Verilog

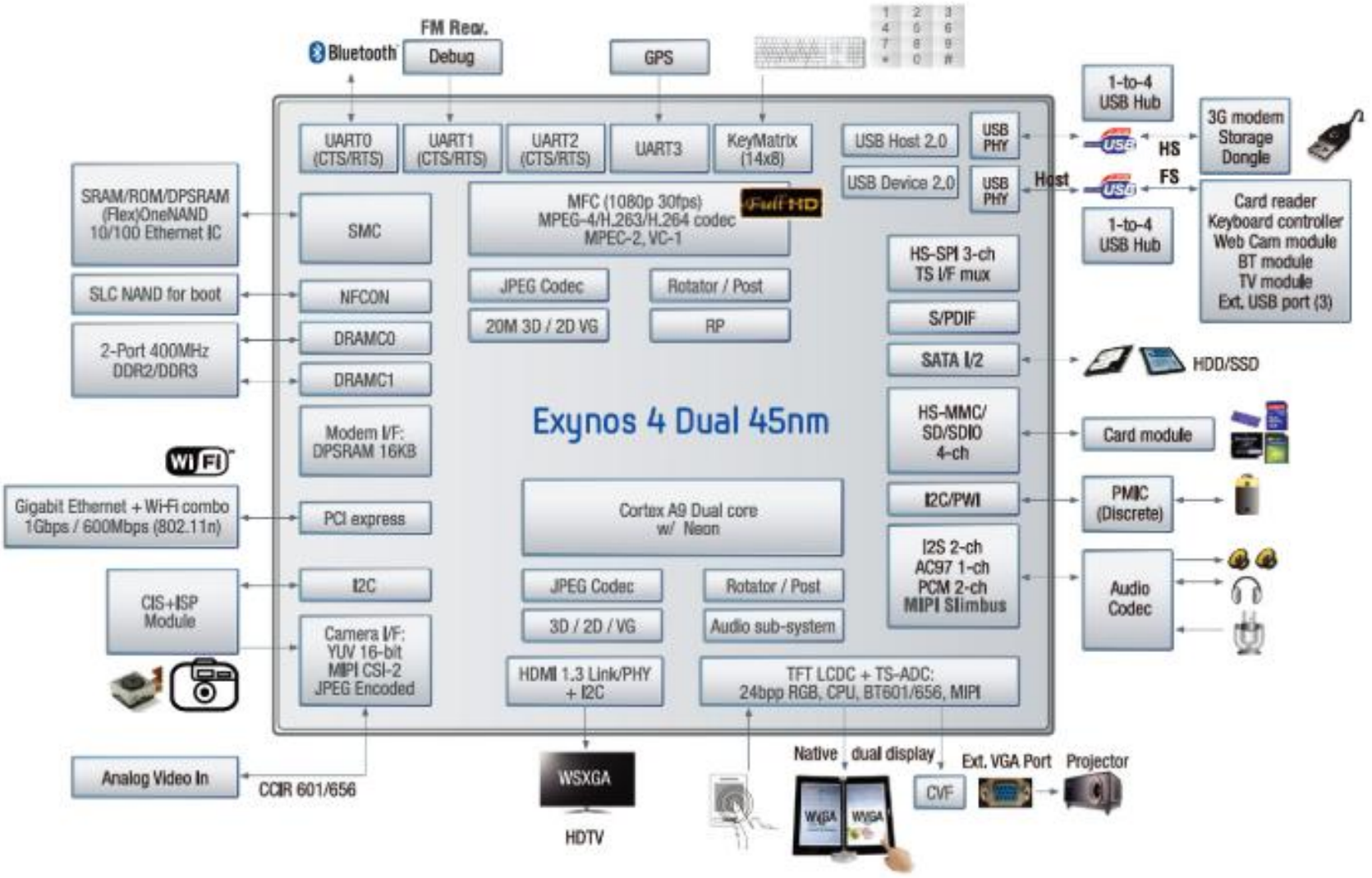
Verilog

1. 개 요
2. Verilog

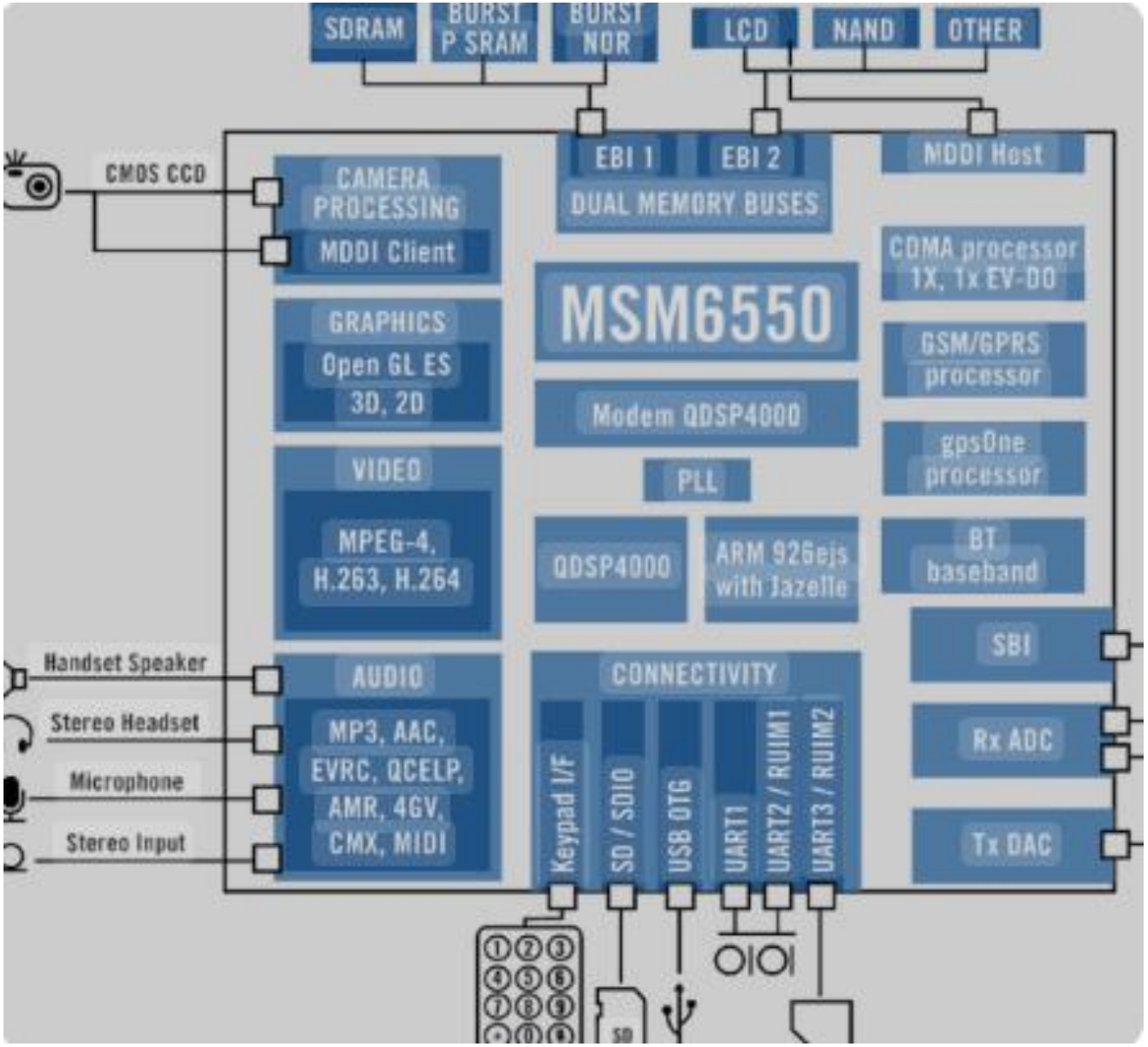
1. 개 요

Galaxy B/D

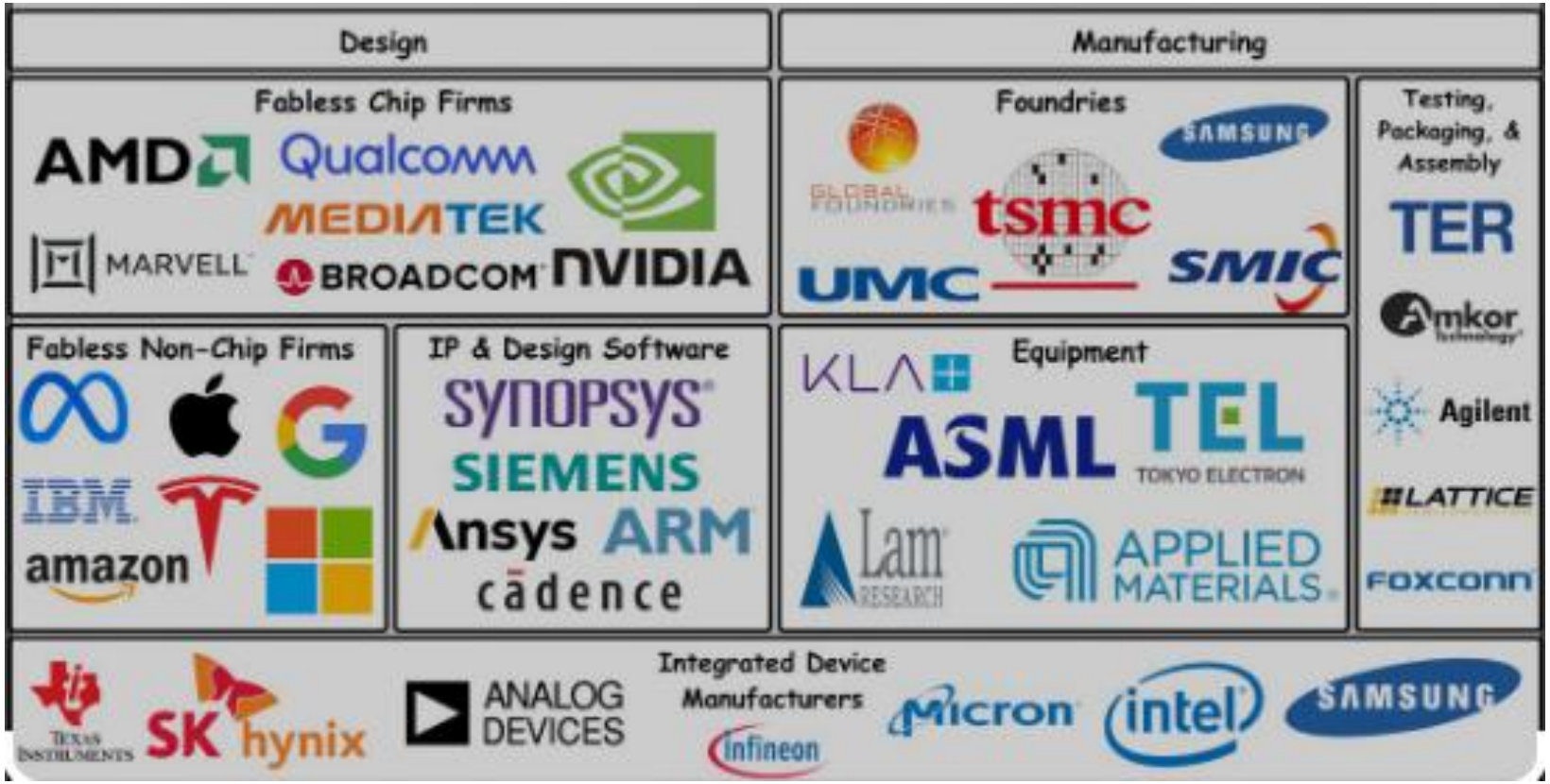
Exynos 4 (Samsung)



MSM6550 (Qualcomm)



Semiconductor Ecosystem



국내 팹리스 현황 (2020년)

		2020년		2019년		실적 상승분	
	기업명	매출	영업익	매출	영업익	매출	영업익
1	실리콘웍스	11618	942	8671	472	34.0%	99.6%
2	에이디테크놀로지	2923	134	2258	121	29.5%	10.7%
3	제주반도체	1829	47	1621	61	12.8%	-23.0%
4	어보브반도체	1441	176	1268	139	13.6%	26.6%
5	아나패스	1009	-224	608	-236	66.0%	적자지속
6	텔레칩스	1007	-84	1321	76	-23.8%	적자전환
7	앤씨앤	860	-119	784	-115	9.7%	적자지속
8	동운아나텍	707	22	547	-21	29.3%	흑자전환
9	알파홀딩스	643	-69	731	-36	-12.0%	적자지속
10	피델릭스	586	-37	560	-4	4.6%	적자지속
11	아이에이	584	44	678	27	-13.9%	63.0%
12	픽셀플러스	390	8	353	-83	10.5%	흑자전환
13	티엘아이	367	-78	419	2	-12.4%	적자전환
14	아이앤씨	228	-56	455	43	-49.9%	적자전환
15	에이디칩스	201	-28	176	-26	14.2%	적자지속
16	칩스앤미디어	154	23	161	35	-4.3%	-34.3%
17	엘디티	120	15	69	-13	73.9%	흑자전환

(단위:억원)

- 파두, 넥스트칩, 라닉스, 에티포스, 와이젯, 에이직랜드, 세미파이브
- 퓨리오사AI, 리벨리온, 사피온, 딥엑스

삼성, 수율 안정화 궤도 올랐다 …
4나노 파운드리 中 고객 문전성시

삼성전자 파운드리사업부가 4나노미터(nm) 공정에서 딥시크가 촉발한 중국 AI(인공지능) 고객사를 유치하며 활로를 찾고 있다. 무조건적인 나노 경쟁보다는 안정적인 수율을 확보하는데 우선순위를 둔 삼성 파운드리의 전략 변화가 효과를 보기 시작했다는 평가가 나온다.

20일 반도체업계에 따르면 삼성 파운드리는 최근 4나노 공정에서 중국 주문형 반도체(ASIC) 고객사를 잇따라 확보하면서 기대 이상의 성과를 내고 있다.

특히 중국 AI 스타트업인 딥시크가 내놓은 AI 모델이 주목받으면서 중국 ASIC 고객들의 러브콜이 이어지는 것으로 보인다. 앞서서도 삼성 파운드리는 3나노 첫 고객사를 중국에서 유치했을만큼 중국시장에서 인정받았고 최근 딥시크를 계기로 중국 AI 기업들이 투자에 속도를 내면서 또 다시 삼성 파운드리를 찾는 고객사들이 늘고 있는 것으로 해석된다.

삼성 파운드리가 최선단 공정인 3나노가 아닌 4나노 공정에서 고객사 확보에 성과를 내는 비결은 다름 아닌 '수율'이다. 지난해 4나노 공정에서 수율을 70% 이상으로 끌어올린데 이어 올해도 80%에 육박하는 수준의 수율 달성에 성공하면서 4나노 분야에선 자신감을 완전히 찾았다는 분석이 나온다.

트렌드포스 등 시장조사업체들은 이미 지난 2023년 삼성 파운드리가 4나노 공정에서 수율을 70%대까지 끌어올렸다고 봤지만 이후에도 시장의 반응은 미지근했다. 안정적인 수율을 확보하고도 결국 핵심은 고객사를 유치하는 것에 있다는 의미이기도 하다.

현재 TSMC와 삼성 모두 2나노 양산 경쟁에 돌입한 상황이다. 양사 모두 올해 양산을 목표로 막바지 수율 확보에 한창인 것으로 알려졌다. 삼성이 3나노에서 원하는 수율에 도달하지 못해 고전했던 경험과 4나노에서의 성과를 기반으로 2나노에서도 양산 준비에 속도를 내지만 이전 3나노 때보다 안정적인 수율을 낼 수 있는 수준에서 양산을 공식화할 것으로 업계는 예상하고 있다.

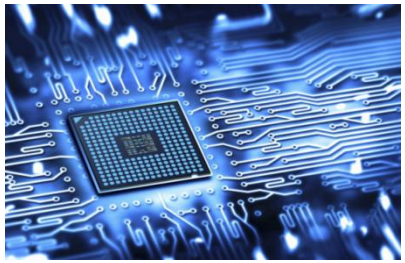
Ex) ICT&IoT 기술 구조

L7	응용 계층 (Application Layer)
L6	표현 계층 (Presentation Layer)
L5	세션 계층 (Session Layer)
L4	전송 계층 (Transport Layer)
L3	네트워크 계층 (Network Layer)
L2	데이터 링크 계층 (Data Link Layer)
L1	물리 계층 (Physical Layer)

<OSI 7-Layer>

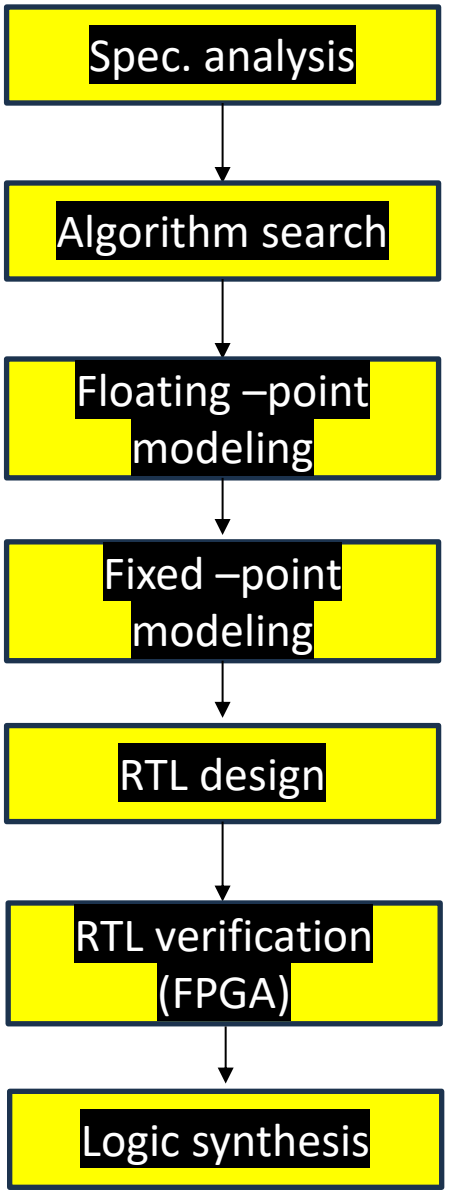
✓ Software
: 응용 기술
(사용자 인터페이스, 앱, 통계)

✓ Middleware
: Hardware 제어 기술
(제어 알고리즘)

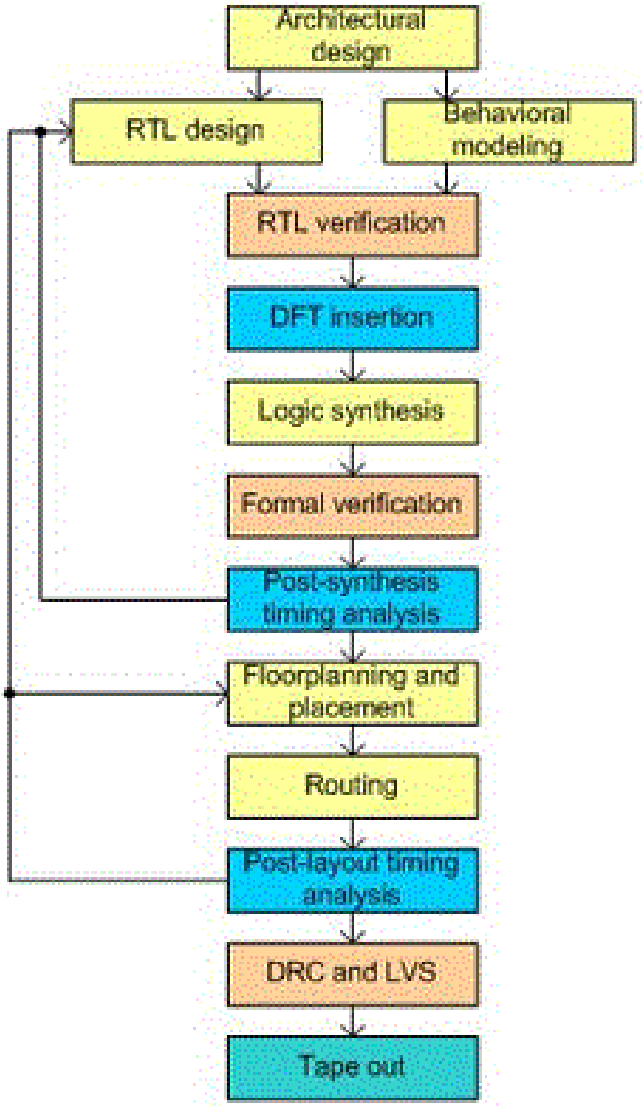


✓ Hardware
: 원천 기술
(프로토콜, 신호처리 알고리즘)

Front-end process (@Fabless)



Back-end process (@Design House)



2. Verilog

QUIZ :

```
module ham_7_4_enc (
clk,
reset,
data_in,
dvin,
code,
dvout);

input clk, reset;
input [3:0] data_in;
input dvin;

output [6:0] code;
output dvout;

wire [6:0] code;
wire dvout;

wire [3:0] datareg;
wire dvinreg;

////////////////////////////////////////////
////// matrix multiplication ////////////////
////////////////////////////////////////////
reg c0 = datareg[3]^datareg[2]^datareg[0];
reg c1 = datareg[3]^datareg[1]^datareg[0];
reg c2 = datareg[3];
reg c3 = datareg[2]^datareg[1]^datareg[0];
reg c4 = datareg[2];
reg c5 = datareg[1];
reg c6 = datareg[0];

////////////////////////////////////////////
////////////// Getting inputs ///////////////
////////////////////////////////////////////
always@(posedge clk or negedge reset)
if (!reset)
        dvinreg<=1;
else
        dvinreg<=dvin;

always@(posedge clk)
if (reset)
        datareg<=0;
else if (!dvin)
        datareg<=data_in;

////////////////////////////////////////////
// Codeword is generated 1-clock cycle /////
// after the inputs are registered.    /////
////////////////////////////////////////////
always@(posedge clk)
if (reset)
        dvout<=1;
else
        dvout<=dvinreg;

always@(posedge clk)
if (reset)
        code=0;
else
        code={c6, c5, c4, c3, c2, c1, c0};

endmodule
```

initial vs. always

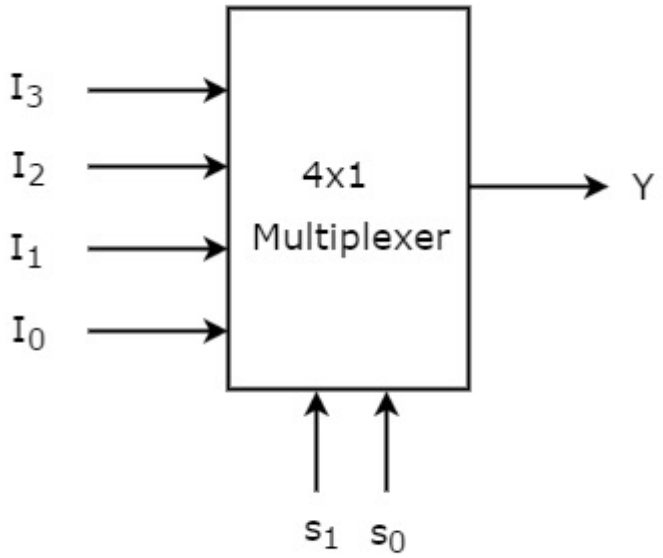
- initial
  시간 0에서 시작, simulation 동안 한 번만 수행

- always
  시간 0에서 시작, always block 내 문장을 loop 형식으로 연속되게 수행

```
module clk_gen(
    output reg clk
    );

    // 단위 시간 0에 클럭을 초기화 한다.
    initial
        clk = 1'b0;

    // 매 1/2 주기마다 클럭이 바뀐다. (T = 20)
    always
        #10 clk = ~clk;
    initial
        #1000 $finish
endmodule
```

Combinational Logic (1)

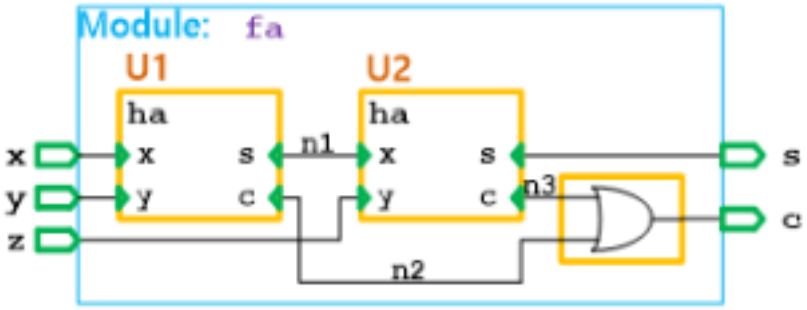
```
module mux_1(i0, i1, i2, i3, sel, y);
	input i0, i1, i2, i3;
	input [1:0] sel;
	output y;

	reg y;

	always @(sel, i0, i1, i2, i3)
		case (sel)
			0 : y = i0;
			1 : y = i1;
			2 : y = i2;
			default : y = i3;
		endcase

endmodule
```



4:1 MUX 블록도

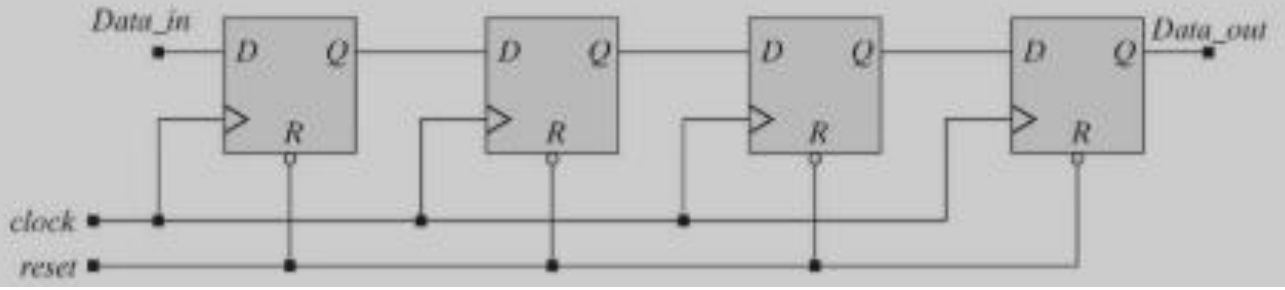
Combinational Logic (2)

```
/////-----------------------------------------/////
module fa (
/////-----------------------------------------/////
    input       x, y, z,
    output      s, c
);

    wire    n1, n2, n3;
    assign  c = n2 | n3;
/////-----------------------------------------/////
    ha U1 (.x(x),  .y(y), .s(n1),  .c(n2));
    ha U2 (.x(n1), .y(z), .s(s),   .c(n3));

endmodule
```



Sequential Logic

```
module Shift_Register #(parameter n = 4)
  (input Data_in, clock, reset, output Data_out);
  reg [n-1:0] Q;
  assign Data_out = Q[0];
  always @(posedge clock, negedge reset) // Asynchronous reset
    if (!reset) Q <= 0;                  // Active Low reset
    else Q <= {Data_in, Q[n-1:1]};       // Shifts to the right
endmodule
```



Verilog Operator (1)

- Arithmetic Operator

Operator	Description
a + b	a plus b
a - b	a minus b
a * b	a muliplied by b
a / b	a divided by b
a % b	a modulo b
a ** b	a to the power by b

Verilog Operator (2)

- Relational Operator

Operator	Description
a < b	a less than b
a > b	a greater than b
a <= b	a less than b or equal to b
a >= b	a greater than b or equal to b

Verilog Operator (3)

- Equality Operator

Operator	Description
a === b	a equal to b, including x and z
a !== b	a not equal to b, including x and z
a == b	a equal to b, result can be unknown
a != b	a not equal to b, result can be unknown

Verilog Operator (4)

- Logical Operator

Operator	Description
a && b	evaluates to true if a and b are true
a || b	evaluates to true if a or b are true
!a	converts non-zero value to zero, and vice versa

Verilog Operator (5)

- Bitwise Operator

Operator	Description
a & b	evaluates to true if a and b are true with its corresponding bit
a | b	evaluates to true if a or b are true with its corresponding bit
a ^ b	evaluates to true if a xor b are true with its corresponding bit

Verilog Operator (6)

- Shift Operator

Operator	Description
<<	Logical Shift Left
>>	Logical Shift Right
<<<	Arithmetic Shift Left
>>>	Arithmetic Shift Right

Reg vs. Wire (1)

- Wire
  물리적 연결선으로 인식, Continuous Assignment

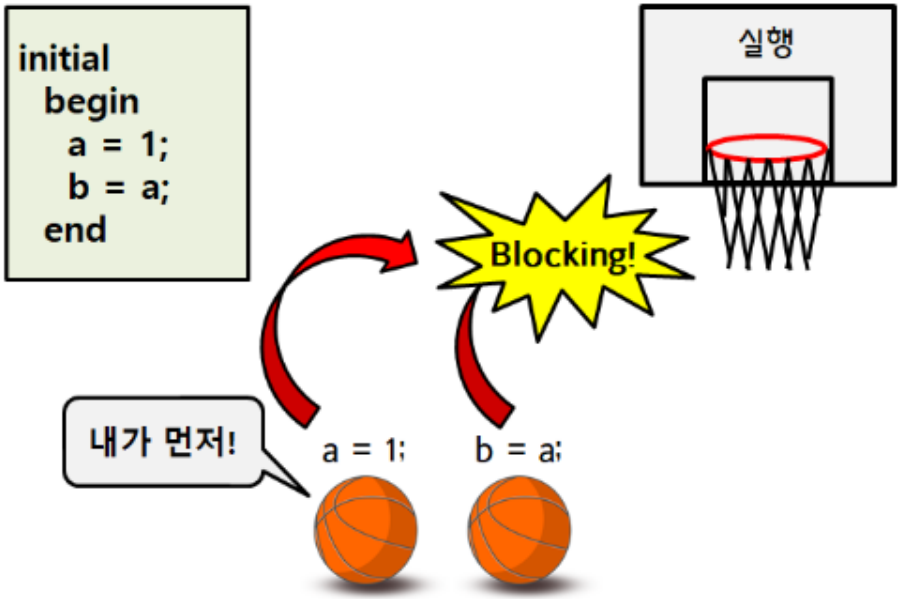
```
// 1. continuous assignment

wire in_0, in_1, out;

assign out = in_0 & in_1;
```

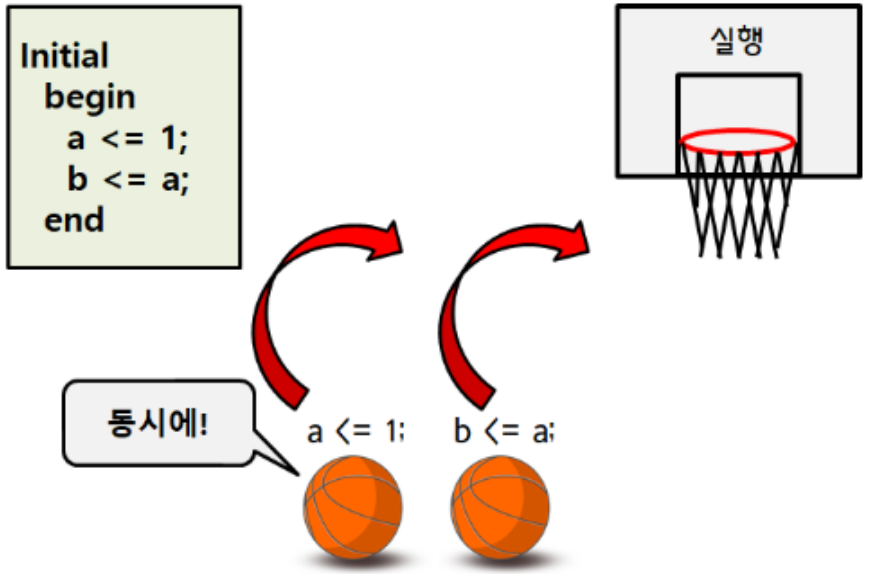
Reg vs. Wire (2)

- Reg
  다음 값이 할당되기까지 현재 값 유지
  (Sequential logic), Procedural Assignment
  Combinational logic을 reg로 구현하면,
  wire 처럼 합성됨

```
// 2. procedural assignment
// ex1) F/F
reg Q;
wire D;
always @(posedge clk or negedge reset_i_n) begin
if (reset_i_n)
   Q <= 1'b0;
else
   Q <= D;
end

// ex2) mux
reg out;
wire sel, in_0, in_1;
always (*) begin
if (sel == 0)
   out = in_0;
else // sel == 1
   out = in_1;
end
```

Blocking vs. Non-blocking (1)





Blocking (=) :
time-stamp에서 선언된 순서대로

Non-Blocking (<=) :
time-stamp에서 순서 관계없이 동시에

Blocking vs. Non-blocking (2)

```
module blocking(
    input a,
    output reg b, c, d, e
    );

    always @ (a) begin
        b = a; // blocking
        c = b;
        d = c;
        e = d;
    end

endmodule
```

```
module non_blocking (
      input clk,
      input en,
      input a,
      output b, c, d, e
);

always @(posedge clk) begin
  if (en) begin
                b <= a; // Non-blocking
                c <= b;
                d <= c;
                e <= d;
 end
end

endmodule
```

Latch vs. Flip-Flop

```
module dlatch_rst(rst,clk,d,q);
    input rst,clk,d;
    output reg q;

    always@(*)begin
        if(!rst) q=1'b0;
        else if(clk) q = d;
    end
endmodule
```

```
module dff_sync_rst(clk,d,rst_n,q,qb);
    input clk,d,rst_n;
    output reg q;
    output qb;

    assign qb = ~q;

    always@(posedge clk)begin
        if(!rst_n) q <= 1'b0;
        else q <= d;
    end
endmodule
```

Latch Problem

```
module mux4_1 (
      input [1:0] sel,
      input [1:0] a,
      output [1:0] c
);

always @(*) begin
  case (sel)
     2'b00 : c = 2'd0;
     2'b01 : c = 2'd0;
     2'b10 : c = 2'd1;
endcase
end

endmodule
```

Latch (O)

```
module mux4_1 (
      input [1:0] sel,
      input [1:0] a,
      output [1:0] c
);

always @(*) begin
  case (sel)
     2'b00 : c = 2'd0;
     2'b01 : c = 2'd0;
     2'b10 : c = 2'd1;
     2'b11 : c = 2'd2;
     default : c = 2'd0;
endcase
end

endmodule
```

Latch (X)

* Combinational logic에서 else 조건을 반드시 기재해 주어야 Latch 방지 !!!

Counter Design (1)

```
module counter1(
    input clk, rst,
    output [3:0] cnt
);

reg [3:0] count;

assign cnt = count;

always @ (posedge clk or posedge rst) begin
    if (rst) begin
        count <= 4'b0;
    end
    else begin
        count <= count + 4'b1;
    end
end

endmodule
```

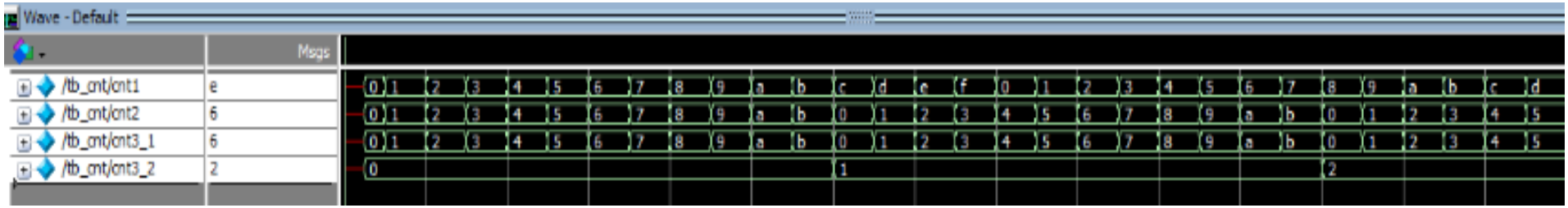
```
module counter2(
    input clk, rst,
    output [3:0] cnt
);

reg [3:0] count;

assign cnt = count;

always @ (posedge clk or posedge rst) begin
    if (rst) begin
        count <= 4'b0;
    end
    else begin
        if (count == 4'd11) begin
            count <= 4'b0;
        end
        else begin
            count <= count + 4'b1;
        end
    end
end

endmodule
```

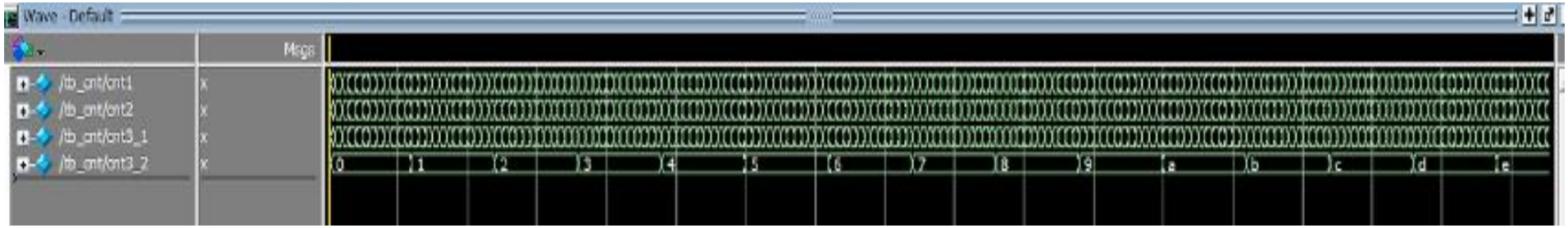
Counter Design (2)

```
module counter3(
    input clk, rst,
    output [3:0] cnt1, cnt2
);

reg [3:0] count1, count2;

assign cnt1 = count1;
assign cnt2 = count2;

always @ (posedge clk or posedge rst) begin
    if (rst) begin
        count1 <= 4'b0;
        count2 <= 4'b0;
    end
    else begin
        if (count1 == 4'd11) begin
            count1 <= 4'b0;
            if (count2 == 4'd14) begin
                count2 <= 4'b0;
            end
            else begin
                count2 <= count2 + 4'b1;
            end
        end
        else begin
            count1 <= count1 + 4'b1;
        end
    end
end

endmodule
```

```
`timescale 1ns/1ps

module tb_cnt();

reg clk, rst;
wire [3:0] cnt1, cnt2, cnt3_1, cnt3_2;

initial begin
    clk <= 1'b1;
    rst <= 1'b0;
    #5
    rst <= 1'b1;
    #5
    rst <= 1'b0;
    #400
    $finish;
end

counter1 TEST1(clk, rst, cnt1);
counter2 TEST2(clk, rst, cnt2);
counter3 TEST3(clk, rst, cnt3_1, cnt3_2);

always #5 clk <= ~clk;

endmodule
```

Counter Design (3)





Shift Register Design (1)

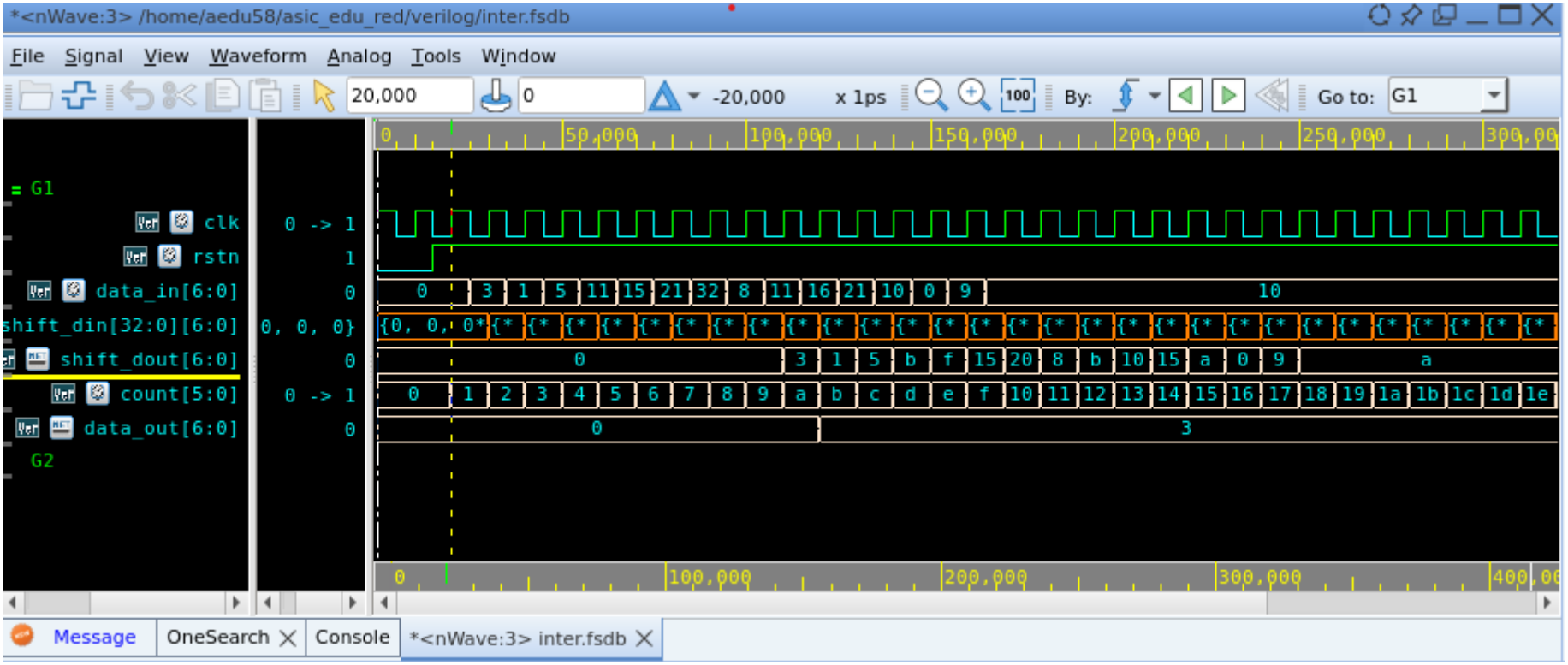
```
`timescale 1ns / 1ps

module shift_reg #(
    parameter WIDTH = 7
)(
    input   clk,
    input   rstn,

    input   signed [WIDTH-1:0]  data_in,
    output  reg signed [WIDTH-1:0]  data_out
);


reg signed [WIDTH-1:0] shift_din [32:0];
integer i;
always@(posedge clk or negedge rstn) begin
 if (~rstn) begin
  for(i = 32; i >= 0; i=i-1) begin
        shift_din[i] <= 0;
  end
 end
 else begin
  for(i = 32; i > 0; i=i-1) begin
        shift_din[i] <= shift_din[i-1];
  end
        shift_din[0] <= data_in;
 end
end

wire [WIDTH-1:0] shift_dout;
//assign shift_dout = shift_din[32];
assign shift_dout = shift_din[8];
```

```
reg [5:0] count;
always @(posedge clk or negedge rstn) begin
 if (~rstn) begin
        count <= 4'b0;
 end
 else begin
        count <= count + 4'b1;
 end
end

reg [WIDTH-1:0] data_out;
always @(posedge clk or negedge rstn) begin
 if (~rstn) begin
        data_out <= 4'b0;
 end
 else if (count==6'd10) begin
        data_out <= shift_dout;
 end
end

endmodule
```

Shift Register Design (2)

```
`timescale 1ns/10ps

module tb_shift_reg();

reg clk,rstn;
reg [6:0] data_in;
wire [6:0] data_out;

initial begin
 clk <= 1'b1;
 rstn <= 1'b0;
 #15 rstn <= 1'b1;
 #400 $finish;
end

initial begin
 data_in <= 7'd0;
 //#20 data_in <= 7'd3; // Abnormal
 #25 data_in <= 7'd3; // Normal
 #10 data_in <= 7'd1;
 #10 data_in <= 7'd5;
 #10 data_in <= 7'd11;
 #10 data_in <= 7'd15;
 #10 data_in <= 7'd21;
 #10 data_in <= 7'd32;
 #10 data_in <= 7'd8;
 #10 data_in <= 7'd11;
 #10 data_in <= 7'd16;
 #10 data_in <= 7'd21;
 #10 data_in <= 7'd10;
 #10 data_in <= 7'd0;
 #10 data_in <= 7'd9;
 #10 data_in <= 7'd10;
end

shift_reg #(.WIDTH(7)) i_shift_reg (
        .clk(clk),
        .rstn(rstn),
        .data_in(data_in),
        .data_out(data_out));

always #5 clk <= ~clk;

endmodule
```

Shift Register Design (3)



Shift Register Design (4)

```
`timescale 1ns/10ps

module tb_shift_reg();

reg clk,rstn;
reg [6:0] data_in;
wire [6:0] data_out;

initial begin
 clk <= 1'b1;
 rstn <= 1'b0;
 #15 rstn <= 1'b1;
 #400 $finish;
end

initial begin
 data_in <= 7'd0;
 #20 data_in <= 7'd3; // Abnormal
 //#25 data_in <= 7'd3; // Normal
 #10 data_in <= 7'd1;
 #10 data_in <= 7'd5;
 #10 data_in <= 7'd11;
 #10 data_in <= 7'd15;
 #10 data_in <= 7'd21;
 #10 data_in <= 7'd32;
 #10 data_in <= 7'd8;
 #10 data_in <= 7'd11;
 #10 data_in <= 7'd16;
 #10 data_in <= 7'd21;
 #10 data_in <= 7'd10;
 #10 data_in <= 7'd0;
 #10 data_in <= 7'd9;
 #10 data_in <= 7'd10;
end

shift_reg #(.WIDTH(7)) i_shift_reg (
        .clk(clk),
        .rstn(rstn),
        .data_in(data_in),
        .data_out(data_out));

always #5 clk <= ~clk;

endmodule
```

Shift Register Design (5)