Verilog

Jia Chen
jiac@ucr.edu

Slides are adapted from Hung-Wei Tseng

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.

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 (*underscore character _ is just for readability*)
  - 16-bit hex: 16'h034f
  - 32-bit decimal: 32'd270

Operators

Arithmetic		Logical		Bitwise		Relational	
+	addition	!	not	~	not	>	greater than
-	substraction	&&	and	&	and	<	less than
*	multiplication	ll	or	l	or	>=	greater or equal
/	division			^	xor	<=	less or equal
%	modulus			~^	xnor	==	equal (doesn't work if there is x, z)
**	power			<<	shift left	!=	not equal
				>>	shift right	===	really equal
Concatenation		{} (e.g., {1'b1,1'b0} is 2'b10)		**Replication**		{{}} (e.g., {4{1'b0}} is 4'b0)	
Conditional		condition ? value_if_true : value_if_false					

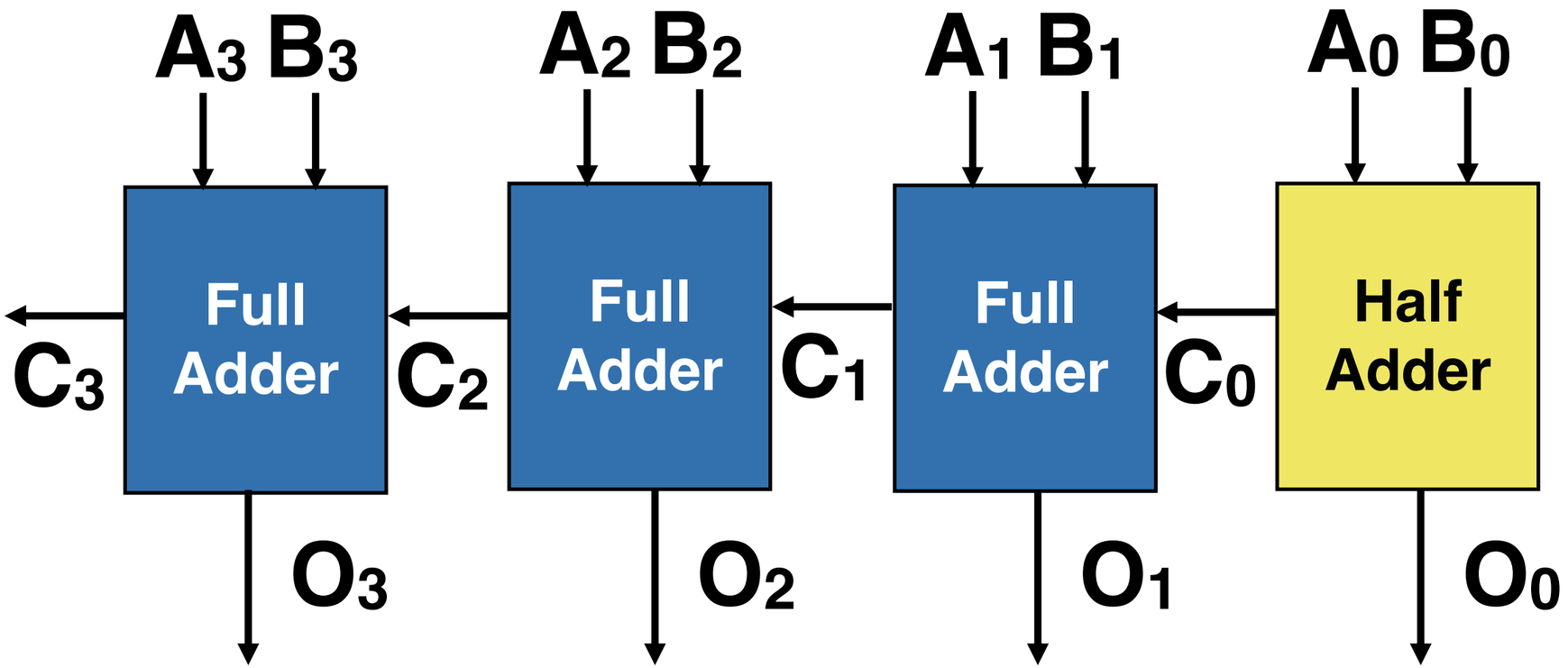
Don't use

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 result; // 1-bit reg

    reg[7:0] result; // 8-bit reg
  - reg is not necessarily a hardware register

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

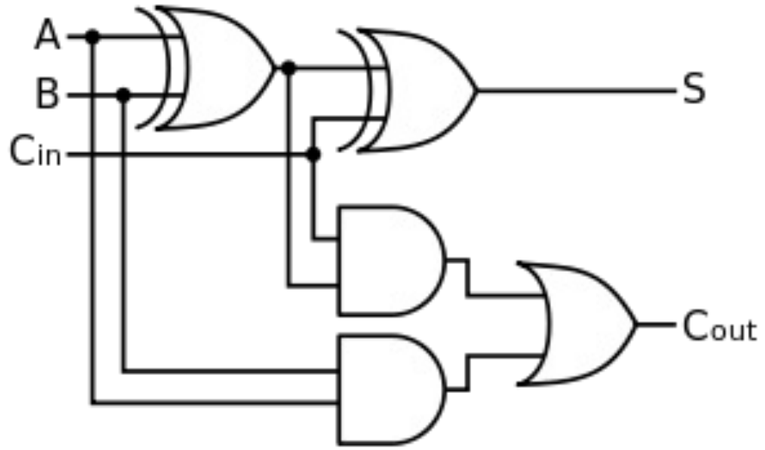
Output ports

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
```



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.
  - ✓ Ex: testEn and testEN are different.

- A ***keyword*** is a word that is part of the language
  - ✓ Eg. Module, input, output

  - ✓ A designer cannot use a keyword as an identifier.

Full adder

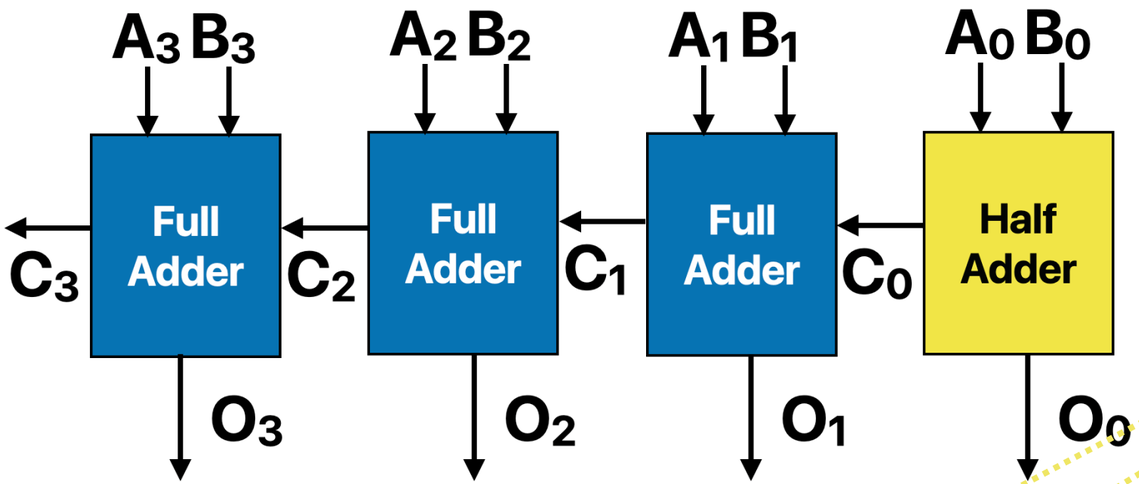
Out = A'BCin'+ AB'Cin'+ A'B'Cin+ ABCin

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

Instantiation： ha0, fa1, fa2, fa3

The Adder

- A verilog module can instantiate other modules

```
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
```

Q: can we remove the *always* block?

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 (time unit base / precision base)
module testbench();
reg [3:0] A, B;
reg C0;
wire [3:0] S;
wire C4;
adder uut (.B(B), .A(A), .O(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
```

How many will get “1”s

- For the following Verilog code snippet, how many of their “output” decimal 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 = 4’b1000;
  b = 4’b1001;
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 = 2’b00;
   b = 2’b10;
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 = 4’b10x1;
  b = 4’b1001;
end

always @(*)
begin
        output =  (a == b) ?   4’b0001: 4’b0000;
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” decimal 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
```

^ xor

```
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’0001
```

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

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

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

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

✓<= is a non-blocking assignment

✓All the assignments happens at the end of the block

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
```

Verilog tutorials and examples

NAND.LAND

https://www.nandland.com/verilog/tutorials/index.html