

# Chapter 5

---

## 조합회로 모델링

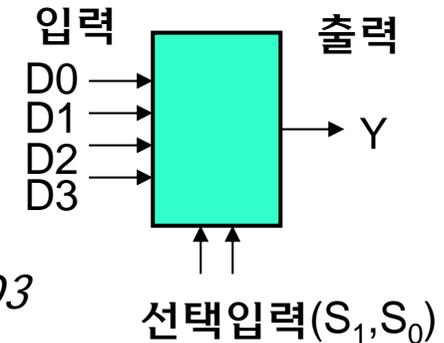


## 5.1 Multiplexer

### ■ Multiplexer

- 여러 개( $2^m$ )의 입력과 한 개의 출력이 있고 선택 입력(m-bit)에 의해서 한 개의 입력을 선택하여 출력에 전달하는 조합회로
- Boolean equation:

$$Y = S_1'S_0'D0 + S_1'S_0D1 + S_1S_0'D2 + S_1S_0D3$$

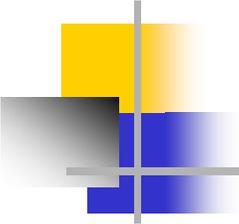


### ■ 4x1 multiplexer: continuous assignment model

```
module mux4(y, d3, d2, d1, d0, s);
  output y;
  input d3, d2, d1, d0;
  input [1:0] s;

  assign y = ~s[1] & ~s[0] & d0 | ~s[1] & s[0] & d1 |
            s[1] & ~s[0] & d2 | s[1] & s[0] & d3;
endmodule
```





## Multiplexer (cont')

- continuous assignment model (2)

```
module mux4(y, d3, d2, d1, d0, s);  
  output y;  
  input d3, d2, d1, d0;  
  input [1:0] s;  
  
  assign y = (s==2'b00) & d0 | (s==2'b01) & d1 |  
            (s==2'b10) & d2 | (s==2'b11) & d3;  
endmodule
```

- 앞의 모델과 이 모델은 1-bit 4x1 multiplexer 모델을 n-bit 4x1 multiplexer로 확장하기 어려움



## Multiplexer (cont')

- conditional operator를 사용한 model

```
module mux4_32(y, d3, d2, d1, d0, sel);  
  output [31:0] y;  
  input [31:0] d3, d2, d1, d0;  
  input [1:0] sel;  
  
  assign y = (sel==0) ? d0 :  
             (sel==1) ? d1 :  
             (sel==2) ? d2 :  
             (sel==3) ? d3 : 32'bx;  
  
end  
endmodule
```



## Multiplexer (cont')

### ■ if – else 문을 사용한 model

```
module mux4_32(y, d3, d2, d1, d0, sel);
  output [31:0] y;
  input [31:0] d3, d2, d1, d0;
  input [1:0] sel;
  reg [31:0] y;

  always @(d3 or d2 or d1 or d0 or sel) begin
    if (sel==0) y = d0;
    else if (sel==1) y = d1;
    else if (sel==2) y = d2;
    else if (sel==3) y = d3;
    else y = 32'bx;
  end
endmodule
```

조합회로는 모든  
입력이 포함됨

- 반드시 있어야 함
- 없으면 기억소자가 포함될 수 있음



## case문과 Multiplexer

- case문을 사용한 model: **case ... endcase**

```
module mux4_32(y, d3, d2, d1, d0, sel);  
  output [31:0] y;  
  input [31:0] d3, d2, d1, d0;  
  input [1:0] sel;  
  reg [31:0] y;  
  
  always @(d3 or d2 or d1 or d0 or sel) begin  
    case (sel)  
      0: y = d0;  
      1: y = d1;  
      2: y = d2;  
      3: y = d3;  
      default: y = 32'bx;  
    endcase  
  end  
endmodule
```

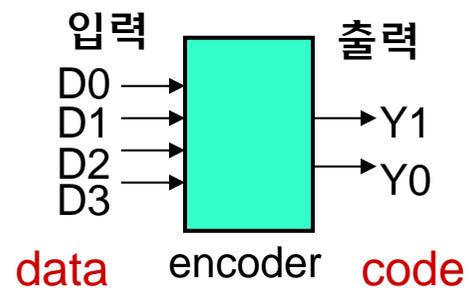


## 5.2 Encoder

### ■ Encoder

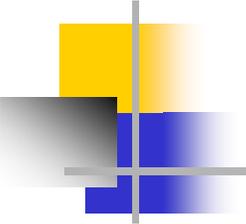
- $n$ 개의 입력과  $m$ 개의 출력( $n \leq 2^m$ )이 있는 회로로서 각 입력에 대해서  $m$ -bit의 출력 코드를 부여하고 해당 입력이 1일 때에 부여된 코드를 출력함
- 여러 개의 입력이 동시에 1인 경우는 없음: (예) decoder 출력

### ■ 예: 4x2 encoder



D0	D1	D2	D3	Y1	Y0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1



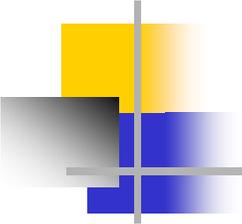


## Encoder (cont')

- if – else문을 사용한 model

```
module encoder4x2(code, data);  
  output [1:0] code;  
  input [3:0] data;  
  reg [1:0] code;  
  
  always @(data) begin  
    if (data==4'b0001) code = 0;  
    else if (data==4'b0010) code = 1;  
    else if (data==4'b0100) code = 2;  
    else if (data==4'b1000) code = 3;  
    else code = 2'bx;  
  end  
endmodule
```





## Encoder (cont')

- case문을 사용한 model

```
module encoder4x2(code, data);  
  output [1:0] code;  
  input [3:0] data;  
  reg [1:0] code;  
  
  always @(data) begin  
    case (data)  
      4'b0001: code = 2'b00;  
      4'b0010: code = 2'b01;  
      4'b0100: code = 2'b10;  
      4'b1000: code = 2'b11;  
      default: code = 2'bx;  
    endcase  
  end  
endmodule
```



# Priority Encoder

- priority encoder
  - 한 입력만이 1인 제한 조건 없으며 여러 개의 입력이 동시에 1인 경우에는 가장 우선순위가 높은 입력에 대한 코드를 출력
- if – else를 사용한 model

```
module priority4x2(code, data);  
  output [1:0] code;  
  input [3:0] data;  
  reg [1:0] code;  
  
  always @(data) begin  
    if (data[0]) code = 0;  
    else if (data[1]) code = 1;  
    else if (data[2]) code = 2;  
    else if (data[3]) code = 3;  
    else code = 2'bx;  
  end  
endmodule
```

data[0]가 가장  
우선순위가 높음



## casex과 casez 문

- case문
  - 값을 비교할 때에 0, 1, x, z를 정확하게 비교함
- casex문
  - x와 z를 don't care로 처리함
- casez문
  - z를 don't care로 처리함 (z대신 ?를 사용할 수 있음)
  - 0,1,x는 정확하게 비교함
- priority encoder의 진리표

D0	D1	D2	D3	Y1	Y0
1	x	x	x	0	0
0	1	x	x	0	1
0	0	1	x	1	0
0	0	0	1	1	1



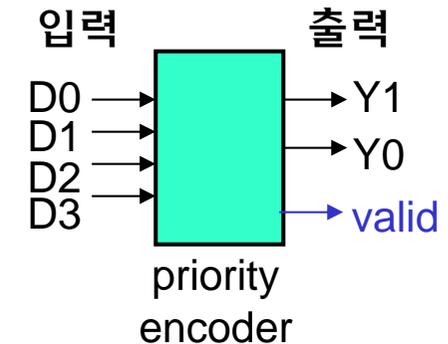
# case문과 Priority Encoder

case문은 0,1,x,z을 정확히 비교  
하므로 이 예처럼 사용할 수 없음

## ■ case 문을 사용한 model

```
module priority4x2(code, data, valid);
  output [1:0] code;
  input [3:0] data;
  output valid;
  reg [1:0] code;

  always @(data) begin
    casex (data)
      4'bxxx1: code = 0;
      4'bxx10: code = 1;
      4'bx100: code = 2;
      4'b1000: code = 3;
      default: code = 2'bx;
    endcase
  end
  assign valid = | data;
endmodule
```



입력에 1인 bit가 존재할 때  
valid=1임

Reduction operator:

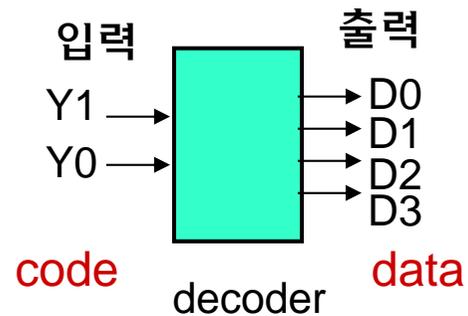
- vector의 모든 bit에 대해서 연산을 수행  
(예) &, |, ~&, ~|, ^, ~^

= data[0] | data[1] | data[2] | data[3]

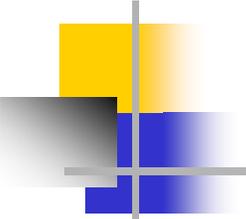


## 5.3 Decoder

- Decoder
  - encoder의 반대 동작
  - 입력 코드에 대응되는 출력을 1로 하고 나머지 출력은 0으로 함
- 예: 2x4 decoder



Y1	Y0	D0	D1	D2	D3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

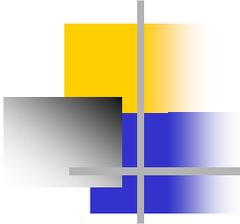


## Decoder (cont')

- if – else 문을 사용한 model

```
module decoder2x4(data, code);  
  output [3:0] data;  
  input [1:0] code;  
  reg [3:0] data;  
  
  always @(code) begin  
    if (code==0) data=4'b0001;  
    else if (code==1) data=4'b0010;  
    else if (code==2) data=4'b0100;  
    else if (code==3) data=4'b1000;  
    else data = 4'bx;  
  end  
endmodule
```





## Decoder (cont')

- case 문을 사용한 model

```
module decoder2x4(data, code);
  output [3:0] data;
  input [1:0] code;
  reg [3:0] data;

  always @(code) begin
    case (code)
      0: data=4'b0001;
      1: data=4'b0010;
      2: data=4'b0100;
      3: data=4'b1000;
      default: data = 4'bx;
    endcase
  end
endmodule
```

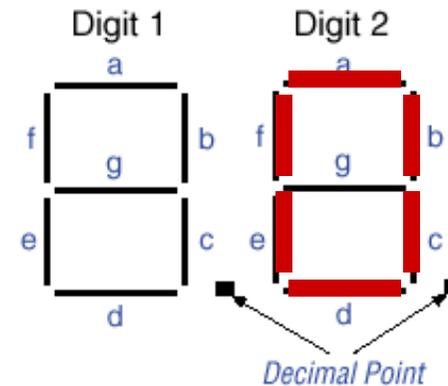


## 5.4 7-segment LED display

### ■ seven segment LED display의 model

```
module LED7seg(display, bcd);  
    output [6:0] display;  
    input [3:0] bcd;  
    reg [6:0] display;  
  
    always @(bcd) begin  
        case (bcd)  
            0: display=7'b111_1110;  
            1: display=7'b011_0000;  
            2: display=7'b110_1101;  
            3: display=7'b111_1001;  
            4: display=7'b011_0011;  
            5: display=7'b101_1011;  
            6: display=7'b101_1111;  
            7: display=7'b111_0000;  
        endcase  
    end  
endmodule
```

(계속)



abcdefg 순서로 data 부여  
display[6] → a  
display[5] → b  
...  
display[0] → g

1일 때 LED on, 0일 때 off



## 7-segment LED display (cont')

```
        8: display=7'b111_1111;  
        9: display=7'b111_1011;  
    default: display = 7'b000_0000 // blank  
endcase  
end  
endmodule
```

invalid 입력에 대해서는 LED를 off 시킴

LED가 0일 때에 ON이라면 output을 display 대신에 disp로 선언한 후에 다음 문장을 추가하면 됨.

```
assign disp = ~display
```



## 5.5 Verilog Loop

- for 문
  - while 문
  - repeat 문
- } C언어와 비슷한 형식

반복문은 initial 또는 always  
내에서만 사용 가능

```
addr = 0;  
repeat (size) begin  
    memory[addr] = 0;  
    addr = addr + 1;  
end
```

→ size번 반복 수행

- forever 문

```
clock = 0;  
forever begin  
    #5 clock = 1;  
    #5 clock = 0;  
end
```

→ 무한히 반복 수행  
disable문에 의해서 중단됨



# Majority Circuit

- Majority 회로 – 입력들 중에서 1인 것이 과반수이면 1을 출력
- (예) 4-bit Majority circuit

```
module Majority_4b (Y, A, B, C, D);  
  input  A, B, C, D;  
  output Y;  
  reg    Y;  
  
  always @ (A or B or C or D) begin  
    case ({ A, B,C, D} )  
      7, 11, 13, 14, 15: Y = 1;  
      default:          Y = 0;  
    endcase  
  end  
endmodule
```

→ 여러 경우는 ,로 나열함

단점: bit 수가 바뀌면 프로그램을 재작성해야함



# Majority Circuit – 반복문 사용

## ■ for문을 사용한 Majority circuit

```
module Majority (Y, Data);
  parameter size = 8;
  parameter max = 3;
  parameter majority = 5;
  input [size-1: 0] Data;
  output Y;
  reg Y;
  reg [max-1: 0] count;
  integer k;

  always @ (Data) begin
    count = 0;
    for (k = 0; k < size; k = k + 1) begin
      if (Data[k] == 1) count = count + 1;
    end
    Y = (count >= majority);
  end
endmodule
```

장점: 임의의 bit 수에 대해서  
사용 가능

단점: 복잡하게 합성될 수 있음  
(adder가 포함될 수 있음)



# IP 재사용과 Parameterized model

- Parameterized model
  - 변경 가능한 값을 parameter로 지정한 model
  - parameter로 지정된 값은 application에서 새로 지정 가능
- Intellectual Property(IP) Reuse
  - 다른 응용 회로에서 설계된 model을 재사용하기 위해서는 parameterized model을 사용해야 함.
- 예

parameter를 사용한 module

```
module shift(in, out);  
  input [7:0] in;  
  output [7:0] out;  
  parameter n = 1;  
  
  assign out = (in << n);  
endmodule
```

parameter 값 지정

```
module t_shift(...);  
  ...  
  shift u1(x, a);  
  shift #(2) u2(x, b);  
  shift #(4) u3(x, c);  
  ...  
endmodule
```

→ n=1  
→ n=2  
→ n=4



# Clock Generator

- forever와 disable을 사용한 clock generator 모델링
  - disable은 named block의 수행을 종료 시킴

```
reg clock;  
  
initial begin : clock_loop  
    clock = 0;  
    forever begin  
        #10 clock = 1;  
        #10 clock = 0;  
    end  
end  
  
initial  
    #200 disable clock_loop;
```

named block

주기가 20인 clock

clock\_loop 블록의 수행 종료

## Clock Generator (cont')

- always 문을 사용한 clock generator 모델링

```
reg clock;  
  
initial begin  
    clock = 0;  
    #200 $finish;  
end  
  
always  
    #10 clock = 1;  
    #10 clock = 0  
end
```

- always와 forever의 비교

- always: concurrent behavior, simulation을 시작할 때에 active
- forever: always 또는 initial 내에서 사용하며 activity flow가 도달했을 때에 수행됨, nested 가능



# Adder

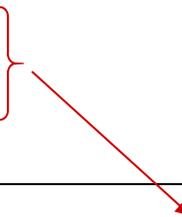
```
module adder4(sum, cout, a, b, cin);  
    output [3:0]    sum;  
    output          cout;  
    input [3:0]    a, b;  
    input          cin;  
    reg [4:0]      carry;  
    reg [3:0]      sum;  
    reg            cout;  
    integer        i;  
  
    always @ (a or b or cin) begin  
        carry[0] = cin;  
        for (i = 0; i <= 3; i = i + 1) begin  
            carry[i+1] = a[i] & b[i] | a[i] & carry[i] | b[i] & carry[i];  
            sum[i] = a[i] ^ b[i] ^ carry[i];  
        end  
        cout = carry[4];  
    end  
endmodule
```

full-adder



## Adder (cont')

```
module adder4(sum, cout, a, b, cin);  
    output [3:0]    sum;  
    output          cout;  
    input [3:0]    a, b;  
    input          cin;  
    reg [4:0]      carry;  
    integer        i;  
  
    always @ (a or b or cin) begin  
        carry[0] = cin;  
        for (i = 0; i <= 3; i = i + 1) begin  
            carry[i+1] = a[i] & b[i] | a[i] & carry[i] | b[i] & carry[i];  
        end  
    end  
    assign cout = carry[4];  
    assign sum = a ^ b ^ carry[3:0];  
endmodule
```



sum과 cout은 assign문으로 기술



## 5.7 Functions and Tasks

### ■ function과 task

- 공통적으로 반복하여 사용되는 코드를 function 또는 task로 작성함
- 코드를 반복하여 사용하는 것 대신에 function 또는 task를 호출함

```
task_name(out1, out2, inout1, in1, in2, ...);    // task 호출  
out1 = func_name(in1, in2, ...);                // function호출
```

### ■ task

- FORTRAN의 SUBROUTINE과 유사
- 인수를 통하여 값을 전달하고 결과를 전달받음
- 여러 개의 값을 받을 수 있음

### ■ function

- FORTRAN의 FUNCTION과 유사
- 인수를 통하여 값을 전달하면 함수이름을 통하여 결과가 반환됨
- 하나의 값만 받을 수 있음
- 시간지연 포함할 수 없음



# Task

## ■ Task

- module 내에서 선언됨: **task ... endtask**

```
module adder4_task(sum, cout, a, b, cin);
  output [3:0] sum;
  output cout;
  input [3:0] a, b;
  input  cin;
  reg [4:0] carry;
  integer i;

  always @ (a or b or cin) begin
    carry[0] = cin;
    for(i = 0; i <= 3; i = i + 1) begin
      gen_carry(carry[i+1], a[i], b[i], carry[i]);
    end
  end
end
```

task 호출

(계속)



## Task (cont')

```
assign cout = carry[4];
assign sum = a ^ b ^ carry[3:0];

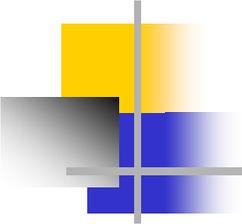
// task 정의
task gen_carry;
    output carry;
    input a, b, c;
    begin
        carry = a & b | a & c | b & c;
    end
endtask
endmodule
```

formal argument가 선언된  
순서대로 actual argument가  
대응됨

actual argument: `gen_carry(carry[i+1], a[i], b[i], carry[i]);`

formal argument:                    carry    a    b    c





# Function

## ■ function

- module 내에서 선언됨: **function ... endfunction**

```
module adder4_function(sum, cout, a, b, cin);
  output [3:0] sum;
  output cout;
  input [3:0] a, b;
  input  cin;
  reg [4:0] carry;
  integer i;

  always @ (a or b or cin) begin
    carry[0] = cin;
    for(i = 0; i <= 3; i = i + 1) begin
      carry[i+1] = gencarry(a[i], b[i], carry[i]);
    end
  end
end
```

(계속)



## Function (cont')

```
assign cout = carry[4];
assign sum = a ^ b ^ carry[3:0];

// function 정의
function gencarry;
  input a, b, c;
  begin
    gencarry = a & b | a & c | b & c;
  end
endfunction
endmodule
```

반환 값은 함수이름의  
변수에 저장됨

- 벡터값 반환 가능

```
function [7:0] func_name;           // 8-bit vector값을 반환
...
endfunction
```

