Combinational Logic (II)

Jia Chen
jiac@ucr.edu

Slides adopted from Hung-Wei Tseng

Outline

- Let’s start designing the first circuit
- Designing circuit with HDL

Let’s design a circuit!

The sum-of-product form of the full adder

- How many of the following minterms are part of the sum-of-product form of the full adder in generating the output bit?
  1. A'B'Cin'
  2. A'BCin' ✓
  3. AB'Cin' ✓
  4. ABCin'
  5. A'B'Cin ✓
  6. A'BCin
  7. AB'Cin
  8. ABCin ✓

A. 0
B. 1
C. 2
D. 3
E. 4

Input			Output	
A	B	Cin	Out	Cout
0	0	0	**0**	**0**
0	1	0	**1**	**0**
1	0	0	**1**	**0**
1	1	0	**0**	**1**
0	0	1	**1**	**0**
0	1	1	**0**	**1**
1	0	1	**0**	**1**
1	1	1	**1**	**1**

Out = A'BCin' + AB'Cin' + A'B'Cin + ABCin

Cout = ABCin' + A'BCin + AB'Cin + ABCin

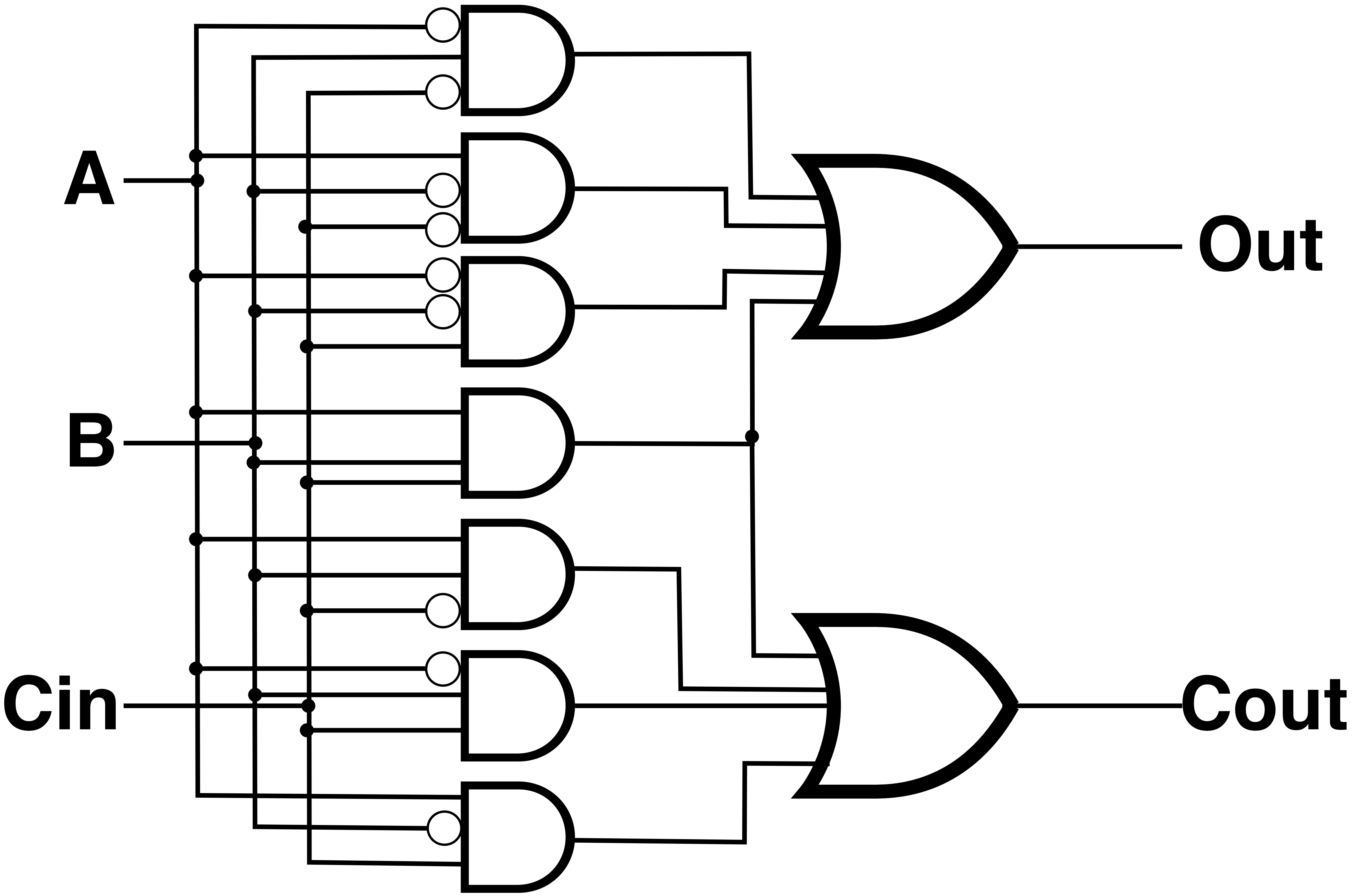
Out = A'BCin' + AB'Cin' + A'B'Cin + $\boxed{\text{ABCin}}$

Cout = ABCin' + A'BCin + AB'Cin + $\boxed{\text{ABCin}}$

The same

The full adder

Input			Output	
A	B	Cin	Out	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



Do we need to perform hardware design in gate-level?

— Not when you can use an HDL!

Turn a design into Verilog

Verilog

- Verilog is a Hardware Description Language (HDL)
  - Used to describe & model the operation of digital circuits.
  - Specify simulation procedure for the circuit and check its response — simulation requires a logic simulator.
  - Synthesis: transformation of the HDL description into a physical implementation (transistors, gates)
    - When a human does this, it is called logic design.
    - When a machine does this, it is called synthesis.
- In this class, we use Verilog to implement and verify your processor.
- C/Java like syntax

Data types in Verilog

- Bit vector is the only data type in Verilog
- A bit can be one of the following
  - 0: logic zero
  - 1: logic one
  - X: unknown logic value, don’t care
  - Z: high impedance, floating
- Bit vectors expressed in multiple ways
  - 4-bit binary: 4‘b11_10 (_ is just for readability)
  - 16-bit hex: 16‘h034f
  - 32-bit decimal: 32‘d270

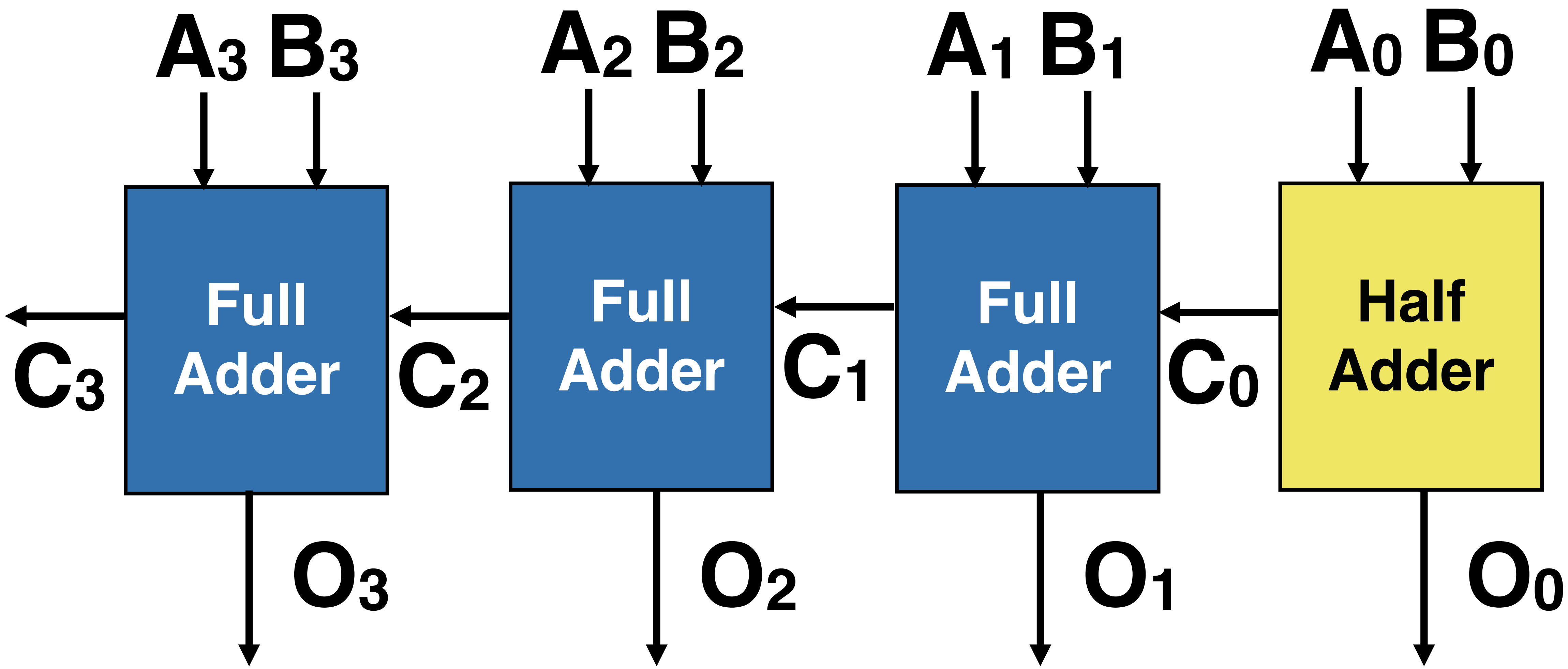
Operators

<table>
<tr><th colspan="2">Arithmetic</th><th colspan="2">Logical</th><th colspan="2">Bitwise</th><th colspan="2">Relational</th></tr>
<tr><td>+</td><td>addition</td><td>!</td><td>not</td><td>~</td><td>not</td><td>></td><td>greater than</td></tr>
<tr><td>-</td><td>substraction</td><td>&&</td><td>and</td><td>&</td><td>and</td><td><</td><td>less than</td></tr>
<tr><td>*</td><td>multiplication</td><td>||</td><td>or</td><td>|</td><td>or</td><td>>=</td><td>greater or equal</td></tr>
<tr><td>/</td><td>division</td><td></td><td></td><td>^</td><td>xor</td><td><=</td><td>less or equal</td></tr>
<tr><td>%</td><td>Don’t use modulus</td><td></td><td></td><td>~^</td><td>xnor</td><td>==</td><td>equal (doesn’t work if there is x, z)</td></tr>
<tr><td>**</td><td>power</td><td></td><td></td><td><<</td><td>shift left</td><td>!=</td><td>not equal</td></tr>
<tr><td></td><td></td><td></td><td></td><td>>></td><td>shift right</td><td>===</td><td>really equal</td></tr>
<tr><th colspan="2">Concatenation</th><td colspan="2">{} (e.g., {1b’1,1b’0} is 2b’10)</td><th colspan="2">Replication</th><td colspan="2">{{}} (e.g., {4{1b’0}} is 4b’0)</td></tr>
<tr><th colspan="2">Conditional</th><td colspan="6">condition ? value_if_true : value_if_false</td></tr>
</table>

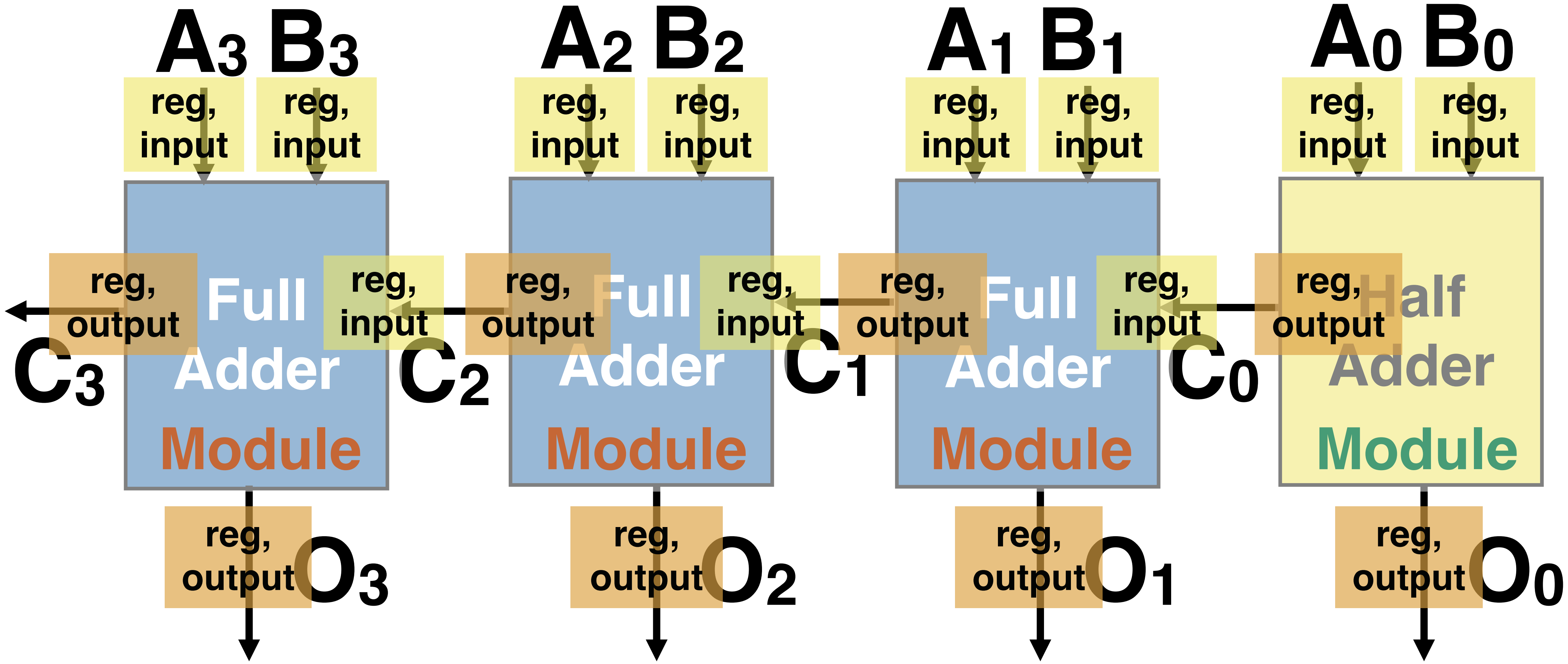
Wire and Reg

- wire is used to denote a hardware net — "continuously assigned" values and do not store
  - single wire
    wire my_wire;
  - array of wires
    wire[7:0] my_wire;
- reg is used for procedural assignments — values that store information until the next value assignment is made.
  - again, can either have a single reg or an array
    reg[7:0] result; // 8-bit reg
  - reg is not necessarily a hardware register
  - you may consider it as a variable in C

Revisit the 4-bit adder



Revisit the 4-bit adder



Half adder

Input		Output	
A	B	Out	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Out = A'B + AB'

Cout = AB

```
module HA( input a,
           input b,
           output cout,
           output out );
assign out = (~a & b)|(a & ~b);
assign cout = a&b;
endmodule
```

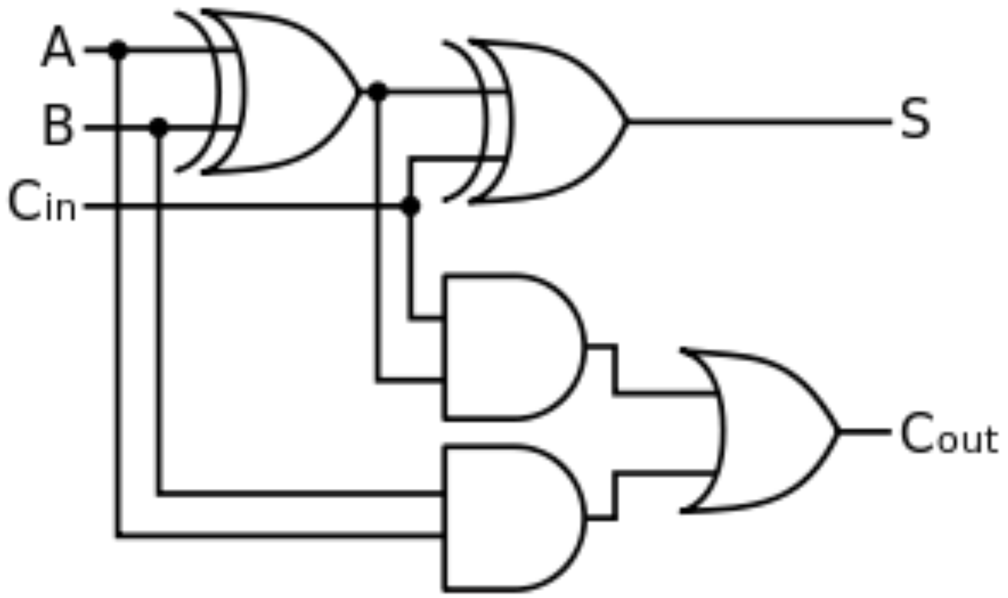
Input ports

Output ports

assign is used for **driving** wire/net type declarations. Since wires change values according to the value driving them, whenever the operands on the RHS changes, the value is **evaluated** and assigned to LHS(thereby simulating a wire).

Modules

- A Verilog module has a name and a port list
  - ports: must have a direction (input, output, inout) and a bitwidth
- Think about an 1-bit adder
  - input: 1-bit * 3
  - output 1-bit * 1 and 1-bit * 1




```
module FA( input a,
           input b,
           input cin,
         output cout,
         output sum );
assign sum = a^b^cin;
assign cout = (a&b) | (a&cin) | (b&cin);
endmodule
```

Adapted from Arvind & Asanovic's MIT 6.375 lecture

Identifier and keyword

An ***identifier*** is a designer-defined name used for items such as modules, inputs, and outputs. An identifier must start with a letter (A-Z, a-z) or underscore (_), followed by any number of letters (A-Z, a-z), digits (0-9), underscores (_), or dollar signs ($). Identifiers are case sensitive, meaning upper and lower case letters differ. So testEn and testEN are different.

A ***keyword*** is a word that is part of the language, like the words module and input. A designer cannot use a keyword as an identifier.

Full adder

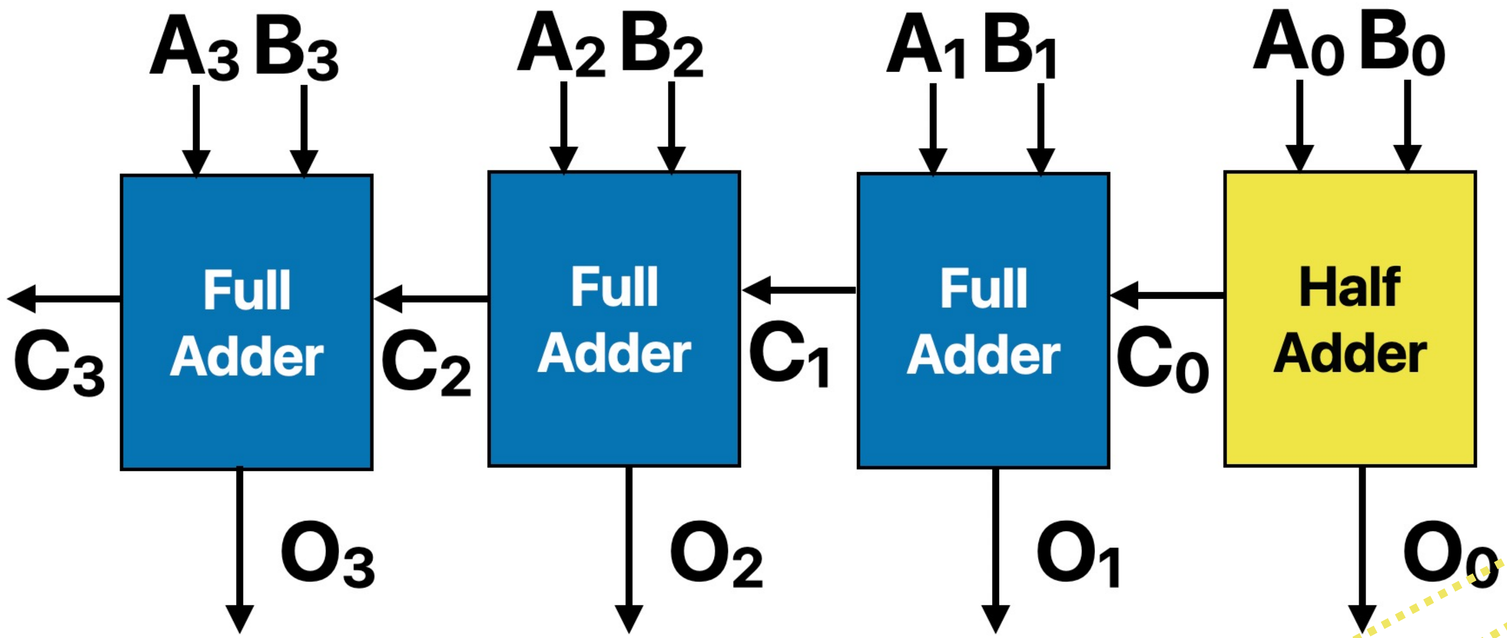
Out = A'BCin' + AB'Cin' + A'B'Cin + ABCin

Cout = ABCin' + A'BCin + AB'Cin + ABCin

Input			Output	
A	**B**	**Cin**	**Out**	**Cout**
0	0	0	**0**	**0**
0	1	0	**1**	**0**
1	0	0	**1**	**0**
1	1	0	**0**	**1**
0	0	1	**1**	**0**
0	1	1	**0**	**1**
1	0	1	**0**	**1**
1	1	1	**1**	**1**

```
module FA( input a,
           input b,
           input cin,
           output cout,
           output out );
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);
endmodule
```

The Adder




```
module FA( input a,
           input b,
           input cin,
           output cout,
           output out );
assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);
endmodule
```

```
module HA( input a,
           input b,
           output cout,
           output out );
assign out = (~a & b)|(a & ~b);
assign cout = a&b;
endmodule
```

```
module adder( input[3:0]  A,
              input[3:0]  B,
              output[3:0] O,
              output cout);
wire [2:0] carries;
HA ha0(.a(A[0]), .b(B[0]), .out(O[0]), .cout(carries[0]));
FA fa1(.a(A[1]), .b(B[1]), .cin(carries[0]), .out(O[1]), .cout(carries[1]));
FA fa2(.a(A[2]), .b(B[2]), .cin(carries[1]), .out(O[2]), .cout(carries[2]));
FA fa3(.a(A[3]), .b(B[3]), .cin(carries[2]), .out(O[3]), .cout(cout);
endmodule
```

Connecting ports by name yields clearer and less buggy code.

The Adder

- A verilog module can instantiate other moudles

```
module adder( input[3:0]  A,
              input[3:0]  B,
              output[3:0] O,
              output cout);
wire [2:0] carries;
HA ha0(.a(A[0]), .b(B[0]), .out(O[0]), .cout(carries[0])); // explicit binding
FA fa1(A[1], B[1], carries[0], O[1], carries[1])); // implicit binding
FA fa2(.a(A[2]), .b(B[2]), .cin(carries[1]), .cout(carries[2]), .out(O[2])); // explicit binding
FA fa3(.a(A[3]), .b(B[3]), .cin(carries[2]), .out(O[3]), .cout(cout); // explicit binding
endmodule
```

Always block — combinational logic

- Executes when the condition in the sensitivity list occurs

```
    module FA( input a,
               input b,
               input cin,
               output cout,
               output out );
always@(a or b or cin)
begin                          // the following block changes outputs when a, b or cin changes
    assign out = (~a&b&~cin)|(a&~b&~cin)|(~a&~b&cin)|(a&b&cin);
    assign cout = (a&b&~cin)|(~a&b&cin)|(a&~b&cin)|(a&b&cin);
end
    endmodule
```

Always block — sequential logic

- Executes when the condition in the sensitivity list occurs

```
always@(posedge clk)        // the following block only triggered by a positive clock
begin
...
...
end
```

Blocking and non-blocking

- Inside an always block, = is a blocking assignment
  - assignment happens immediately and affect the subsequent statements in the always block
- <= is a non-blocking assignment
  - All the assignments happens at the end of the block
- Assignment rules:
  - The left hand side, LHS, must be a reg.
  - The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Initially, a = 2, b = 3

```
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(posedge clock)
begin
a <= b;
c <= a;
end
```

Afterwards: a = 3 and c = 2

```
reg a[3:0];
reg b[3:0];
reg c[3:0];
always @(*)
begin
a = b;
c = a;
end
```

Afterwards: a = 3 and c = 3

“Always blocks” permit more advanced sequential idioms

```
module mux4( input a,b,c,d,
             input [1:0] sel,
             output out );
reg out;
always @( * )
begin
  if ( sel == 2’d0 )
    out = a;
  else if ( sel == 2’d1 )
    out = b
  else if ( sel == 2’d2 )
    out = c
  else if ( sel == 2’d3 )
    out = d
  else
    out = 1’bx;
  end
end
endmodule
```

```
module mux4( input a,b,c,d,
             input [1:0] sel,
             output out );
reg out;
always @( * )
  begin
    case ( sel )
      2’d0 : out = a;
      2’d1 : out = b;
      2’d2 : out = c;
      2’d3 : out = d;
      default : out = 1’bx;
    endcase
  end
endmodule
```

Initial block

- Executes only once in beginning of the code

```
initial
begin
...
...
end
```

Testing the adder!

```
`timescale 1ns/1ns // Add this to the top of your file to set time scale
module testbench();
reg [3:0] A, B;
reg C0;
wire [3:0] S;
wire C4;
adder uut (.B(B), .A(A), .sum(S), .cout(C4)); // instantiate adder

initial
begin
A = 4'd0; B = 4'd0; C0 = 1'b0;
#50 A = 4'd3; B = 4'd4;       // #50 in front of the statement delays its execution by 50 time units
#50 A = 4'b0001; B = 4'b0010;
end

endmodule
```

2 Min Poll

How many will get “1”s

- For the following Verilog code snippet, how many of their “output” values will be 1 after the “always” block finishes execution?

```
reg a[3:0];
reg b[3:0];
reg output[3:0];

initial
begin
a = 4b’1000;
b = 4b’1001;
end

always @(posedge clock)
begin
a <= a^b;
output <= a;
end
```

```
reg a[1:0];
reg b[1:0];
reg output[3:0];

initial
begin
a = 2b’00;
b = 2b’10;
end

always @(posedge clock)
begin
b <= a;
output <= {a,~b};
end
```

```
reg a[3:0];
reg b[3:0];
reg output[3:0];

initial
begin
a = 4b’10x1;
b = 4b’1001;
end

always @(*)
begin
assign output = (a == b) ? 4b’0001: 4b’0000;
end
```

A. 0
B. 1
C. 2
D. 3

How many will get "1"s

- For the following Verilog code snippet, how many of their "output" values will be 1 after the "always" block finishes execution?

```
reg a[3:0];
reg b[3:0];
reg output[3:0];

initial
begin
a = 4b'1000;
b = 4b'1001;
end

always @(posedge clock)
begin
a <= a^b;          // a=4b'0001
output <= a;
end      // output=4b'1000
```

✔

```
reg a[1:0];
reg b[1:0];
reg output[3:0];

initial
begin
a = 2b'00;
b = 2b'10;
end

always @(posedge clock)
begin
b <= a;          // b=2b'00
output <= {a,~b};
end          // output=4b'{00,01}
```

```
reg a[3:0];
reg b[3:0];
reg output[3:0];

initial
begin
a = 4b'10x1;
b = 4b'1001;
end

always @(*)                a==b —> x
begin
assign output = (a == b) ? 4b'0001: 4b'0000;
end                        //output = 4b'0000
```

A. 0

B. 1

C. 2

D. 3

Parameterize your module

```
module adder #(parameter WIDTH=32)(
        input[WIDTH-1:0]  A,
        input[WIDTH-1:0]  B,
        output[WIDTH-1:0] O,
        output cout);

endmodule
```

Coding guides

- When modeling sequential logic, use nonblocking assignments.
- When modeling latches, use nonblocking assignments.
- When modeling combinational logic with an always block, use blocking assignments.
- When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.
- Do not mix blocking and nonblocking assignments in the same always block.
- Do not make assignments to the same variable from more than one always block.
- Use $strobe to display values that have been assigned using nonblocking assignments.
- Do not make assignments using #0 delays.

http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA.pdf