Combinational Logic I

CS207 Lecture 5

James YU

Mar. 18, 2020

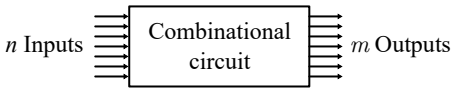
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Combinational logic

- Logic circuits for digital systems:
  - combinational logic,
  - sequential logic (next lectures).
- *Combinational*?
  - Output determined by the combination of inputs.
  - Perform an operation specified by a set (combination) of Boolean functions.

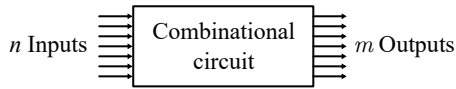
Combinational logic

- An interconnection of logic gates that
  - react to values of input signals,
  - produce output signal values,
  - n inputs from external sources, and
  - m outputs go to external destinations.



Combinational logic

- Two representations of a combinational circuit:
  - 2^n possible input combinations: truth table, or
  - m outputs: m Boolean functions, each expressed with the n inputs.



Analysis of combinational logic

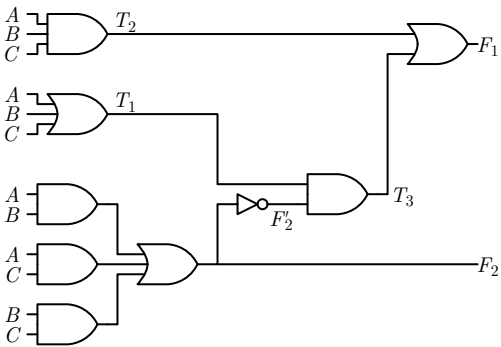
- Analysis of a combinational circuit: determine the function of the circuit.
  - Given logic diagram,
  - develop a set of Boolean functions, a truth table, an optional explanation of the circuit operation.
- If a function name or an explanation is given along the circuit, just verify if the given information is correct.

Analysis of combinational logic

- Obtain the output Boolean functions:
  - Label all gate outputs that are a function of only inputs, no other intermediate variables. Determine their Boolean functions.
  - Label all gates that are a function of inputs and the gates in the previous step. Determine their Boolean functions.
  - List output Boolean functions, squeeze the intermediate variables.

Analysis of combinational logic

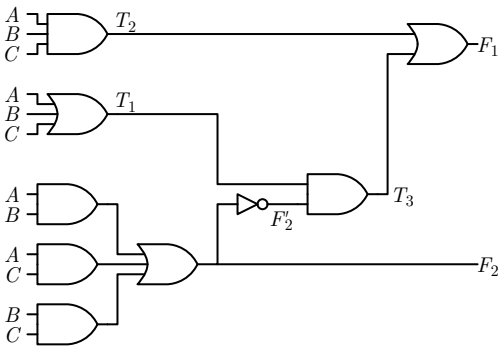
- Obtain the output Boolean functions:
  - Label all gate outputs that are a function of only inputs, no other intermediate variables. Determine their Boolean functions.



- $T_1 = A + B + C.$
- $T_2 = ABC.$
- $F_2 = AB + AC + BC.$

Analysis of combinational logic

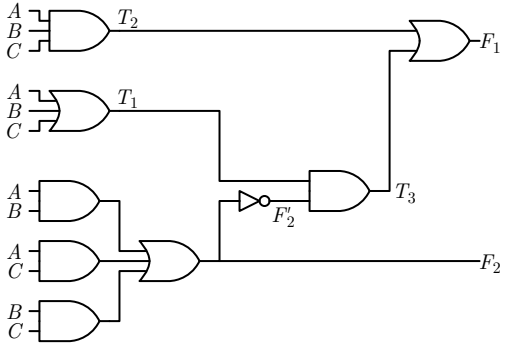
- Obtain the output Boolean functions:
  - Label all gates that are a function of inputs and the gates in the previous step. Determine their Boolean functions.



- $T_3 = F_2' + T_1.$
- $F_1 = T_2 + T_3.$

Analysis of combinational logic

- Obtain the output Boolean functions:
  - List output Boolean functions, squeeze the intermediate variables.



$$
\begin{aligned}
F_1 &= T_2 + T_3 \\
&= ABC + F_2' T_1 \\
&= ABC + \\
&\quad (AB + AC + BC)' \\
&\quad (A + B + C) \\
&= ABC + A'B'C + \\
&\quad AB'C' + A'BC'
\end{aligned}
$$

.

Analysis of combinational logic

- Truth table is simple with Boolean function
  - Determine the number of input variables. For n inputs, form the 2^n combinations from 0 to 2^n-1.
  - Label the outputs of the intermediate gates.
  - Obtain the truth table for these outputs.
  - Obtain the truth table for the remaining outputs.

A	B	C	F_2	F_2'	T_1	T_2	T_3	F_1
0	0	0	0	1	0	0	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	1	0	1	0	0	0
1	1	0	1	0	1	0	0	0
1	1	1	1	0	1	1	0	1

Design of combinational logic

- Design of a combinational circuits: develop a logic circuit diagram or a set of Boolean functions.
  - From specification of the design objective
- Involves the following steps:
  - Determine required number of inputs and outputs.
  - Derive the truth table.
  - Obtain simplified Boolean function for each output.
  - Draw logic diagram and verify the correctness.

Design of combinational logic

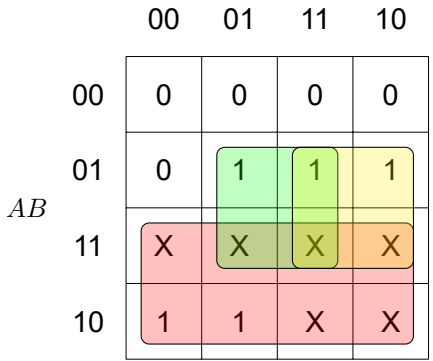
- Example: Convert from BCD decimal code to excess-3 code.

Input BCD				Output Code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

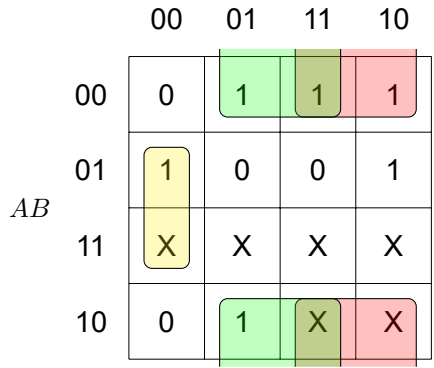
- Obviously, four inputs and four outputs
- And we already have the truth table
- Then the Boolean functions. How?
  - Remember K-maps?

Design of combinational logic

$w = A + BC + BD$



$x = B'C + B'D + BC'D'$



Design of combinational logic

$y = CD + C'D'$

AB \ CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

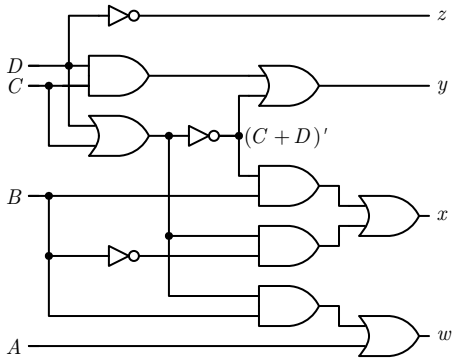
$z = D'$

AB \ CD	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	X	X	X	X
10	1	0	X	X

Design of combinational logic

- We manipulate these Boolean functions to reuse common gates:
  - $w = A + BC + BD = A + B(C + D)$.
  - $x = B'C + B'D + BC'D' = B'(C + D) + BC'D'$.
  - $y = CD + C'D' = CD + (C + D)'$.
  - $z = D'$.

Design of combinational logic

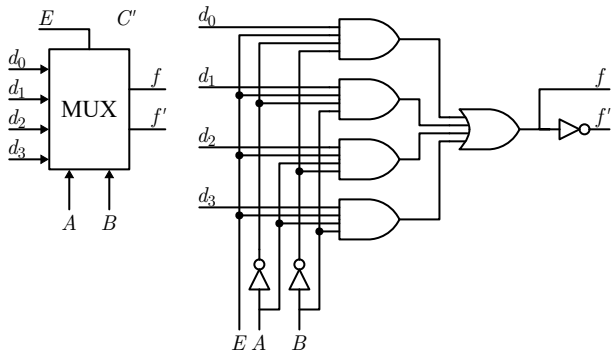


Combinational logic design with MSI circuits

- Since the introduction of MSI and LSI circuits, the traditional methods of logic design have largely been superseded.
  - Traditionally, the design engineer has developed a Boolean equation as the solution to a particular problem.
  - This function has then been minimised and implemented using SSI circuits.
- In practice, many combinational circuits may have a large number of inputs and outputs.
  - Consequently the use of truth tables in the design of such circuits is impractical.
- The development of MSI circuits has led to the technique of splitting a complex design into a number of sub-systems.

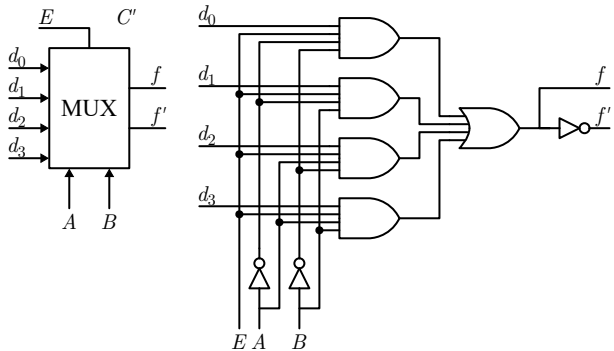
Multiplexer

- A multiplexer (MUX) selects 1-out-of-n lines where n is usually `2`, `4`, `8`, or `16`.
- A block diagram of a multiplexer having four input data lines `d0`, `d1`, `d2`, and `d3` and complementary outputs `f` and `f'`.



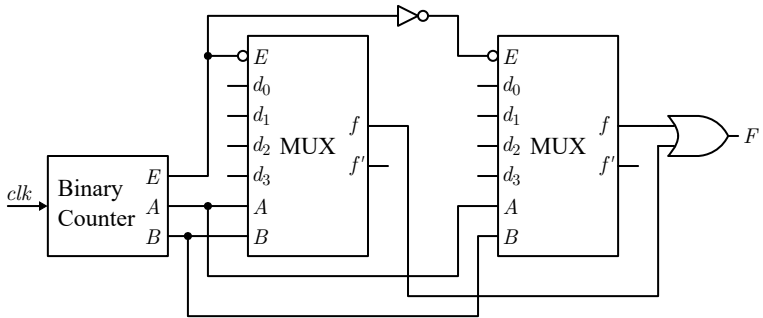
Multiplexer

- The device has two control or selection lines A and B and an enable line E.
- The characteristic equation of the multiplexer is
  $f = A'B'd_0 + A'Bd_1 + AB'd_2 + ABd_3$.



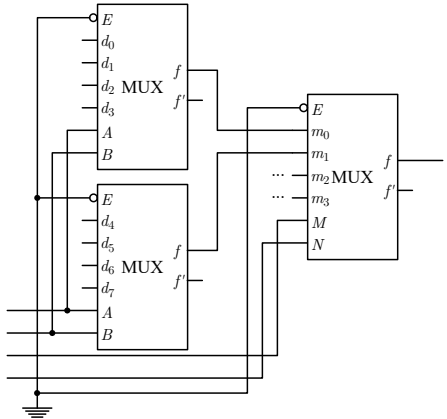
Interconnecting multiplexers

- Data within a digital system is normally processed in parallel form in order to increase the speed of operation.
- If the output of the system has to be transmitted over a relatively long distance then a parallel-to-serial conversion will take place.
- Example: An 8-bit word is presented in parallel at the data inputs.



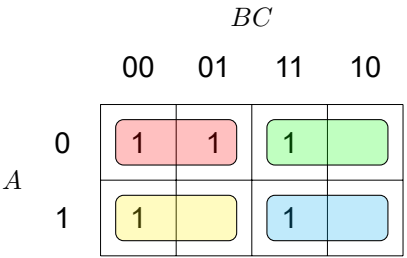
Interconnecting multiplexers

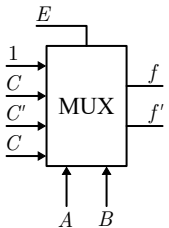
- The principle of data selection can be extended to allow the selection of 1-out-of-64 lines.
  - Using nine 8-to-1 multiplexers arranged in two levels of multiplexing.



Multiplexer as a Boolean function generator

- For a 4-to-1 MUX the characteristic equation is $f = A'B'd_0 + A'Bd_1 + AB'd_2 + ABd_3$.
  - A and B are Boolean variables, applied at the select inputs, which can be factored out of any Boolean function of n variables.
  - The remaining $n - 2$ variables, referred to as the *residue variables*, can be formed into residue functions which can then be applied at the data inputs.
- Example: $f(A, B, C) = \sum(0, 1, 3, 4, 7)$.

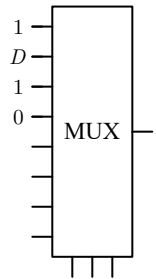




Multiplexer as a Boolean function generator

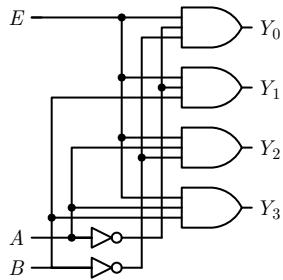
- Example: $f(A, B, C, D) = \sum(0, 1, 3, 4, 5, 9, 10, 11, 14, 15)$.

A	B	C	D	f
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
...				



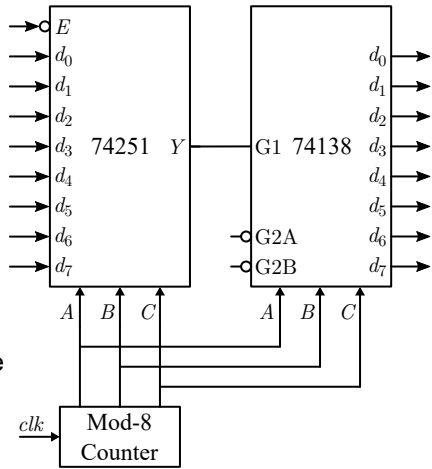
Demultiplexer

- As the name infers, a demultiplexer performs the opposite function to that of a multiplexer.
  - A single data line can be connected to any one of the output lines provided by the choice of an appropriate select signal.
  - If there are s select inputs then the number of output lines to which the data can be routed is $n = 2^s$.



Multiplexer/demultiplexer data transmission system

- A simple data transmission system can be implemented using a multiplexer and a demultiplexer in conjunction with an interconnecting single line link.
  - Such a system used over a relatively short distance such as 500 metres can result in a significant reduction in the number of lines required to transmit the data.
  - The data presented in parallel at the MUX inputs is converted into a serial format for transmission.
  - At the receiving end the demultiplexer routes the serial data, in the correct sequence, to one of the eight output lines.



Verilog: Control statements

- Similar to Java, Verilog has the following control statements:
  - `if-else`
  - `case`
  - `forever`
  - `repeat`
  - `while`
  - `for` loop

The conditional statement `if-else`

- The `if-else` statement controls the execution of other statements. In programming language like Java, `if-else` controls the flow of program.
  - When more than one statement needs to be executed for an `if` condition, then we need to use `begin` and `end` as seen in earlier examples.
- Syntax:

```
1 if (condition)
2   statements;
```

```
1 if (condition)
2   statements;
3 else
4   statements;
```

```
1 if (condition)
2   statements;
3 else if (condition)
4   statements;
5 ...
6 else
7   statements;
```

The conditional statement `if-else`

```
module if_else();
reg dff;
wire clk,din,reset;

always @ (posedge clk)
if (reset) begin
  dff <= 0;
end else begin
  dff <= din;
end
endmodule
```

The conditional statement case

```
module mux (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;
reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  default : $display("Error in SEL");
endcase
endmodule
```

The conditional statement case

```
module mux (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;
reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
  2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display("Error in SEL");
endcase
endmodule
```

The conditional statement case

- Verilog case statement does an identity comparison (like the == operator)
  - One can use the case statement to check for logic x and z values as shown in the example below.

```
1  module case_xz(enable);
2  input enable;
3
4  always @ (enable)
5  case(enable)
6    1'bz : $display ("enable is floating");
7    1'bx : $display ("enable is unknown");
8    default : $display ("enable is %b",enable);
9  endcase
10 endmodule
```

The conditional statement `casex` and `casez`

- They are special versions of the case statement allow the x and z logic values to be used as “don’t care”.
  - `casez`: Treats `z` as don’t care.
  - `casex`: Treats `x` and `z` as don’t care.

The conditional statement casex and casez

```
1  module case_compare;
2  reg sel;
3
4  always @ (sel)
5  case (sel)
6    1'b0 : $display("Normal : Logic 0 on sel");
7    1'b1 : $display("Normal : Logic 1 on sel");
8    1'bx : $display("Normal : Logic x on sel");
9    1'bz : $display("Normal : Logic z on sel");
10 endcase
```

The conditional statement casex and casez

```
always @ (sel)
casex (sel)
  1'b0 : $display("CASEX  : Logic 0 on sel");
  1'b1 : $display("CASEX  : Logic 1 on sel");
  1'bx : $display("CASEX  : Logic x on sel");
  1'bz : $display("CASEX  : Logic z on sel");
endcase

always @ (sel)
casez (sel)
  1'b0 : $display("CASEZ  : Logic 0 on sel");
  1'b1 : $display("CASEZ  : Logic 1 on sel");
  1'bx : $display("CASEZ  : Logic x on sel");
  1'bz : $display("CASEZ  : Logic z on sel");
endcase
```

The conditional statement casex and casez

```
26 initial begin
27   #1  $display ("\n  Driving 0");
28   sel = 0;
29   #1  $display ("\n  Driving 1");
30   sel = 1;
31   #1  $display ("\n  Driving x");
32   sel = 1'bx;
33   #1  $display ("\n  Driving z");
34   sel = 1'bz;
35   #1  $finish;
36 end
37 endmodule
```

The conditional statement casex and casez

```
  Driving 0
Normal : Logic 0 on sel
CASEX  : Logic 0 on sel
CASEZ  : Logic 0 on sel

  Driving 1
Normal : Logic 1 on sel
CASEX  : Logic 1 on sel
CASEZ  : Logic 1 on sel
```

```
  Driving x
Normal : Logic x on sel
CASEX  : Logic 0 on sel
CASEZ  : Logic x on sel

  Driving z
Normal : Logic z on sel
CASEX  : Logic 0 on sel
CASEZ  : Logic 0 on sel
```

The looping statement `forever`

- The `forever` loop executes continually, the loop never ends.
  - Normally we use forever statements in initial blocks.

```
module forever_example ();
reg clk;
initial begin
  #1  clk = 0;
  forever begin
    #5  clk =  ! clk;
  end
end
initial begin
  $monitor ("Time = %d  clk = %b",$time, clk);
  #100  $finish;
end
endmodule
```

The looping statement `repeat`

- The `repeat` loop executes statements a fixed number of times.

```
module repeat_example();
reg [3:0] opcode; reg [15:0] data; reg temp;
always @ (opcode or data) begin
  if (opcode == 10) begin
    // Perform rotate
    repeat (8) begin
      #1  temp = data[15];
      data = data << 1;
      data[0] = temp;
    end
  end
end
endmodule
```

The looping statement `while`

- The `while` loop executes as long as the statement evaluates as true.

```
module while_example();
reg [5:0] loc; reg [7:0] data;
always @ (data or loc) begin
  loc = 0;
  if (data == 0) loc = 32;
  else begin
    while (data[0] == 0) begin
      loc = loc + 1;
      data = data >> 1;
    end
  end
  $display ("DATA = %b   LOCATION = %d",data,loc);
end
endmodule
```

The looping statement `for`

- The `for` loop is the same as the for loop used in any other programming language.
  - Executes an *initial assignment* once at the start of the loop.
  - Executes the loop as long as an *expression* evaluates as true.
  - Executes a *step assignment* at the end of each pass through the loop.

```
module for_example();
integer i; reg [7:0] ram [0:255];
initial begin
  for (i = 0; i < 256; i = i + 1) begin
    ram[i] <= 0; // Initialize the RAM with 0
    #1  $display(" Address = %g  Data = %h",i,ram[i]);
  end
  #1  $finish;
end
endmodule
```