Two-level Implementation

CS207 Lecture 4

James YU

Mar. 11, 2020

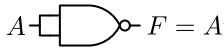
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

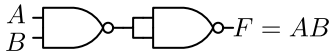
NAND and NOR implementation

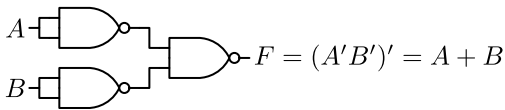
- Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates.
  - NAND and NOR gates are easier to fabricate with electronic components.
  - They are the basic gates used in all IC digital logic families.
- Rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND and NOR logic diagrams.

NAND circuits

- The NAND gate is said to be a universal gate.
- A convenient way to implement a Boolean function with NAND gates:
  - Obtain the simplified Boolean function in terms of Boolean operators;
  - Convert the function to NAND logic.
- Recall that:



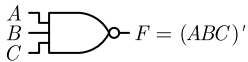




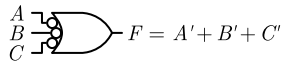
NAND circuits

- To facilitate the conversion to NAND logic, it is convenient to define an alternative graphic symbol for the gate.

AND-invert:

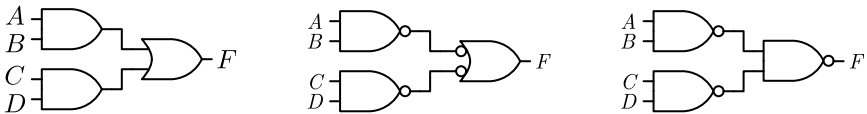


Invert-OR:



NAND circuits

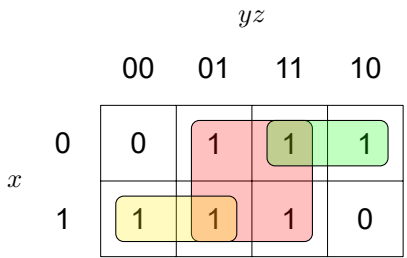
- The implementation of Boolean functions with NAND gates requires that the functions be in sum-of-products form.
- Take $F = AB + CD$ as an example:



- $F = AB + CD = ((AB)'(CD)')'$ according to DeMorgan property.

NAND circuits

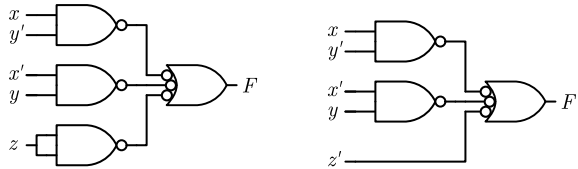
- Example: Implement the following Boolean function with NAND gates: $F(x, y, z) = \sum(1, 2, 3, 4, 5, 7)$.



- $F = xy' + x'y + z$.

NAND circuits

- Example: Implement the following Boolean function with NAND gates: $F = xy' + x'y + z$.

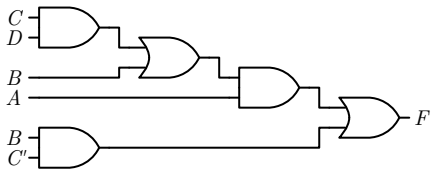


NAND circuits

- A Boolean function can be implemented with two levels of NAND gates.
  1. Simplify the function and express it in **sum-of-products form**.
  2. Draw **a NAND gate for each product term** of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.
  3. Draw **a single gate** using the AND-invert or the invert-OR graphic symbol **in the second level**, with inputs coming from outputs of first-level gates.
  4. A term with **a single literal requires an inverter** in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second level NAND gate.

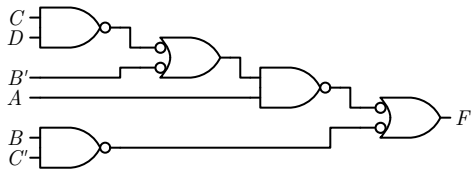
Multilevel NAND circuits

- The standard form of expressing Boolean functions results in a two-level implementation.
  - There are occasions when the design of digital systems results in gating structures with three or more levels.
  - Example: $F = A(CD + B) + BC'$.



Multilevel NAND circuits

- The standard form of expressing Boolean functions results in a two-level implementation.
  - There are occasions when the design of digital systems results in gating structures with three or more levels.
  - Example: $F = A(CD + B) + BC'$.

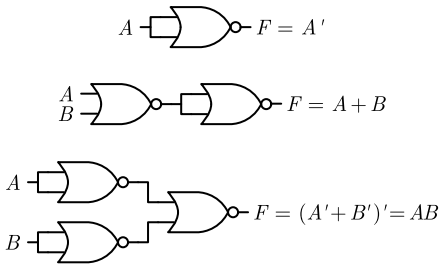


Multilevel NAND circuits

- The general procedure for converting a multilevel AND–OR diagram into an all-NAND diagram using mixed notation is as follows:
  1. Convert all AND gates to NAND gates with AND-invert graphic symbols.
  2. Convert all OR gates to NAND gates with invert-OR graphic symbols.
  3. Check all the bubbles in the diagram. For every bubble that is not compensated by another small circle along the same line, insert an inverter (a one-input NAND gate) or complement the input literal.

NOR circuits

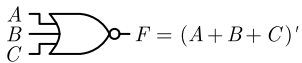
- The NOR operation is the dual of the NAND operation.
- All procedures and rules for NOR logic are the duals of the corresponding procedures and rules developed for NAND logic.



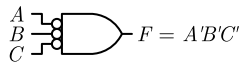
NOR circuits

- To facilitate the conversion to NOR logic, it is convenient to define an alternative graphic symbol for the gate.

OR-invert:



Invert-AND:

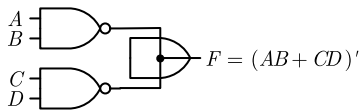


NOR circuits

- A two-level implementation with NOR gates requires that the function be simplified into product-of-sums form.
- Change the OR gates to NOR gates with OR-invert graphic symbols and the AND gate to a NOR gate with an invert-AND graphic symbol.

Other two-level implementations

- The types of gates most often found in integrated circuits are NAND and NOR gates.
- Some (but not all) NAND or NOR gates allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function.
  - Called *wired logic*.
- Example: open-collector TTL NAND gates, when tied together, perform wired-AND logic



Non-degenerate forms

- It will be instructive from a theoretical point of view to find out how many two-level combinations of gates are possible.
- We consider four types of gates: AND, OR, NAND, and NOR.
  - There are 16 possible combinations of two-level forms.
- Eight of these combinations are said to be *degenerate* forms.
  - They degenerate to a single operation.
  - Example: AND in the first level and second level degenerates to an AND of all inputs.
- The remaining eight nondegenerate forms produce an implementation in sum-of-products form or product-of-sums form.
  1) AND-OR 2) OR-AND 3) NAND-NAND 4) NOR-NOR
  5) NOR-OR 6) NAND-AND 7) OR-NAND 8) AND-NOR

AND-OR-INVERT implementation

- The two forms, NAND–AND and AND–NOR, are equivalent.
  - Both perform the AND–OR–INVERT function.
  - Example: $F = (AB + CD + E)'$.
- An AND–OR implementation requires an expression in sum-of-products form.
- The AND–OR–INVERT implementation is similar, except for the inversion.
  - If the complement of the function is simplified into sum-of-products form (by combining the 1's in the map), it will be possible to implement F with the AND–OR part of the function.

OR-AND-INVERT implementation

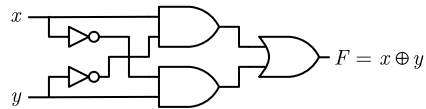
- The two forms, OR-NAND and NOR-OR, are equivalent.
  - Both perform the OR-AND–INVERT function.
  - Example: $F = [(A + B)(C + D)E]'$.
- The AND–OR–INVERT implementation requires an expression in product-of-sums form.

Exclusive-OR function

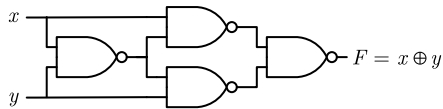
- Exclusive-OR, *XOR*: $x \oplus y = xy' + x'y$.
- Exclusive-NOR, *XNOR* or *equivalency*: $(x \oplus y)' = xy + x'y'$.
- The following identities apply to the XOR operation:
  - $x \oplus 0 = x$.
  - $x \oplus 1 = x'$.
  - $x \oplus x = 0$.
  - $x \oplus x' = 1$.
  - $x \oplus y' = x' \oplus y = (x \oplus y)'$.

XOR

- XOR is hard to fabricate, so it is typically constructed by other gates.
  - $(x' + y')x + (x' + y')y = xy' + xy' = x \oplus y.$



- Or use NAND gates:



  - The first NAND gate performs the operation $(xy)' = x' + y'$.
  - The other two-level NAND circuit produces the sum of products.

XOR

- Only a limited number of Boolean functions can be expressed in terms of XOR operations.
- Particularly useful in arithmetic operations and error detection/correction circuits.

Odd function

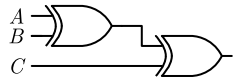
- The XOR operation with three or more variables can be converted into an ordinary Boolean function by replacing the $\oplus$ with its equivalent Boolean expression.

$$\begin{aligned} A \oplus B \oplus C &= (AB' + A'B)C' + (AB + A'B')C \\ &= AB'C' + A'BC' + ABC + A'B'C \\ &= \sum(1, 2, 4, 7) \end{aligned}$$

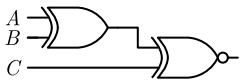
- The three-variable XOR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1.
  - An **odd** number of variables are equal to 1.
  - *Odd function*.

Odd function

- The three-input odd function is implemented by means of two-input XOR gates.



- *Even function* can also be implemented:



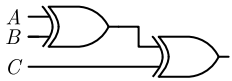
Parity generation and checking

- XOR functions are very useful in systems requiring error detection and correction codes.
  - A parity bit is an extra bit included with a binary message to make the number of 1's either odd or even.
- The circuit that generates the parity bit in the transmitter is called a *parity generator*.
- The circuit that checks the parity in the receiver is called a *parity checker*.

Parity generation and checking

- Consider a three-bit message to be transmitted together with an even-parity bit.

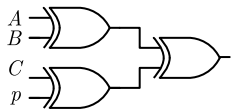
x	y	z	Parity bit p
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



- p constitutes an odd function.

Parity generation and checking

- The three bits in the message, together with the parity bit, are transmitted to their destination.
- The four bits received must have an even number of 1's with even parity.



Verilog behavioral modeling

- Behavioral models in Verilog contain procedural statements.
  - Control the simulation and manipulate variables of the data types.
  - Contained within procedures.
- There are two types of procedural blocks in Verilog:
  - `initial`: initial blocks execute only once at time zero (start execution at time zero).
  - `always`: always blocks loop to execute over and over again; in other words, as the name suggests, it executes always.

`initial` block

```
1  module initial_example();
2  reg clk,reset,enable,data;
3
4  initial begin
5   clk = 0;
6   reset = 0;
7   enable = 0;
8   data = 0;
9  end
10 endmodule
```

- In the above example, the initial block execution and always block execution starts at time 0.

`always` block

```
1  module always_example();
2  reg clk,reset,enable,q_in,data;
3
4  always @ (posedge clk)
5  if (reset)  begin
6     data <= 0;
7  end else if (enable) begin
8     data <= q_in;
9  end
10 endmodule
```

- In an always block, when the trigger event occurs, the code inside begin and end is executed.

Procedural assignment groups

- If a procedure block contains more than one statement, those statements must be enclosed within:
  - sequential `begin-end` block, or
  - parallel `fork-join` block

Procedural assignment groups

```
1  module initial_begin_end();
2  reg clk,reset,enable,data;
3
4  initial begin
5    $monitor("%3t clk=%b reset
         =%b enable=%b data=%b",
         $time, clk, reset,
         enable, data);
6    #1   clk = 0;
7    #10  reset = 0;
8    #5   enable = 0;
9    #3   data = 0;
10   #1  $finish;
11 end
12 endmodule
```

```
  0 clk=x reset=x enable=x data=x
  1 clk=0 reset=x enable=x data=x
 11 clk=0 reset=0 enable=x data=x
 16 clk=0 reset=0 enable=0 data=x
 19 clk=0 reset=0 enable=0 data=0
```

- Cause the statements to be evaluated sequentially (one at a time)
- Any timing within the sequential groups is relative to the previous statement.
- Block finishes after the last statement in the block.

Procedural assignment groups

```
1  module initial_fork_join();
2  reg clk,reset,enable,data;
3
4  initial begin
5    $monitor(...);
6    fork
7      #1   clk = 0;
8      #10  reset = 0;
9      #5   enable = 0;
10     #3   data = 0;
11   join
12   #1 $display ("%3t Stop",
       $time);
13   $finish;
14 end
15 endmodule
```

```
 0 clk=x reset=x enable=x data=x
 1 clk=0 reset=x enable=x data=x
 3 clk=0 reset=x enable=x data=0
 5 clk=0 reset=x enable=0 data=0
10 clk=0 reset=0 enable=0 data=0
11 Stop
```

- Cause the statements to be evaluated in parallel (all at the same time).
- Timing within parallel group is absolute to the beginning of the group.
- Block finishes after the last statement completes.

Blocking and nonblocking assignment

- Blocking assignments are executed in the order they are coded.
  - Assignment are made with `=` symbol.
  - Example: `a = b;`
- Nonblocking assignments are executed in parallel.
  - Assignments are made with `<=` symbol.
  - Example: `a <= b;`
- Correct way to spell *nonblocking* is “nonblocking” and not “non-blocking”.

Blocking and nonblocking assignment

```
1  module blocking_nonblocking();
2  reg a,b,c,d;
3
4  initial begin
5    #10  a = 0;
6    #11  a = 1;
7    #12  a = 0;
8    #13  a = 1;
9  end
10
11 initial begin
12   #10  b <= 0;
13   #11  b <= 1;
14   #12  b <= 0;
15   #13  b <= 1;
16 end
```

```
17 initial begin
18   c = #10 0;
19   c = #11 1;
20   c = #12 0;
21   c = #13 1;
22 end
23
24 initial begin
25   d <= #10 0;
26   d <= #11 1;
27   d <= #12 0;
28   d <= #13 1;
29 end
```

Blocking and nonblocking assignment

```
30 initial begin
31   $monitor("TIME = %2t A = %b B = %b C = %b D = %b",$time, a, b, c
       , d);
32    #50  $finish;
33 end
34 endmodule
```

```
TIME =  0 A = x B = x C = x D = x
TIME = 10 A = 0 B = 0 C = 0 D = 0
TIME = 11 A = 0 B = 0 C = 0 D = 1
TIME = 12 A = 0 B = 0 C = 0 D = 0
TIME = 13 A = 0 B = 0 C = 0 D = 1
TIME = 21 A = 1 B = 1 C = 1 D = 1
TIME = 33 A = 0 B = 0 C = 0 D = 1
TIME = 46 A = 1 B = 1 C = 1 D = 1
```

`assign`, `deassign`, `force`, and `release`

- The `assign` and `deassign` procedural assignment statements allow continuous assignments to be **placed onto registers** for controlled periods of time.
  - The `assign` procedural statement overrides procedural assignments to a register.
  - The `deassign` procedural statement ends a continuous assignment to a register.
- Another form of procedural continuous assignment is provided by the `force` and `release` procedural statements.
- These statements have a similar effect on the `assign-deassign` pair, but a `force` can be **applied to nets as well as to registers**.