Modeling Combinational Circuits with Verilog

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:34534
Email: jimmy@ee.ncu.edu.tw

Combinational Circuit Design
- Outputs are functions of inputs
- Description styles
  - Gate-level
  - Data-flow
  - Behavior
- Some guidelines are given for synthesis
Sensitivity List

- The sensitivity list must include all inputs of the block
  - All variables in condition statements
  - All variables on the right hand side of procedural assignments
- If not all inputs are listed
  - The changes of inputs may not change outputs immediately
  - May cause functional mismatch in the synthesized circuits
    - The sensitivity list will be skipped during synthesis

<table>
<thead>
<tr>
<th>Incomplete sensitivity list</th>
<th>Complete sensitivity list</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>always@(s1) begin</code></td>
<td><code>always@(s1 or a or b) begin</code></td>
</tr>
<tr>
<td><code>if (!s1) q=a;</code></td>
<td><code>if (!s1) q=a;</code></td>
</tr>
<tr>
<td><code>else q=b;</code></td>
<td><code>else q=b;</code></td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>end</code></td>
</tr>
</tbody>
</table>

Non-Synthesizable Verilog Constructs

- `initial`
- Loops
  - `repeat`
  - `forever`
  - `while`
- Data types
  - `event`
  - `real`
  - `time`
- `UDPs`
- `fork` ... `join` blocks
- Procedural assignments
  - `assign` and `deassign`
  - `force` and `release`
  - `disable`
- Some operators
  - `/` and `%`
  - `===` and `!==`
Inconsistent Results

- The following situations may cause simulation to disagree with synthesis
  - Incomplete sensitivity list
    - Sensitivity list is ignored in synthesis routines
  - Code with delays
    - The specified delay values are also ignored
  - Non-local reference within a function
  - Order dependency of concurrent statements
  - Comparisons to X or Z
  - Those cases should be avoided at all

Non-Local Reference

- Not all inputs are declared in the input list (sensitivity list) of a function

```verilog
function byte_compare;
  input [15:0] vector1, vector2;
  input [7:0] length;

  begin
    if (byte_sel) // compare the upper byte
      else // compare the lower byte
  end
endmodule
```
Order Dependency

always @(posedge CLK)
begin: CONCURRENT_1
    Y1 = A;
    end

always @(posedge CLK)
begin: CONCURRENT_2
    if (Y1 == 1)
        Y2 = B;
    else
        Y2 = 0;
    end

always @(posedge CLK)
begin: ALL_IN_ONE
    if (Y1 == 1)
        Y2 = B;
    else
        Y2 = 0;
    Y1 <= A;
end

Who is executed first will have different results

Y1 is definitely to be changed after Y2 change

Comparisons to X or Z

always @(A)
begin
    if (A === 1'bX)
        B = 0;
    else
        B = 1;
end

Comparisons to a "don’t care" are treated as always being false in synthesis routines
Priority for If Statements

The last "if" has highest priority

```verbatim
always@(a or b or c or d or sel)
begin
    z=0;
    if (sel[0]) z=a;
    if(sel[1]) z=b;
    if(sel[2]) z=c;
    if (sel[3]) z=d;
end
```

Late Arriving Data Signal

```verbatim
always@(a or b_is_late or c or d or sel)
begin
    z=0;
    if (sel[0]) z1=a;
    if(sel[2]) z1=c;
    if (sel[3]) z1=d;
        z=b_is_late;
    else
        z=z1;
end
```

Put the last arriving signal to the last if condition
Avoid Latch Inference (1/3)

- When if or case statements are used without specifying outputs in all possible condition, a latch will be created.

```vhdl
always@(enable or data)
  if (enable)
    q = data;
else
  q = 0;
end
```

Latch infer

Specify all output value

Avoid Latch Inference (2/3)

```vhdl
always@(a or b or c)
case(a)
  2'b11 : e = b;
  2'b10 : e = ~c;
end case
```

No latch

Add default statement

```vhdl
always@(a or b or c)
case(a)
  2'b11 : e = b;
  2'b10 : e = ~c;
  default : e = 0;
end case
```

Give initial value

result in a latch

```vhdl
always@(a or b or c)
e = 0;
case(a)
  2'b11 : e = b;
  2'b10 : e = ~c;
end case
```
Avoid Latch Inference (3/3)

Not all possible cases are defined

```
always@(a or b or c)
case(a)
  2'b11 : e = b;
  2'b10 : e = ~c;
end case
```

result in a latch

```
always@(a or b or c)
case(a) // synopsys full_case
  2'b11 : e = b;
  2'b10 : e = ~c;
end case
```

- To avoid latch inference and the need for default logic, add case directive:
  ```
  //synopsys full_case
  // ambit synthesis case = full
  ```

If vs. Case (1/2)

```
if (A(0) == 0 and A(1) == 0) then
  B = C;
else if (A(0) == 1 and A(1) == 0) then
  B = D;
else if (A(0) == 0 and A(1) == 1) then
  B = E;
else
  B = F;
end if;
```
If vs. Case (2/2)

```vhdl
2'b00 : B = C;
2'b01 : B = D;
2'b10 : B = E;
2'b11 : B = F;
endcase;
```

Case Directive

- A non-parallel (overlapped) case statement will generate the logic for a priority encoder
  - The first case item has the highest priority
- If only one case item is executed at a time
  - Force to generate multiplexer logic instead
    - Use `// synopsys parallel_case`
    - Use `//ambit synthesis case = parallel`
  - The results are unexpected when more than one case items are executed together
Inferring Multiplexers

- Use directives to force inferring multiplexers

```verilog
always@(SEL or DIN) begin: mux_lab
    case(SEL)
        2'b00: DOUT = DIN[0];
        2'b01: DOUT = DIN[1];
        2'b10: DOUT = DIN[2];
        2'b11: DOUT = DIN[3];
    endcase
end
```

Loop Synthesis

- In synthesis, for loops are “unrolled”, and then synthesized

```verilog
integer i;
always@(a or b) begin
    for (i = 0; i <= 3; i = i + 1)
        out[i] = a[i] & b[3-i];
end
```

```verilog
integer i;
always@(a or b) begin
    out[0] = a[0] & b[3];
    out[3] = a[3] & b[0];
end
```
Non-Static Loops

- Non-static loops are not synthesizable

```verilog
module NonStaticLoop (A, B, R, Y);
    input [7:0] A, B;
    input [2:0] R;
    output [7:0] Y;
    reg [7:0] Y;
    integer i;
    always @(A) begin
        Y = 8’d0;
        for (i=0; i<R; i=i+1) Y[i] = A[i] & B[i];
    end
endmodule
```

R is non-static

Resource Sharing (1/2)

```verilog
always@ (a or b or c or d) out = (a) ? (b+c): (b+d);
```

```verilog
always@ (a or b or c or d) if (a) out = b+c;
else out = b+d;
```

without resource sharing with resource sharing

Keep sharable resource in the:

- same conditional statement
- same always block
- same module
Resource Sharing (2/2)

- The operators that can share resources must be in the mutual exclusive paths

```verilog
if (sel1) out1 = a1+b1;
else begin
  out1 = c1+d1;
  if (sel2) out2 = a2+b2;
  else out2 = c2+d2;
end
```

operator "c1+d1" and "a2+b2" or "c2+d2" are not in the mutual exclusive paths

```
if (sel1) out1 = a1+b1;
else begin
  out1 = a1+b1;
  if (sel2) out2 = a2+b2;
  else out2 = c2+d2;
end
```

All operators are in the mutual exclusive paths

Modeling Examples with Verilog for Combinational Circuits
Code Converter (1/3)

- A code converter transforms one representation of data to another
- Ex: A BCD to excess-3 code converter
  - BCD: Binary Coded Decimal
  - Excess-3 code: the decimal digit plus 3

<table>
<thead>
<tr>
<th>TABLE 4-2</th>
<th>Truth Table for Code Converter Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Output Excess-3</td>
</tr>
<tr>
<td>BCD</td>
<td>W X Y Z</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1 0 1 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 1 0 0</td>
</tr>
</tbody>
</table>

Code Converter (2/3)

- Equations: (share terms to minimize cost)
  
  \[ W = A + BC + BD = A + B(C+D) \]
  \[ X = \overline{BC} + \overline{BD} + \overline{B\overline{CD}} = \overline{B}(C+D) + B\overline{CD} \]
  \[ Y = CD + \overline{CD} = C \oplus D \]
  \[ Z = D \]

[Diagram of code converter]
Code Converter (3/3)

Data-flow style

assign W = A|(B&(C|D));
assign X = ~B&(C|D)|(B&~C&~D);
assign Y = ~(C^D);
assign Z = ~D;

Behavior style

assign ROM_in = {A, B, C, D};
assign {W, X, Y, Z} = ROM_out;
always @(ROM_in) begin
  case (ROM_in)
    4'b0000: ROM_out = 4'b0011;
    4'b0001: ROM_out = 4'b0100;
    4'b1001: ROM_out = 4'b1100;
    default: ROM_out = 4'b0000;
  endcase
end

Decoder (1/2)

A decoder is to generate the $2^n$ (or fewer) minterms of $n$ input variables

Ex: a 3-to-8 line decoder

<table>
<thead>
<tr>
<th>Inputs</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Outputs

<table>
<thead>
<tr>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>
**Decoder (2/2)**

- **Behavior style 1**

```verilog
input x, y, z;
reg [7:0] D;
always @(x or y or z) begin
    case ({x, y, z})
        3'b000: D = 8'b00000001;
        3'b001: D = 8'b00000010;
        .
        3'b111: D = 8'b10000000;
        default: D = 8'b0;
    endcase
end
```

- **Behavior style 2**

```verilog
input x, y, z;
wire [2:0] addr;
reg [7:0] D;
assign addr = {x, y, z};
always @(addr) begin
    D = 8'b0;
    D[addr] = 1;
end
```

D is unknown when addr is unknown

---

**Encoder (1/2)**

- An encoder performs the inverse operation of a decoder
  - Have \(2^n\) (or fewer) input lines and \(n\) output lines
  - The output lines generate the binary code of the input positions
  - Only one input can be active at any given time
  - Ex: a octal-to-binary encoder

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D_0)</td>
<td>(x)</td>
</tr>
<tr>
<td>(D_1)</td>
<td>(D_2)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
z = D_1 + D_2 + D_3 + D_7\]
\[
y = D_2 + D_3 + D_6 + D_7\]
\[
x = D_4 + D_5 + D_6 + D_7\]
Encoder (2/2)

Data-flow style

Behavior style
always @(D) begin
  case (D)
    8'b00000001: {x, y, z} = 3'b000;
    8'b00000010: {x, y, z} = 3'b001;
    .
    8'b10000000: {x, y, z} = 3'b111;
    default: {x, y, z} = 3'b000;
  endcase
end

Priority Encoder (1/2)

- An encoder circuit that includes the priority function
- If two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{Inputs} & D_0 & D_1 & D_2 & D_3 & x & y & z & V \\
\hline
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
X & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
X & X & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
X & X & X & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\( V = 0 \): no valid inputs

\( x = D_2 + D_3 \)
\( y = D_1 + D_1D_2' \)
\( V = D_0 + D_1 + D_2 + D_3 \)
Priority Encoder (2/2)

- **Data-flow style**
  
  ```
  assign x = D[2] | D[3];
  ```

- **Behavior style**
  
  ```
  always @(D) begin
    V = 1;
    case(D)
      4'b1000: {x, y} = 2'b11;
      4'b0100: {x, y} = 2'b10;
      4'b0010: {x, y} = 2'b01;
      4'b0001: {x, y} = 2'b00;
      default: begin
        {x, y} = 2'bx;
        V = 0;
      end
    endcase
  end
  ```

Multiplexer (1/2)

- A multiplexer uses \( n \) selection bits to choose binary info. from a maximum of \( 2^n \) unique input lines
- Like a decoder, it decodes all minterms internally
- Unlike a decoder, it has only one output line

![Decoder Diagram](image)
Multiplexer (2/2)

- Behavior style 1
  
  ```
  input [1:0] S;
  input [3:0] I;
  output Y;
  always @(S or I) begin
    case (S)
      2'b00: Y = I[0];
      2'b01: Y = I[1];
      2'b10: Y = I[2];
      2'b11: Y = I[3];
      default: Y = 0;
    endcase
  end
  ```

- Behavior style 2
  
  ```
  input [1:0] S;
  input [3:0] I;
  output Y;
  always @(S or I) begin
    Y = I[S];
  end
  ```

  Suitable for the cases that inputs are not coded (ex: A, B, C, ...)

Demultiplexer (1/2)

- It performs the inverse function of a multiplexer
- It receives info from a single line and transmits it to one of \(2^n\) possible output lines
- A decoder with enable input can function as a demultiplexer
  - Often referred to as a decoder/demultiplexer

![Demultiplexer Diagram](image_url)
**Demultiplexer (2/2)**

- **Behavior style 1**
  
  ```
  input A, B, E;
  reg [3:0] D;
  always @(A or B or E) begin
    D = 4’b1111;
    case ({A, B})
      2’b00: D[0] = E;
      2’b01: D[1] = E;
      default: D = 4’b0;
    endcase
  end
  ```

- **Behavior style 2**
  
  ```
  input A, B, E;
  wire [1:0] S;
  reg [3:0] D;
  assign S = {A, B};
  always @(S or E) begin
    D = 4’b1;
    D[S] = E;
  end
  ```

**Binary Adder Cell (1/2)**

<table>
<thead>
<tr>
<th>TABLE 3-7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Truth Table of Half Adder</strong></td>
</tr>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table 3-7 Truth Table of Half Adder

<table>
<thead>
<tr>
<th>TABLE 3-8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Truth Table of Full Adder</strong></td>
</tr>
<tr>
<td>Inputs</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Table 3-8 Truth Table of Full Adder

Fig. 3-25 Logic Diagram of Half Adder

Fig. 3-27 Logic Diagram of Full Adder
Binary Adder Cell (2/2)

- Half adder
  assign \( \{C, S\} = X + Y; \)
  assign \( C = X \& Y; \)
  assign \( S = X \oplus Y; \)

- Full adder
  assign \( \{C, S\} = X+Y+Z; \)
  assign \( C = (X\&Y) \| Z&(X\oplus Y); \)
  assign \( S = X \oplus Y \oplus Z; \)

preferred writing style
that can tell synthesizer
the existence of an adder

Ripple Carry Adder

module FA4 (S, C4, A, B, C0);
input [3:0] A, B;
input C0;
output [3:0] S;
output C4;

FA1 U0(S[0], C1, A[0], B[0], C0);
FA1 U1(S[1], C2, A[1], B[1], C1);
FA1 U2(S[2], C3, A[2], B[2], C2);
FA1 U3(S[3], C4, A[3], B[3], C3);
endmodule
Carry Propagation

- The **carry propagation time** is a limiting factor on the speed with which two numbers are added.
- All other arithmetic operations are implemented by successive additions.
  - The time consumed during the addition is very critical.
- To reduce the carry propagation delay:
  - Employ faster gates with reduced delays.
  - Increase the equipment complexity.
- Several techniques for reducing the carry propagation time in a parallel adder:
  - The most widely used technique employs the principle of **carry lookahead**.

Carry Lookahead Adder

- The critical path “carry” is calculated separately to reduce delay.
- \( G_n = A_nB_n \)
- \( P_n = A_n \oplus B_n \)
- \( S_n = P_n \oplus C_n \)
- \( C_n = G_{n-1} + P_{n-1}C_{n-1} \)
- \( C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \)
- Gate-level descriptions are often used.
Hierarchical CLA

- \( C_n = G_{n-1} + P_{n-1}C_{n-1} \)
- \( C_4 = (G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0) + (P_3P_2P_1P_0)C_0 \)
- \( P_{0-3} = P_3P_2P_1P_0 \)
- \( G_{0-3} = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 \)

Carry Skip Adder

- Looks for the cases in which the carry out is identical to carry in
  - All propagation
  - The most time-consuming path
- For such cases, **bypass** the original carry path
  - Generate carry out directly (Cout=Cin)
  - Use a MUX to select
Carry Select Adder

- Compute two results in parallel, each for different carry assumptions
- Use actual carry in to select correct results
- Reduce delay to a multiplexer

Comparison of Adders

- Make comparison between Time *(speed)* and Space *(area)*
- Empirical average power consumption of 16-bit adders

<table>
<thead>
<tr>
<th>Adder</th>
<th>Time</th>
<th>Space</th>
<th>Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ripple carry adder</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>0.117</td>
</tr>
<tr>
<td>Carry look-ahead adder</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
<td>0.171</td>
</tr>
<tr>
<td>Carry skip adder</td>
<td>$O(\sqrt{n})$</td>
<td>$O(n)$</td>
<td>0.109</td>
</tr>
<tr>
<td>Carry select adder</td>
<td>$O(\sqrt{n})$</td>
<td>$O(n)$</td>
<td>0.216</td>
</tr>
</tbody>
</table>
**Binary Adder-Subtractor (1/2)**

- Subtraction can be achieved by adding 2’s compliment
  - \( A - B = A + (-B) \)
  - Compliment each bit of \( B \) then add 1 to the result
- \( M = 0 \): addition
- \( M = 1 \): supply the additional “1” in subtraction

**Binary Adder-Subtractor (2/2)**

```verilog
module Add_Sub (S, C4, A, B, M);
    input [3:0] A, B;
    input M;  // 0: Add, 1: Sub
    output [3:0] S;
    output C4;
    wire [3:0] K;
    assign K = B ^ {4{M}};
    FA1 U0(S[0], C1, A[0], K[0], M);
    FA1 U1(S[1], C2, A[1], K[1], C1);
    FA1 U2(S[2], C3, A[2], K[2], C2);
    FA1 U3(S[3], C4, A[3], K[3], C3);
endmodule
```
Signed Adder-Subtractor

- Unsigned
  - C detects a carry after addition or a borrow after subtraction
- Signed
  - V bit detects an overflow
    - 0: no overflow
    - 1: overflow

![Diagram of Signed Adder-Subtractor](image)

BCD Adder (1/2)

C detects whether the binary sum is greater than 9 (1001)

\[ C = K + Z_2 Z_4 + Z_2 Z_2 \]

If \( C = 1 \), it is necessary to add 6 (0110) to binary sum

![Diagram of BCD Adder](image)
**BCD Adder (2/2)**

```verilog
module bcd_add (S, Cout, A, B, Cin);
    input [3:0] A, B;
    input Cin;
    output [3:0] S;
    output Cout;
    reg [3:0] S;
    reg Cout;
    always @(A or B or Cin) begin
        {Cout, S} = A + B + Cin;
        if (Cout != 0 || S > 9) begin
            S = S + 6;
            Cout = 1;
        end
    end
endmodule
```

---

**Binary Multiplier**

- **Ex: 2 x 2 multiplier**

  - Compute partial products, and justify the sum of those partial products
  - $m \times n$ digit multiplication generates up to an $(m+n)$ digit result
  - Can be described using $\times$ (not recommended)
4-Bit By 3-Bit Binary Multiplier

We need 12 AND gates and two 4-bit adders to produce a product of 7 bits.

General Multiplier Architecture

CLA can be used.
Column Compression Tree

- Also called *Wallace Tree*
- Use “carry in” to add three components at a time
- “carry out” is added in next column

Signed Multiplication

- Case: negative multiplicand, positive multiplier
  → sign extension works

- Other cases can be handled by applying 2’s complement to appropriate numbers