Combinational Logic II

CS207 Lecture 6

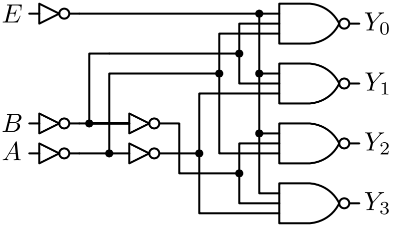
James YU

Mar. 25, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

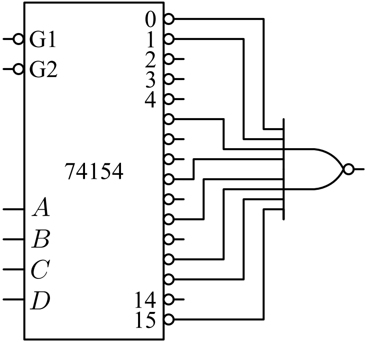
Decoder

- The basic function of an MSI decoder having n inputs is to select 1-out-of-2^n output lines.
  - The selected output is identified either by a `1`, when all other outputs are `0`, or by a `0` when all other outputs are `1`.



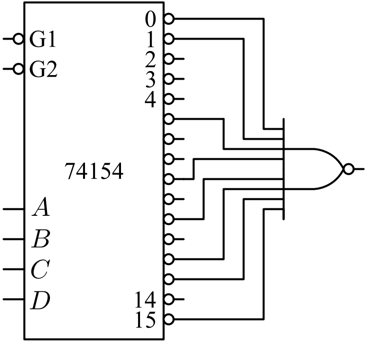
Decoder

- It will be seen that the logic diagram of the basic decoder is identical to that of the basic demultiplexer.
  - Provided the data line is used to enable the decoder.
- The decoder may also be regarded as a minterm generator. Each output generates one minterm.



Decoder

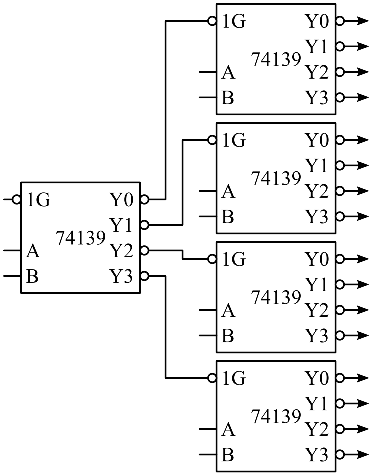
- If $A = B = C = D = 0$ the output `0` of the decoder is active low while all other outputs are `1`.
  - The decoder can generate the inverse of the 16 minterms.
  - Example: $f(A, B, C, D) = \sum(0, 1, 5, 8, 10, 12, 13, 15)$.

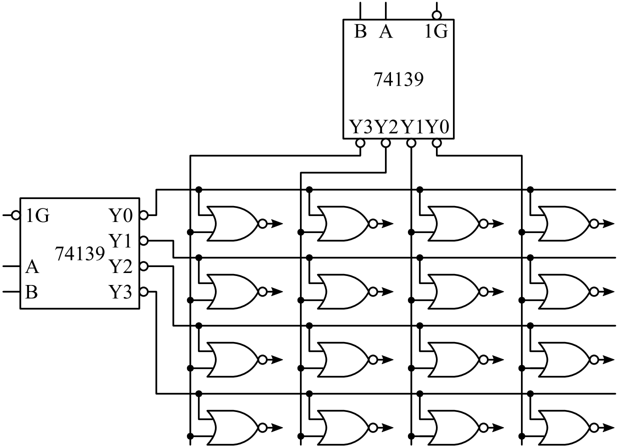


Decoder network

- When a large decoding network is required it cannot be implemented in a single MSI package.
  - Mainly because of the large number of pins needed.
- The decoding range can be extended by interconnecting decoder chips. Two schemes:
  - *Tree decoding*.
  - *Coincident decoding*.

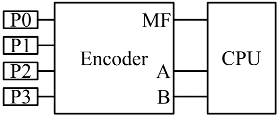
Decoder network

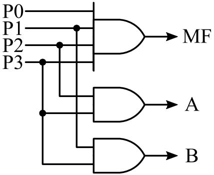




Encoder

- An encoder performs the inverse operation to that of a decoder.
- Example: CPU master flag





- The encoder is designed to identify one, and only one, of the peripherals at any given instant.
  - In practice, there is nothing to prevent two or more peripherals requesting service at the same time.
  - To deal with this situation a system of priorities can be attached to the peripheral flags.

Encoder

- Example: Octal-to-binary encoder.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

- Eight inputs and three outputs connected with OR.

Encoder

- $x = D_4 + D_5 + D_6 + D_7$.
- $y = D_2 + D_3 + D_6 + D_7$.
- $z = D_1 + D_3 + D_5 + D_7$.
- Only one input can be active for one time.
  - D_3 and D_6 are 1 simultaneously: outputs $(111)_2 = 7$.
  - Encoder circuit must have priority: *Priority encoder*.

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	x	x	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Encoder

$x = D_2 + D_3$

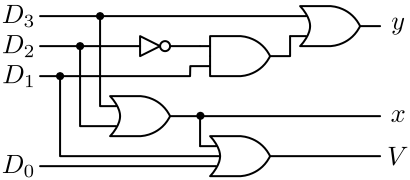
D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	X

$y = D_3 + D_1 D_2'$

D_0D_1 \ D_2D_3	00	01	11	10
00	X	1	1	0
01	1	1	1	0
11	1	1	1	0
10	0	1	1	0

Encoder

- $x = D_2 + D_3$.
- $y = D_3 + D_1 D_2'$.
- $V = D_0 + D_1 + D_2 + D_3$.

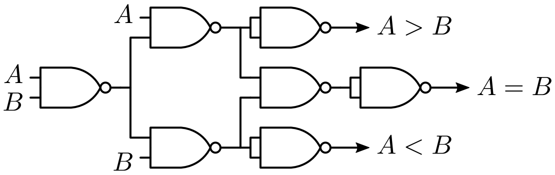


Digital comparator

- The usual problem for a comparator is the comparison of two multi-digit words such as $A = A_2A_1A_0$ and $B = B_2B_1B_0$.
  - Start from most to least significant bit.
  - $A = B$ if all bits are equal: $A_i = B_i$.
  - $x_i = A_iB_i + A_i'B_i'$.
- $A = B$ if $x_2x_1x_0 = 1$.
- $A > B$ if $A_2B_2' + x_2A_2B_2' + x_2x_1A_0B_0' = 1$.
- $A > B$ if $A_2'B_2 + x_2A_2'B_2 + x_2x_1A_0'B_0 = 1$.

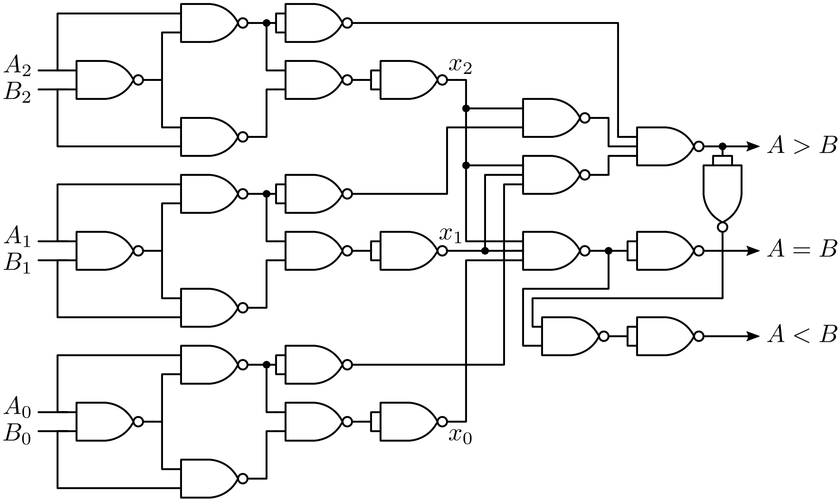
Digital comparator

- The implementation of a 3-bit comparator is based on a single bit comparator.



- Using the equations developed in the last page, we can have a 3-bit comparator.

Digital comparator



Verilog: Task

- Tasks are used in all programming languages, generally known as procedures or subroutines.
- The lines of code are enclosed in `task....end` task brackets.
- Data is passed to the task, the processing done, and the result returned.
- They have to be specifically called, with data `in`s and `out`s, rather than just wired in to the general netlist.
- Included in the main body of code, they can be called many times, reducing code repetition.
- Tasks are defined in the module in which they are used.
  - It is possible to define a task in a separate file and use the compile directive `'include` to include the task in the file which instantiates the task.

Verilog: Task

- A task begins with keyword `task` and ends with keyword `endtask`.
- Inputs and outputs are declared after the keyword `task`.
- Local variables are declared after input and output declaration.

```
module simple_task();
task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
  temp_out = (9/5) *(temp_in + 32)
end
endtask
endmodule
```

Verilog: Task

```verilog
module task_calling (temp_a, temp_b, temp_c, temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
reg [7:0] temp_b, temp_d;
`include "mytask.v"

always @ (temp_a)
begin
  convert (temp_a, temp_b);
end
always @ (temp_c)
begin
  convert (temp_c, temp_d);
end
endmodule
```

Verilog: Function

- A Verilog `function` is the same as a task, with very little differences, like `function` cannot drive more than one output, can not contain delays.
  - Functions can not include timing delays, like `posedge`, `negedge`, `#` delay, which means that functions should be executed in “zero” time delay.
  - Functions can have any number of inputs but only one output.
  - Functions can call other `function`s, but can not call `task`s.

```
module simple_function();
function myfunction;
input a, b, c, d;
begin
  myfunction = ((a+b) + (c-d));
end
endfunction
endmodule
```

Verilog: Function

```verilog
module function_calling(a, b, c, d, e, f);
input a, b, c, d, e;
output f;
wire f;
`include "myfunction.v"

assign f =  (myfunction (a,b,c,d)) ? e : 0;
endmodule
```

Verilog: Operators

- In Verilog, operators can be classified into multiple categories, i.e., arithmetic, relational, equality, logical, bit-wise, reduction, shift, concatenation, replication, and conditional operators.
- **Arithmetic operators**:
  - `+`, `-`, `*`, `/` — binary modulus operator.
  - `+`, `-` — unary operator used to specify the sign.
  - Integer division truncates any fractional part.
  - If any operand bit value is the unknown value `x`, then the entire result value is `x`.

Verilog: Operators

- **Relational operators**:
  - `a < b` — a less than b.
  - `a > b` — a greater than b.
  - `a <= b` — a less than or equal to b.
  - `a >= b` — a greater than or equal to b.
- The result is a scalar value (example `a < b`).
  - `0` if the relation is `false` (`a` is bigger then `b`).
  - `1` if the relation is `true` (`a` is smaller then `b`).
  - `x` if any of the operands has unknown `x` bits (if `a` or `b` contains `x`).

Verilog: Operators

- **Equality operators**:
  - `a === b` — `a` equal to `b`, including `x` and `z` (Case equality).
  - `a !== b` — `a` not equal to `b`, including `x` and `z` (Case inequality).
  - `a == b` — `a` equal to `b`, result may be unknown (logical equality).
  - `a != b` — `a` not equal to `b`, result may be unknown (logical equality).
- Operands are compared bit by bit, with zero filling if the two operands do not have the same length.
  - For the `==` and `!=` operators, the result is `x`, if either operand contains an `x` or a `z` (high impedance, or open circuit).
  - For the `===` and `!==` operators, bits with `x` and `z` are included in the comparison and must match for the result to be true.

Verilog: Operators

- **Logical operators**:
  - `!` — logic negation.
  - `&&` — logical and.
  - `||` — logical or.
- Expressions connected by `&&` and `||` are evaluated from left to right.
- Evaluation stops as soon as the result is known.
- The result is a scalar value:
  - `0` if the relation is false.
  - `1` if the relation is true.
  - `x` if any of the operands has `x` (unknown) bits.

Verilog: Operators

- **Bit-wise operators**:
  - ~ — negation.
  - & — and.
  - | — inclusive or.
  - ^ — exclusive or.
  - ^~ or ~^ — exclusive nor (equivalence).
- When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

Verilog: Operators

- *Concatenations* are expressed using the brace characters { and }, with commas separating the expressions within.
  - Example:

```
{a, b[3:0], c, 4'b1001} // if a and c are 8-bit numbers, the results has 24 bits.
```

  - Un-sized constant numbers are not allowed in concatenations.
- *Replication* operator is used to replicate a group of bits n times.
  - Say you have a 4 bit variable and you want to replicate it 4 times to get a 16 bit variable: then we can use the replication operator.

```
{3{a}} // this is equivalent to {a, a, a}
{b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}
```

Verilog: Operators

- The `conditional operator` has the following Java-like format: `cond_expr ? true_expr : false_expr`.
- The `true_expr` or the `false_expr` is evaluated and used as a result depending on what `cond_expr` evaluates to (`true` or `false`).