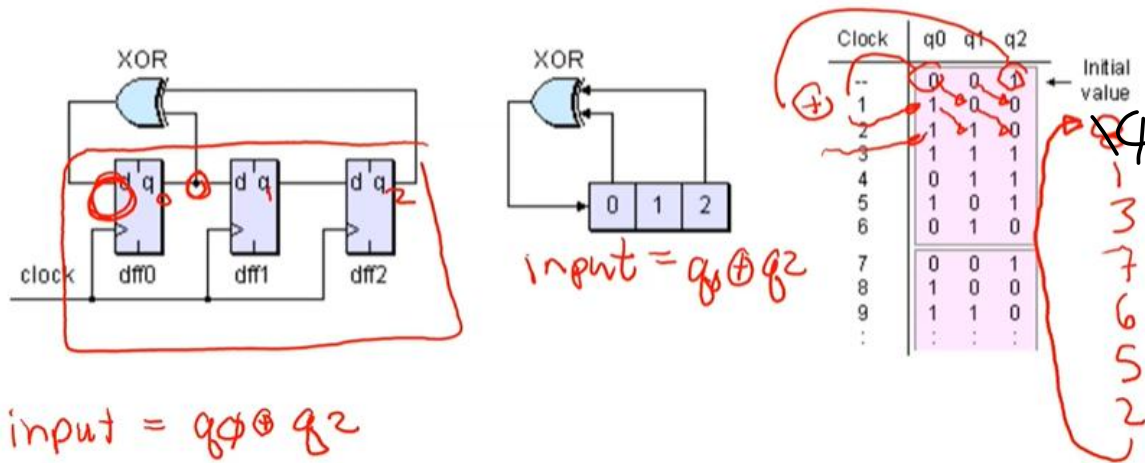
-linear function typically using XOR or XNOR

-deterministic operations

  -sequence of values is determined by its current state   ·

  -there are a finite number of possible states, the pattern is repeating

      -(2^n)-1 is states for an n-bit LFSR

-why use?

    -counter

        -non-binary count sequence

        -very efficient … think fast

44

-pseudo random number generator //pseudo:伪装

-pseudo-noise sequence generator

- stream-cipher for cryptography//流密码加密

• Example:

– Create an n-bit LFSR

  • Where $3 \leq n \leq 16$

– Must create module with

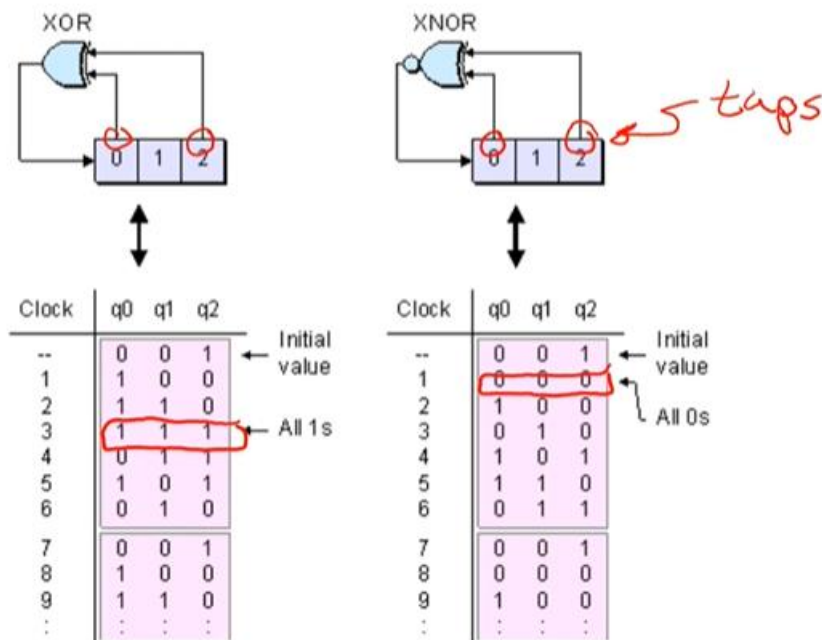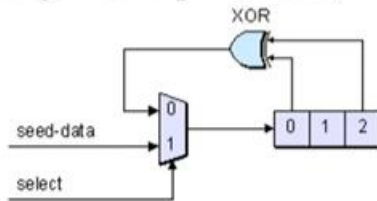 • Clock and an asynchronous enable input

  • Seed input (n-bits) with synchronous seed input

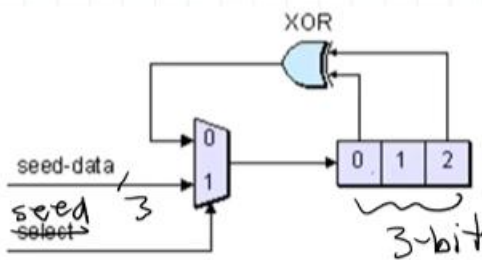 • Repeat flag output (pulse at 2^n-1)

  • Shift register output (n-bits)

# of Bits	Length of Loop	Taps
2	3 *	[0,1]
3	7 *	[0,2]
4	15	[0,3]
5	31 *	[1,4]
6	63	[0,5]
7	127 *	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1,023	[2,9]
11	2,047	[1,10]
12	4,095	[0,3,5,11]
13	8,191 *	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]
17	131,071	[2,16]
18	262,143	[6,17]
19	524,287 *	[0,1,4,18]
20	1,048,575	[2,19]
21	2,097,151	[1,20]
22	4,194,303	[0,21]
23	8,388,607	[4,22]
24	16,777,215	[0,2,3,23]
25	33,554,431	[2,24]
26	67,108,863	[0,1,5,25]
27	134,217,727	[0,1,4,26]
28	268,435,455	[2,27]
29	536,870,911	[1,28]
30	1,073,741,823	[0,3,5,29]
31	2,147,483,647 *	[2,30]
32	4,294,967,295	[1,5,6,31]

//LFSR example:
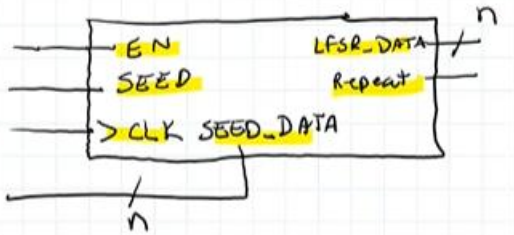
– SEED_DATA = initial value

# of Bits	Length of Loop	Taps
2	3 *	[0,1]
3	7 *	[0,2]
4	15	[0,3]
5	31 *	[1,4]
6	63	[0,5]
7	127 *	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1,023	[2,9]
11	2,047	[1,10]
12	4,095	[0,3,5,11]
13	8,191 *	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]

XOR

seed-data

seed select

value is output of linear function

5	31 *	[1,4]
6	63	[0,5]
7	127 *	[0,6]
8	255	[1,2,3,7]
9	511	[3,8]
10	1,023	[2,9]
11	2,047	[1,10]
12	4,095	[0,3,5,11]
13	8,191 *	[0,2,3,12]
14	16,383	[0,2,4,13]
15	32,767	[0,14]
16	65,535	[1,2,4,15]

linear function changes based on the number of bits in LFSR

tester

Verilog:

```
module LFSR(clk, en, seed, seed_data, LFSR_data, Repeat);
parameter n=4; //default width of LFSR is 4-bit
input clk, en, seed;
input [n-1:0] seed_data;
output Repeat;
output [n-1:0] LFSR_data;

reg [n-1:0] LFSR;
reg XOR; //output from linear function

always @ (posedge clk, posedge en)
begin
    if(en==1'b1)//LFSR is enabled
    begin
        if(seed==1'b1)//seeding the LFSR
        begin
            LFSR<=seed_data;//parallel load the LFSR register
        end
        else//not sedding, just shifting
        begin
            LFSR<={LFSR[n-1:0],XOR};//shift LFSR with linear function (XOR) output
```

```verilog
            end
        end
        else //LFSR is not enabled
        begin
            LFSR<=0;//clear LFSR when not enabled
        end
end

//create the linear function for feedback of XOR value
always @ *
begin
    case(n)
        3:XOR=LFSR[0]^LFSR[2];
        4:XOR=LFSR[0]^LFSR[3];
        5:XOR=LFSR[1]^LFSR[4];
        6:XOR=LFSR[0]^LFSR[5];
        7:XOR=LFSR[0]^LFSR[6];
        8:XOR=LFSR[1]^LFSR[2]^LFSR[3]^LFSR[7];
        9:XOR=LFSR[3]^LFSR[8];
        10:XOR=LFSR[2]^LFSR[9];
        11:XOR=LFSR[1]^LFSR[10];
        12:XOR=LFSR[0]^LFSR[3]^LFSR[5]^LFSR[11];
        13:XOR=LFSR[0]^LFSR[2]^LFSR[3]^LFSR[12];
        14:XOR=LFSR[0]^LFSR[2]^LFSR[4]^LFSR[13];
        15:XOR=LFSR[0]^LFSR[14];
        16:XOR=LFSR[1]^LFSR[2]^LFSR[4]^LFSR[15];
    endcase
end
assign LFSR_data=LFSR;
assign Repeat={LFSR==seed_data}?1'b1:1'b0;//set Repeat to 1 whenever LFSR register equals the initial seed data (repeats)
endmodule

testbench：
module Tester();
parameter N=3;
reg CLK, EN, SEED;
reg [N-1:0] SEED_DATA;
wire [N-1:0] LFSR_DATA;
wire REPEAT;
```

```verilog
//create n 8-bit LFSR instance
LFSR                                                                    #(.n(N))
UUT(.clk(CLK), .en(EN), .seed(SEED), .seed_data(SEED_DATA), .LFSR_data(LFSR_DATA), .Repeat(REPEAT));
initial
begin
    CLK=0;
    EN=0;
    SEED=0;
    SEED_DATA=3'b100;
    #10;
    EN=1;
    SEED=1;
    #20;
    SEED=0;
end

always
begin
    #5;
    CLK=~CLK;
end
endmodule
```

FSM Design with Verilog

Mealy & Moore Machines

• Mealy modeled machines

    -The next state and output is determined by current state of the machine and current input to the machine

        Next State = F(current state, input)

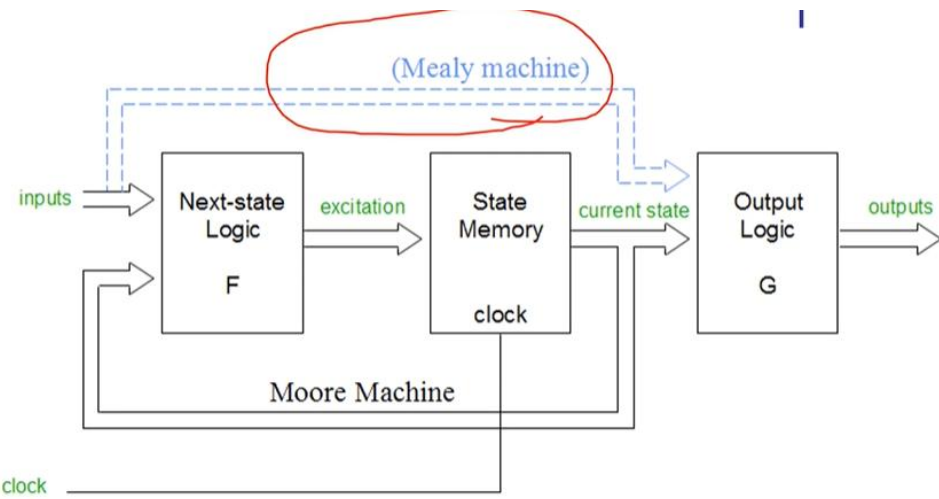   <u>Output = G(current state, input)</u>

• Moore modeled machines

    -The next output is only determined by the current state and not directly on the current input
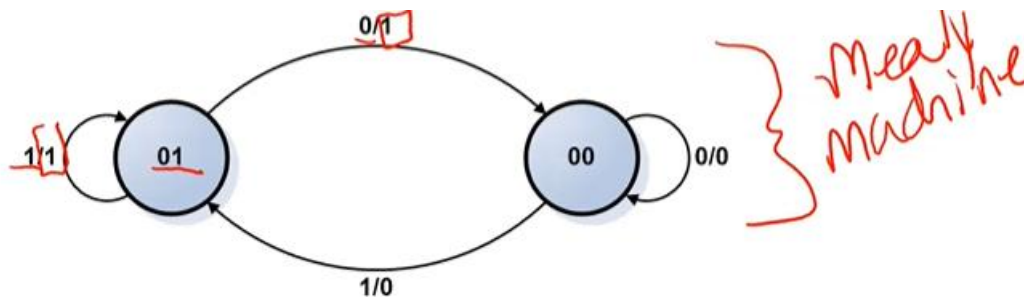
        Next State = F(current state, input)

      <u>Output = G(current state)</u>

State Diagrams

• Graphical representation of sequential circuit operation captured in state table

    -State represented by a circle with corresponding number of the state

    -Transitions between states indicated by directed lines (arcs) connecting the circles

• Numbered as "input x / output y" associated with transition

Verilog State Machines

• We need registers to hold

    -the current state

    -always @(posedge clk) block

• We need next state function

    -Where do we go from each state given the inputs

    -State by state case analysis

        • Next state determined by current state and inputs

• We need the output function

    -State by state analysis

        • Moore: output determined by current state only

        • Mealy: output determined by current state and inputs

• State register

    -Declare two values

        • state : current state – output of state register

        • nxtState : next state – input to state register

        • We rely on next state function to give us nxtState

-Declare symbols for states with state assignment

```
localparam IDLE=0, WAITFORB=1,    ⎫ constant values
          DONE=2, ERROR=3;        ⎬ for each state
```

```
reg [1:0] state,    // Current state
          nxtState; // Next state
```
2-bit registers

- Next State Function

    -Combinational logic function

        • Inputs : state, inputs

        • Output : nxtState

    -We could use assign statements

    -We will use an always @(*) block instead //* means continue

   • Allows us to use procedural block statements like "if" and "case"

    -Describes what happens in each state (typically use case statement)

- Next State Function (cont.)

—Inputs A,B

```
always @(*) begin
  nxtState = state;  // Default next state: don't move
  case (state)
    IDLE : begin
      if (B) nxtState = ERROR;
      else if (A) nxtState = WAITFORB;
    end                    else // A=0, B=0
    WAITFORB : begin            nxtState = IDLE;
      if (B) nxtState = DONE;
    end
    DONE : begin
    end
    ERROR : begin
    end
  endcase
end
```
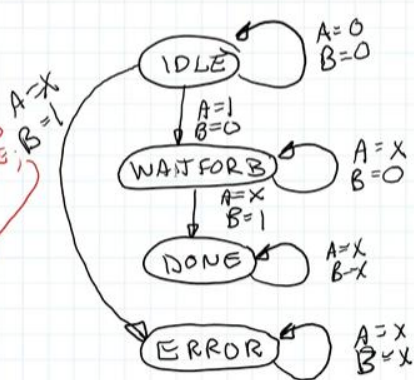

- Output Function

    -Describe the output of each state

    -Typically combined with next state function

```
always @(*) begin
  nxtState = state;  // Default next state: stay where we are
  out = 0;           // Default output
  case (state)
    IDLE : begin
      if (B) nxtState = ERROR;
      else if (A) nxtState = WAITFORB;
    end
    WAITFORB : begin
      if (B) nxtState = DONE;
    end
    DONE : begin
      out = 1;
    end
    ERROR : begin
    end
  endcase
end
```

Moore machine, output only dependent on current state
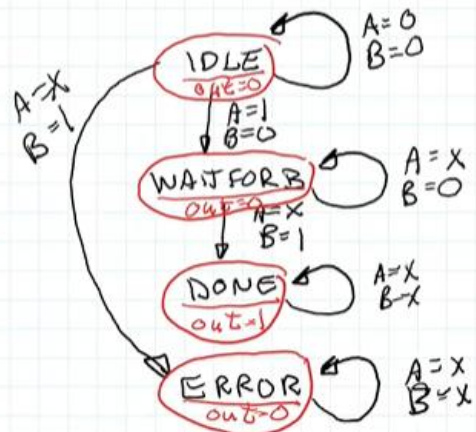
Design of FSM Circuits

    -edge detected

- Example:

    -Moore edge detection state machine

        • Read continuous logic level input signal, and determine when rising and falling edges occur
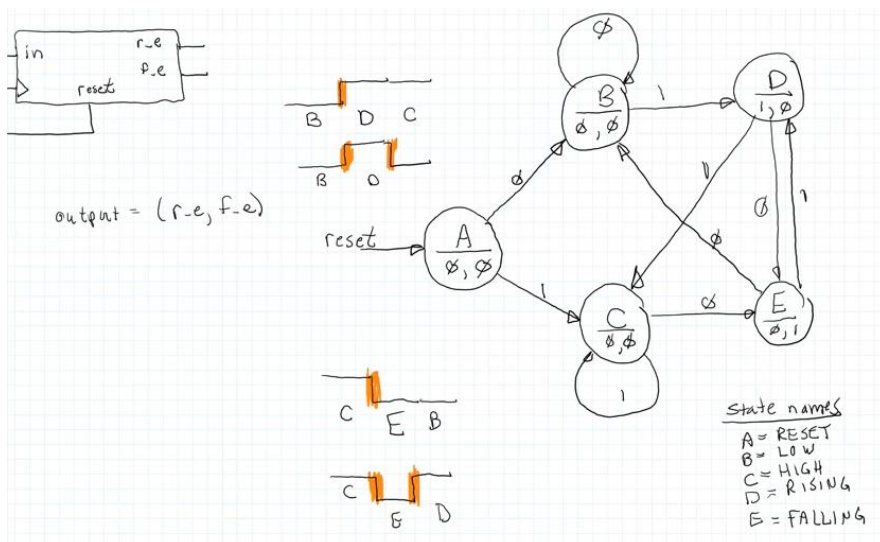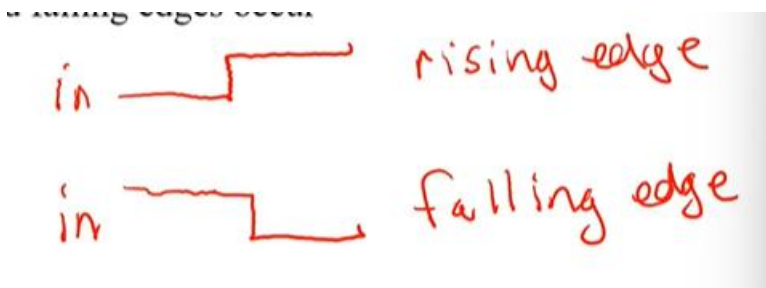
    -state registers

        -state

        -next state

        -5 states→3-bits

```
module MooreEdgeDetect(clk, in, reset, f_e, r_e);
localparam RESET=0, LOW=1, HIGH=2, RISING=3, FALLING=4; //or change it to parameter and upon the module
input clk, in, reset;
output reg f_e, r_e;


reg [2:0] state, nextstate;
always @ (posedge clk)
begin
    if(reset==1'b1)
    begin
        state<=RESET;
    end
    else
    begin
```

```verilog
            state<=nextstate;
        end
    end
end

always @ (*)//next state and output functions
begin
    nextstate=state;//default set next state to current state
    r_e=1'b0;
    f_e=1'b0;

    case(state)
        RESET:begin
                if (in==1'b0)
                begin
                    nextstate<=LOW;
                end
                else
                begin
                    nextstate<=HIGH;
                end
             end
        LOW:begin
                if (in==1'b1)
                begin
                    nextstate<=RISING;
                end
            end
        HIGH:begin
                if (in==1'b0)
                begin
                    nextstate<=FALLING;
                end
            end
        RISING:begin
                r_e=1'b1; //set the rising edge flag to 1
                if (in==1'b0)
                begin
                    nextstate<=FALLING;
                end
                else
```

```verilog
            begin
                nextstate<=HIGH;
            end
        end
    FALLING:begin
            f_e=1'b1; //set the falling edge flag to 1
            if (in==1'b0)
            begin
                nextstate<=LOW;
            end
            else
            begin
                nextstate<=RISING;
            end
        end
    default:begin
            nextstate=RESET;
            end
    endcase
end
endmodule
```

- Example:
  - Mealy edge detection state machine
    - Read continuous logic level input signal, and determine when rising and falling edges occur
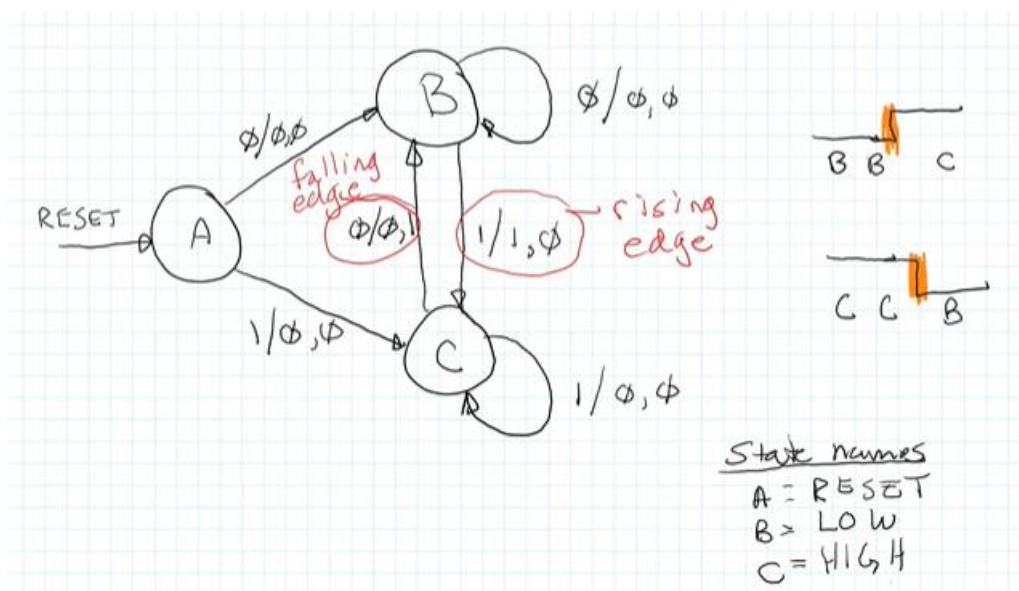

```verilog
module MealyEdgeDetect(clk, in, reset, f_e, r_e);
localparam RESET=0, LOW=1, HIGH=2; //or change it to parameter and upon the module
input clk, in, reset;
```

```verilog
output reg f_e, r_e;

reg [1:0] state, nextstate;
always @ (posedge clk)
begin
    if(reset==1'b1)
    begin
        state<=RESET;
    end
    else
    begin
        state<=nextstate;
    end
end

always @ (*)//next state and output functions
begin
    nextstate=state;//default set next state to current state
    r_e=1'b0;
    f_e=1'b0;

    case(state)
        RESET:begin
                if (in==1'b0)
                begin
                    nextstate<=LOW;
                end
                else
                begin
                    nextstate<=HIGH;
                end
            end
        LOW:begin
                if (in==1'b1)
                begin
                    nextstate<=HIGH;
                    r_e=1'b1; //set rising edge flag
                end
            end
        HIGH:begin
```

```verilog
                if (in==1'b0)
                begin
                    nextstate<=LOW;
                    f_e=1'b1;
                end
            end


        default:begin
                nextstate=RESET;
                end
    endcase
end
endmodule
```

3-8 译码器

（1）verilog 程序

```verilog
module decoder_38(A,Y);
input [2:0] A;
output [7:0] Y;


////use assign  数据流
//assign Y[0]=~A[2]&~A[1]&~A[0];
//assign Y[1]=~A[2]&~A[1]&A[0];
//assign Y[2]=~A[2]&A[1]&~A[0];
//assign Y[3]=~A[2]&A[1]&A[0];
//assign Y[4]=A[2]&~A[1]&~A[0];
//assign Y[5]=A[2]&~A[1]&A[0];
//assign Y[6]=A[2]&A[1]&~A[0];
//assign Y[7]=A[2]&A[1]&A[0];


////use for loop  行为级
//reg [7:0] Y;
//integer cnt;
//always @ (A)
//begin
//      for(cnt=0; cnt<8; cnt=cnt+1) //set the value for each bit of the output
//          begin
//                  if(cnt==A) //check if current bit location equals to the input value
//                  begin
//                      Y[cnt]=1'b1;
//                  end
//              else
//              begin
//                      Y[cnt]=1'b0;
//              end
//          end
//      end


////use continuous assignment  数据流建模
//assign Y=8'b0000_0001<<A;

//use case statement with procedural block  行为级建模
reg [7:0] Y;
```

```verilog
always @(A)
    begin
        case(A)
            3'b000: Y=8'b0000_0001;
            3'b001: Y=8'b0000_0010;
            3'b010: Y=8'b0000_0100;
            3'b011: Y=8'b0000_1000;
            3'b100: Y=8'b0001_0000;
            3'b101: Y=8'b0010_0000;
            3'b110: Y=8'b0100_0000;
            3'b111: Y=8'b1000_0000;
            default:Y=8'b0000_0000;
        endcase
    end

endmodule
```

（2）testbench 程序
```verilog
module test38();
reg [2:0] a;
wire [7:0] y;
decoder_38 uu1 (a,y);
initial
begin
a=3'b000;
#10 a=3'b001;
#10 a=3'b010;
#10 a=3'b011;
#10 a=3'b100;
#10 a=3'b101;
#10 a=3'b110;
#10 a=3'b111;
#10 $finish;
end
endmodule
```

4 位扭环形计数器

Verilog 程序：

```
module johnson2(CLR, CLK, Q);
input CLR, CLK;
output reg [3:0] Q;
always @ (posedge CLK or posedge CLR)
if(CLR)
Q=4'b0000;
else
Q<={~Q[0],Q[3:1]};
endmodule
```

Testbench 程序：

```
module testjohnson2();
wire [3:0] q;
reg clk, clr;
johnson2 u2 (clr, clk, q);
initial
begin
clk=0; clr=0;
#10 clr=1;
#10 clr=0;clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
```

```
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10 clk=~clk;
#10;
end
endmodule
```

Name	Value	0.000 ns	50.000 ns	100.000 ns	150.000 ns	200.000 ns	250.000 ns	300.000 ns
q[3:0]	f	X 0 8 c e f 7 3 1 0 8 c e						
[3]	1							
[2]	1							
[1]	1							
[0]	1							
clk	1							
clr	0							

4 位移位寄存器

Verilog 程序：

```
module yi4(CLR, CLK, D, DSL, DSR, S, Q);
input CLR, CLK, DSL, DSR;
input [3:0] D;
input [1:0] S;
output reg [3:0] Q;
always @ (posedge CLR or posedge CLK)
    if (CLR)
    Q=4'b0000;
    else
    case (S)
        2'b00: Q<=Q;
        2'b01: Q<={Q[2:0], DSR};
        2'b10: Q<={DSL, Q[3:1]};
        2'b11: Q<=D;
    endcase
endmodule
```

Testbench 程序：

```
module testyi4();
reg clr, clk, dsl, dsr;
reg [3:0] d;
reg [1:0] s;
wire [3:0] q;
yi4 uu1 (. CLR(clr), .CLK(clk), .DSL(dsl), .DSR(dsr), .D(d), .S(s), .Q(q));
initial
begin
    clr=0; clk=0; d=4'b1011; s=2'b00; dsl=1; dsr=0;
```

```
    #10 clk=~clk; s=2'b11;

    #10 clk=~clk;

    #10 clk=~clk; s=2'b10;

    #10 clk=~clk;

    #10 clk=~clk; s=2'b01;

    #10 clk=~clk;

    #10 clk=~clk; s=2'b00;

    #10 clk=~clk;

    #10 clr=1;

    #10 $finish;

    end

endmodule
```
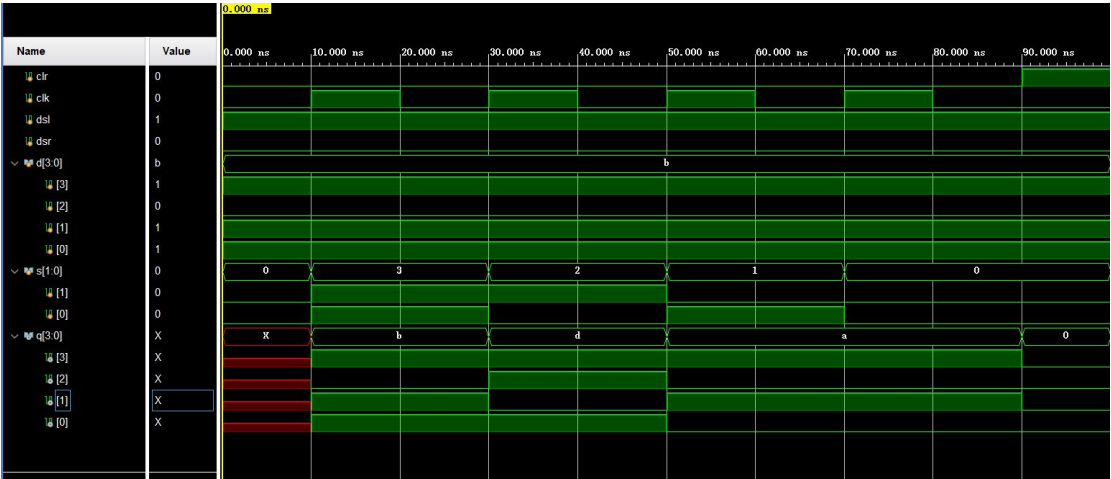
四位脉动全加器

Verilog 程序：

```
module rpcounter(CLK, ST, Q);
input CLK, ST;
output [3:0] Q;
Tff tff0(CLK, ST,Q[0]);
Tff tff1(Q[0], ST, Q[1]);
Tff tff2(Q[1], ST, Q[2]);
Tff tff3(Q[2], ST, Q[3]);
endmodule


module Tff (clk, st, qq);
input clk,st;
output qq;
Dff df (clk, st, ~qq, qq);
endmodule


module Dff(clock, reset, d, q);
input clock, reset, d;
output reg q;
always @ (negedge clock or posedge reset)
    if (reset)
        q=1'b0;
    else
        q<=d;
endmodule
```

Testbench 程序：

```
module rptest();
reg clk, st;
wire [3:0] q;
rpcounter rp (clk, st, q);
initial
begin
    clk=0; st=0;
    #10 clk=~clk; st=1;
    #10 clk=~clk; st=0;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
    #10 clk=~clk;
```

```
        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 clk=~clk;

        #10 $finish;

end

endmodule
```