Digital System Design with HDL (I)

Lecture 9

Dr. Ming Xu

Dept of Electrical & Electronic Engineering

XJTLU

In This Session

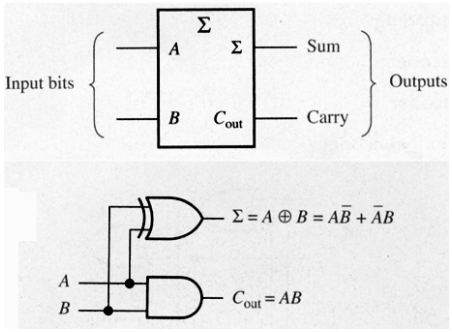
- Verilog for Combinational Circuits
  - Adders
  - Arithmetic Overflow

Half-Adders

The half-adder does not add an input carry.

A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

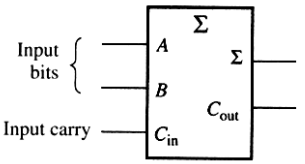
Σ = sum
C_{out} = output carry
A and B = input variables (operands)



- The sum is 1 only when the inputs are different — an XOR operation.
- The output carry is 1 only when both the inputs are 1 — an AND operation.

Full Adders

A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$$\Sigma = (A \oplus B) \oplus C_{in}$$

$$\begin{aligned} C_{out} &= ABC_{in} + AB\bar{C}_{in} + A\bar{B}C_{in} + \bar{A}BC_{in} \\ &= \left(ABC_{in} + AB\bar{C}_{in}\right) + \left(ABC_{in} + A\bar{B}C_{in}\right) + \left(ABC_{in} + \bar{A}BC_{in}\right) \\ &= AB + AC_{in} + BC_{in} \end{aligned}$$

Full Adders

Exclusive-OR operation satisfies associative rule:

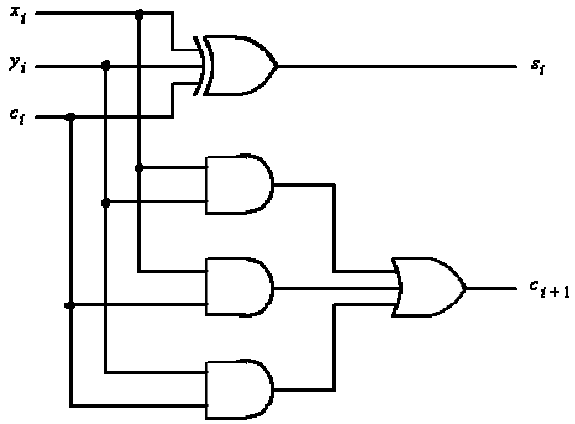
$$\begin{aligned} x \oplus (y \oplus z) &= x \oplus (\overline{y}z + y\overline{z}) \\ &= \overline{x}(\overline{y}z + y\overline{z}) + x(\overline{y} \cdot \overline{z} + yz) \\ &= \overline{x} \cdot \overline{y}z + \overline{x}y\overline{z} + x\overline{y} \cdot \overline{z} + xyz \end{aligned}$$

$$\begin{aligned} (x \oplus y) \oplus z &= (\overline{x}y + x\overline{y}) \oplus z \\ &= (\overline{x} \cdot \overline{y} + xy)z + (\overline{x}y + x\overline{y})\overline{z} \\ &= \overline{x} \cdot \overline{y}z + xyz + \overline{x}y\overline{z} + x\overline{y} \cdot \overline{z} \end{aligned}$$

We have

$$s = (x \oplus y) \oplus c = x \oplus y \oplus c$$

Full Adders



Full Adders

What is the output for an XOR gate with multiple inputs?

- If there are even 1s in the inputs, output a 0.
- If there are odd 1s in the input, output a 1.

Example:
0 ⊕ 0 ⊕ 0 = 0
1 ⊕ 0 ⊕ 0 = 1
1 ⊕ 1 ⊕ 0 = 0
1 ⊕ 1 ⊕ 1 = 1

Arithmetic Overflow

- Arithmetic **overflow** occurs when the outcome of addition does not fit within the magnitude bits used to represent the numbers.

(+7)	0 1 1 1	(–7)	1 0 0 1
+ (+2)	+ 0 0 1 0	+ (+2)	+ 0 0 1 0
(+9)	1 0 0 1	(–5)	1 0 1 1
	c_4 = 0		c_4 = 0
	c_3 = 1		c_3 = 0
(+7)	0 1 1 1	(–7)	1 0 0 1
+ (–2)	+ 1 1 1 0	+ (–2)	+ 1 1 1 0
(+5)	1 0 1 0 1	(–9)	1 0 1 1 1
	c_4 = 1		c_4 = 1
	c_3 = 1		c_3 = 0

Arithmetic Overflow

- It is important to be able to detect the overflow.
- In this example, $\text{Overflow} = c_3 \oplus c_4$
- For n-bit numbers we have

$$\text{Overflow} = c_n \oplus c_{n-1} = c_n \oplus x_{n-1} \oplus y_{n-1} \oplus s_{n-1}$$

Since $s_k = x_k \oplus y_k \oplus c_k$, it follows that

$$\begin{aligned} x_k \oplus y_k \oplus s_k &= (x_k \oplus y_k) \oplus (x_k \oplus y_k \oplus c_k) \\ &= (x_k \oplus y_k) \oplus (x_k \oplus y_k) \oplus c_k \\ &= 0 \oplus c_k \\ &= c_k \end{aligned}$$

Full-Adders

Using continuous assignment

```
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output s, Cout;

   assign s = x ^ y ^ Cin;
   assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```


Full-Adders

A 4-bit adder using module instantiation

```
module adder4 (carryin, X, Y, S, carryout);
    input carryin;
    input [3:0] X, Y;
    output [3:0] S;
    output carryout;
    wire [3:1] C;

    fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
    fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
    fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
    fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```


The **generate** Construct

- n instances of the *fulladd* subcircuit are required to build a n-bit ripple-carry adder.
- The **generate** construct (Verilog 2001) allows subcircuits to be instantiated in a loop.
- The loop index must be declared of type **genvar** – a positive integer that appears only in **generate** blocks.

Syntax:

generate

[procedural statements]

[instantiation statements]

endgenerate

The **generate** Construct

- Each instance has an instance name addbit[i].stage.

```
module ripple_g (carryin, X, Y, S, carryout);
  parameter n = 4;
  input carryin;
  input [n-1:0] X, Y;
  output [n-1:0] S;
  output carryout;
  wire [n:0] C;

  genvar i;
  assign C[0] = carryin;
  assign carryout = C[n];

  generate
    for (i = 0; i <= n-1; i = i+1)
    begin:addbit
      fulladd stage (C[i], X[i], Y[i], S[i], C[i+1]);
    end
  endgenerate

endmodule
```


Full-Adders

An n-bit adder using procedural statements

```
module addern (carryin, X, Y, S, carryout);
   parameter n=32;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout;
   reg [n:0] C;
   integer k;

   always @(X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
      end
      carryout = C[n];
   end

endmodule
```


Full-Adders

An alternative n-bit adder with the overflow signal

```
module addern (carryin, X, Y, S, carryout, overflow);
      parameter n = 32;
      input carryin;
      input [n-1:0] X, Y;
      output reg [n-1:0] S;
      output reg carryout, overflow;

      always @(X, Y, carryin)
      begin
          {carryout, S} = X + Y + carryin;
          overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
      end

endmodule
```


Full-Adders

A one-digit BCD adder

```
   0111      7
+  0101   +  5
------    ----
   1100     12
+  0110
------
  10010


   1000      8
+  1001   +  9
------    ----
  10001     17
+  0110
------
  10111
```

```
module bcdadd(Cin, X, Y, S, Cout);
   input Cin;
   input [3:0] X,Y;
   output reg [3:0] S;
   output reg Cout;
   reg [4:0] Z;

   always@ (X, Y, Cin)
   begin
      Z = X + Y + Cin;
      if (Z < 10)
         {Cout, S} = Z;
      else
         {Cout, S} = Z + 6;
   end

endmodule
```