Verilog

Prof. Usagi

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

Arithmetic		Logical		Bitwise		Relational	
+	addition	!	not	~	not	>	greater than
-	substraction	&&	and	&	and	<	less than
*	multiplication	||	or	|	or	>=	greater or equal
/	division			^	xor	<=	less or equal
%	modulus **Don't use**			~^	xnor	==	equal (doesn't work if there is x, z)
**	power			<<	shift left	!=	not equal
				>>	shift right	===	really equal

Concatenation	{} (e.g., {1b'1,1b'0} is 2b'10)	Replication	{{}} (e.g., {4{1b'0}} is 4b'0)
Conditional	condition ? value_if_true : value_if_false		

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
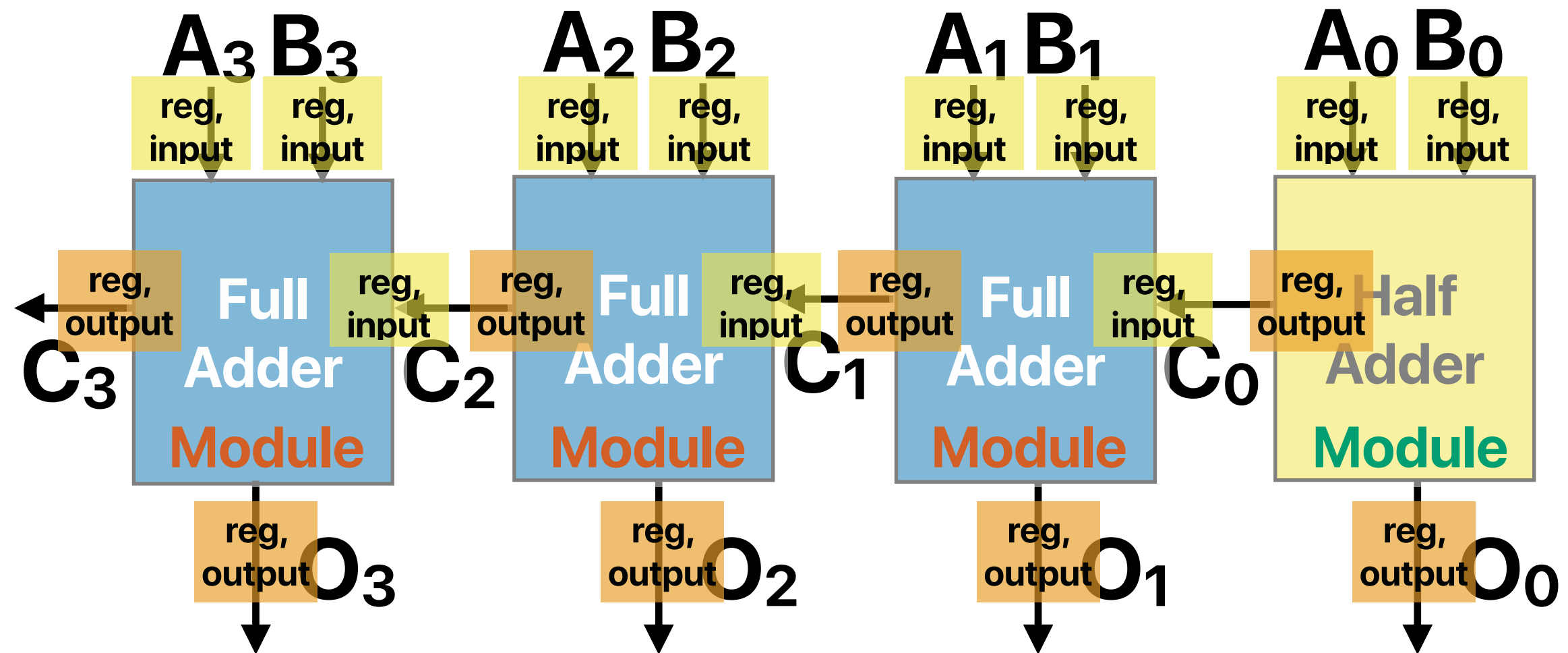
Half adder

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
```
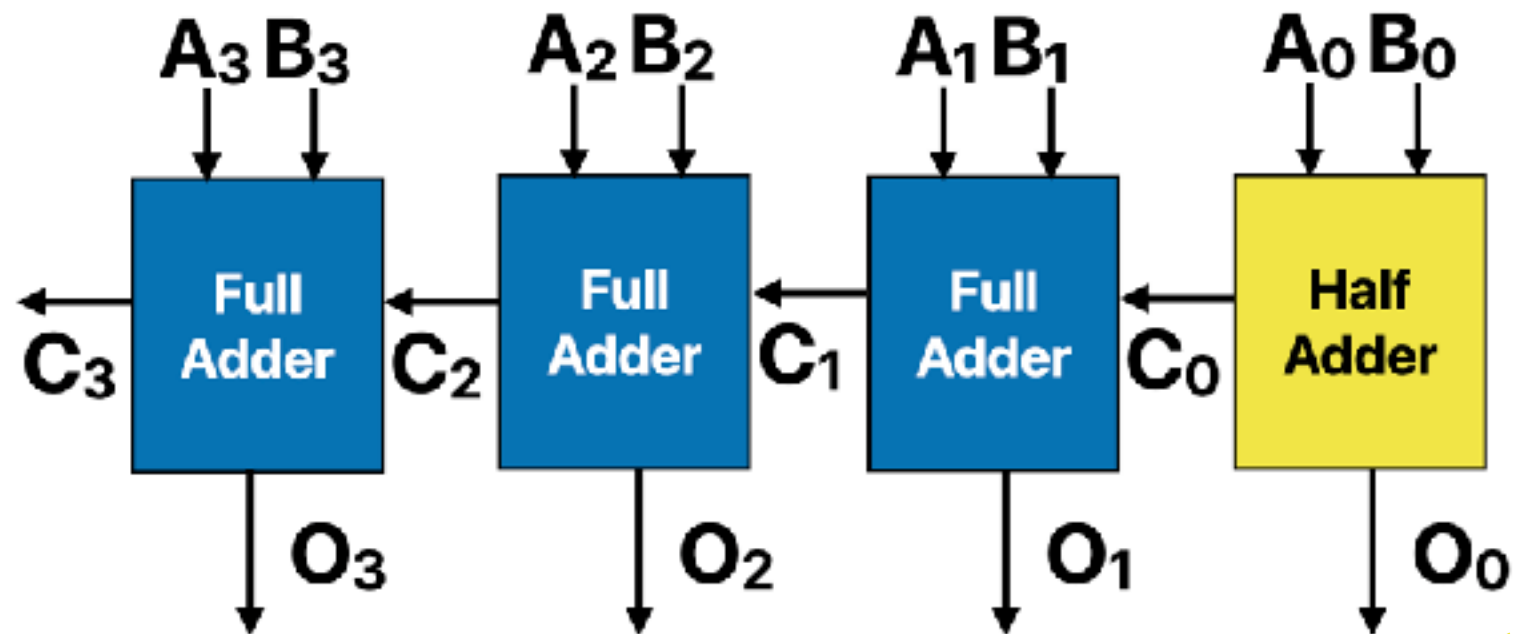
Input ports

Output ports

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**

```verilog
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[2]), .cout(cout);
endmodule
```

Connecting ports by name yields clearer and less buggy code.

10

Always block — combinational logic

- Executes when the condition in the sensitivity list occurs

```verilog
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

```verilog
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
endmodule
```

```verilog
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!

```verilog
`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;       // wait 50 ns before next assignment
#50 A = 4'b0001; B = 4'b0010; // don't use #n outside of testbenches
end

endmodule
```

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