Lecture 7, Jan. 24, 2007

Exclusive - OR function

Exclusive-OR (XOR) is a function measuring the inequality of its inputs, i.e., its output is 1 only when its inputs are different

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

So,

$$x \oplus y = x'y + xy'$$

X-NOR is defined as $(x \oplus y)'$ and we have

$$(x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = x'y' + xy$$

X-NOR is an indicator of equality of its inputs.

It is easy to see that

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

$$x \oplus y' = x' \oplus y = (x \oplus y)'$$

X-OR is both commutative and associative, i.e.,

$$x \oplus y = y \oplus x$$

and

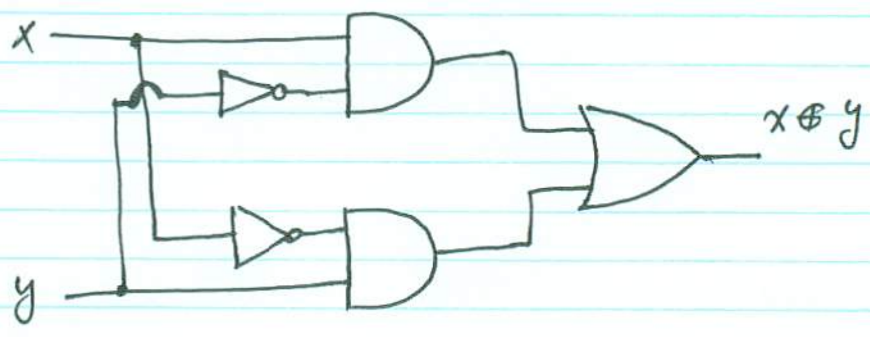
$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

This assures multi-input XOR's.

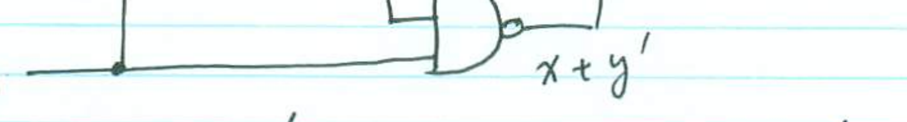
From the equality:

$$x \oplus y = x y' + x' y$$

we have the implementation



XOR can also be implemented using 4 NAND gates



$$\left\{ [x(xy)']' [(xy)'y]' \right\}' = [(x' + xy)(xy + y')]'$$
$$= [x'y + xy]' = (x + y)(x' + y')$$
$$= xy' + yx' = x \oplus y$$

Odd functions

Consider an XOR with three inputs x, y, z

$$
\begin{aligned}
x \oplus y \oplus z &= (x \oplus y)' z + (x \oplus y) z' \\
&= (xy' + x'y)' z + (xy' + x'y) z' \\
&= (x'y' + xy) z + (xy' + x'y) z' \\
&= x'y'z + xyz + xy'z' + x'yz' \\
&= \Sigma(1, 2, 4, 7)
\end{aligned}
$$

The K-map for $x \oplus y \oplus z$ is:

x \ yz	00	01	11	10
0		1		1
1	1		1	

Note that $x \oplus y \oplus z$ is equal to 1 for 001, 010, 100 and 111. These are the 3-bit patterns with 1 or 3 ones in them. That is XOR is true when there are an odd number of 1's among its inputs. So XOR is an odd function. X-NOR, on the other hand, is an even function.

x \ yz	00	01	11	10
0	1		1	
1		1		1

X-NOR

You can easily see that XOR of four variables w, x, y and z is:

$$w \oplus x \oplus y \oplus z = \sum(1,2,4,7,8,11,13,14)$$

wx \ yz	00	01	11	10
00		1		1
01	1		1	
11		1		1
10	1		1	

So, $w \oplus x \oplus y \oplus z$ is also an odd function that has value 1 when the number of 1's at its input is odd.

The above is true for XOR with any number of inputs.

Use of XOR in error detection/Correction

Assume that we are writing a byte (8-bits) in memory, e.g., on a hard disk, a ROM or a CD. When reading back the same byte, if one of the bits get Corrupted, say a zero becoming a 1 or vice versa, then the number of 1's become

one more or one less. If we had kept the record of the number of ones being odd or even, we could detect that something has gone wrong.
For example, assume that the byte was
11010011 if we add one extra bit that represented the XOR of these bits we will have
$1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1$
This shows that the number of ones in the byte is odd. So, we store 110100111 where the last bit is a parity bit. When we read back the bits, we XOR the first 8 bits and if we get the parity bit, we are ok, else, we detect an error.
Note that by adding a parity bit, we have made the parity of the 9 bit word even.
So, we need to XOR all 9 bits and if we get zero, we know all is fine else, there is an error.

An example of error correcting

By adding more parity bits, we can correct the errors. Take four bits w, x, y and z. If we add one parity bit for w, x, y another for x, y, z an a third for w, y, z, then we can correct any single bit error in w, x, y, z or the parities.

w	x	y	z	$P_1 = w \oplus x \oplus y$	$P_2 = x \oplus y \oplus z$	$P_3 = w \oplus y \oplus z$
0	0	0	0	0	0	0
0	0	0	1	0	1	1
0	0	1	0	1	1	1
0	0	1	1	1	0	0
0	1	0	0	1	1	0
0	1	0	1	1	0	1
0	1	1	0	0	0	1
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	0	1	1	1	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	0	1	1
1	1	0	1	0	0	0
1	1	1	0	1	0	0
1	1	1	1	1	1	1

This is called a (7, 4) Hamming Code. Here, 4 bits are coded into 7 bits (4 information bits and 3 parity bits).

Now assume that we start with 1001
we form 3 parities 1 1 0. So, we
store 1001110. Now assume that a bit
is changed due to the memory defect. Say,
when we read back the bits we get 1101110.
We form the parities and compare with the
parities we have 1101 gives us 0 0 0
So, P_1 and P_2 do not match while P_3 matches.
P_1 not matching tells us an error exists in
w, x or y
P_2 not matching lells us there is an error
in x, y, or z
P_3 matching tells us there is no error in
y, z and w.
These three clues tell us there is an error in
x. So, flipping x from 1 to 0 we get
1001
which is the original bit stream.

Hardware Description Language (HDL)

Hardware description languages such as Verilog and VHDL are used to design digital circuits in a manner similar to way you write computer programs.

Developing a digital circuit using HDL consists of the following steps:

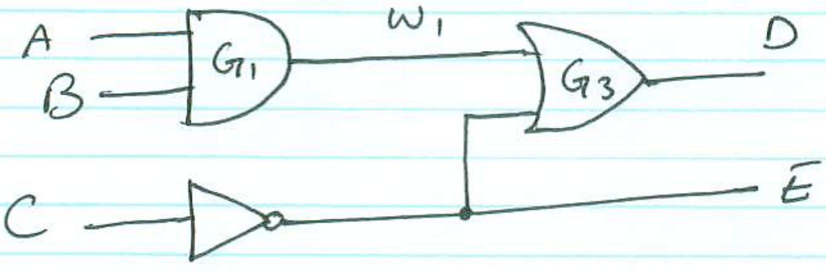
- Design entry: Creating an HDL based definition of the functionality of the circuit

- Logic Simulation: verifying the functionality of the logic using a computer.

- Logic Synthesis: Deriving the list of physical components need to implement the circuit and their interconnection (a net list).

- Timing Verification: To make sure that the fabricated circuit operate at the specified speed. In this stage, we need to take into consideration the gate delays.

Takes as an example, the HDL design of



First we need to <u>declare</u> the <u>module</u> using the language's <u>primitives</u>, e.g.,

<u>module</u>, <u>input</u>, <u>output</u>

<u>wire</u>, <u>and</u>, <u>or</u>, <u>not</u>.

Like C language // signifies the start of a comment.

The module declaration for the above ckt is:

```
// Description of the 3 gate circuit
module  my_circuit (A, B, C, D, E);
output  D, E;
input   A, B, C;
wire    w1;
and     G1 (w1, A, B);
not     G2 (E, C);
or      G3 (D, w1, E);
endmodule
```

Each module description starts with the keyword module followed by the name of the module (the name you choose for your circuit) followed by the list of ports in parantheses.
Then it is specified which ports are input and which are output. Then the different components of the module are defined as instances of previously defined blocks. At this point, we use the elementary gates that are part of the language. Later, you may instantiate other blocks from a library. These blocks may be the ones you have declared yourself or defined by someone else.

Gate delays

In order to simulate a circuit we may need the gate delays. The gate delays are defined in terms of time units and specified by #.
A time unit can be assigned a physical time value using 'timescale compiler directive.

For example:

'timescale 1ns/100ps

fixes a time unit as 1 ns with the precision of 100 ps = 0.1 ns.

Assume that the propagation delays for G_1, G_2 and G_3 are 30 ns, 10 ns and 20 ns, respectively. Then the module description with time delays included will be:

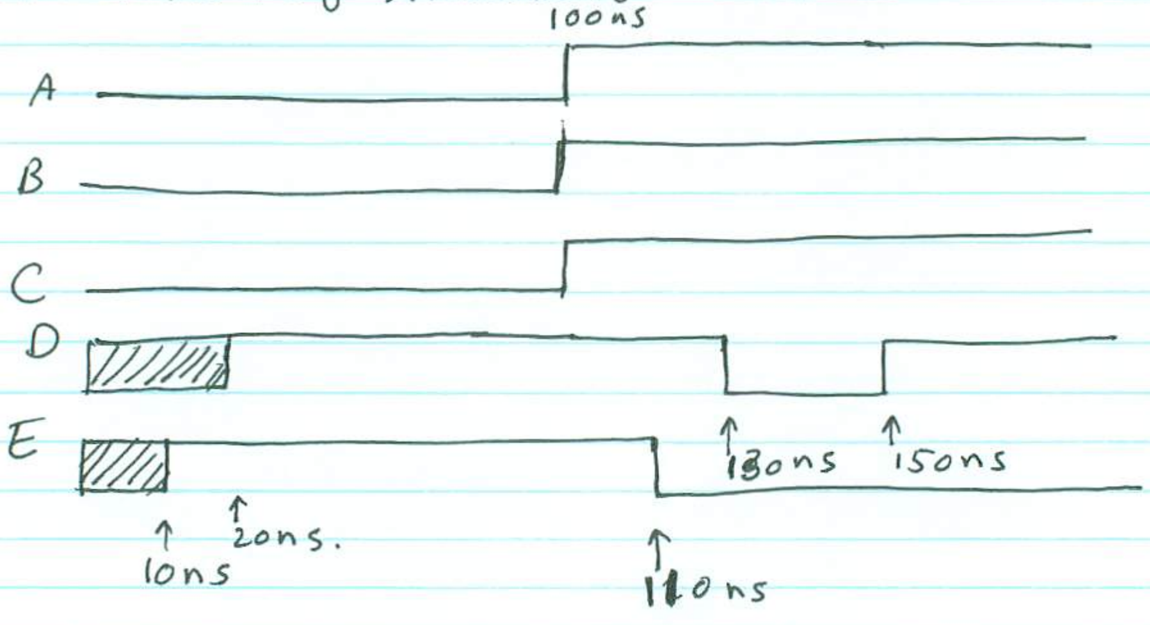
```
// Verilog mode with propagation delay
module my-circuit-prop_delay(A,B,C,D,E)
output  D,E;
input   A,B,C;
wire    w1;
and     #(30) G1(w1,A,B)
not     #(10) G2(E,C)
or      #(20) G3(D,w1,E)
endmodule
```

Simulation

In order to simulate our design, we need to write a test bench. A test bench is an HDL program that shows how to apply stimula to the circuit. It starts with the keyword module followed by the module name. The module name in the test bench has a prefix t_. The inputs are introduced with the keyword reg and the outputs with the keyword wire. The circuit to be simulated is instantiated with the instance name M1.
Then we use the keyword initial to give the sequence of simulation steps. Here, we ask that A be set to 0 at the beginning A=1'b0 means A is a 1 bit variable with value 0. Similarly B and C are asked to start at 0. The it is asked that after 100ns to change ABC to 111 and finish the simulation after 200 ns.

```
// Test bench for my-circuit
module t_my_circuit_prop_delay;
wire   D, E;
reg    A, B, C;
my_circuit_prop_delay M1 (A, B, C, D, E);
initial
begin
  A = 1'b0; B = 1'b0; C = 1'b0;
 #100 A = 1'b1; B = 1'b1; C = 1'b1;
end
initial #200 $finish;
endmodule
```

The result of simulation will be



Note that D is not defined for the first 20ns. and E for the first 10ns.

Boolean expressions in HDL

Boolean expressions can be defined using

& for AND

| for OR

~ for NOT

any Boolean function definition is done using the keyword assign. For example, the output of the circuit discussed above can be declared using the statement:

```
assign D = (A&B)|~C
```

Example: Write an HDL program implementing

$$E = A + BC + B'D$$

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

```
// Verilog model: My Boolean Circuit
module  my_Boolean_Circuit(E, F, A, B, C, D);
output  E, F;
input   A, B, C, D;
assign  E = A|(B&C)|(~B&D);
assign  F = (~B&C)|(B&~C&~D)
endmodule
```

User-Defined Primitives (UDP)

Some basic logic operations, e.g., AND, OR, NOT are included in the language. We can add other primitives (UDP) using a table. These user-defined primitives can be later instantiated when designing more complex circuits.

Example: Introduce a UDP implementing

$$X(A,B,C) = \Sigma(0,2,4,6,7)$$

```
// User Defined Primitive (UDP) example
primitive crctp(x,A,B,C);
output    x;
input     A,B,C;
// Truth Table for x(A,B,C)= Σ(0,2,4,6,7)
table
// A   B   C   :  x     (Note, this is a comment)
   0   0   0   :  1;
   0   0   1   :  0;
   0   1   0   :  1;
   0   1   1   :  0;
   1   0   0   :  1;
   1   0   1   :  0;
   1   1   0   :  1;
   1   1   1   :  1;
  endtable
endprimitive
```

The newly introduced primitive can then be instantiated using the following lines of code:

```
// Instantiate primitive
module  declare_crctp;
reg     x, y, z;
wire    w;
crctp (w, x, y, z);
endmodule
```