

EECS151/251A

Introduction to Digital Design and ICs

Lecture 4 –Verilog II

Intel Is Now Ordering Chips From TSMC Following Delays

Intel Corp. ([INTC](#)) is ordering its 6-nanometer (nm) chips to be manufactured by TSMC ([TSM](#)) for next year, according to a report from the Taiwan-based newspaper Commercial Times.

<https://www.yahoo.com/news/intel-now-ordering-chips-tsmc-072255657.html>

https://en.wikichip.org/wiki/technology_node

Sophia Shao



Number of Semiconductor Manufacturers with a Cutting Edge Logic Fab									
SilTerra									
X-FAB									
Dongbu HiTek									
ADI	ADI								
Atmel	Atmel								
Rohm	Rohm								
Sanyo	Sanyo								
Mitsubishi	Mitsubishi								
ON	ON								
Hitachi	Hitachi								
Cypress	Cypress								
Sony	Sony								
Infineon	Infineon								
Sharp	Sharp								
Freescale	Freescale								
Renesas (NEC)	Renesas	Renesas							
Toshiba	Toshiba	Toshiba							
Fujitsu	Fujitsu	Fujitsu							
TI	TI	TI							
Panasonic	Panasonic	Panasonic	Panasonic	Panasonic	Panasonic				
STMicroelectronics	STM	STM	STM	STM	STM				
HLMC	HLMC	HLMC	HLMC	HLMC	HLMC				
UMC	UMC	UMC	UMC	UMC	UMC		UMC		
IBM	IBM	IBM	IBM	IBM	IBM		IBM		
SMIC	SMIC	SMIC	SMIC	SMIC	SMIC		SMIC		
AMD	AMD	AMD	GlobalFoundries	GF	GF		GF		
Samsung	Samsung	Samsung	Samsung	Samsung	Samsung		Samsung		
TSMC	TSMC	TSMC	TSMC	TSMC	TSMC		TSMC		
Intel	Intel	Intel	Intel	Intel	Intel		Intel		
180 nm	130 nm	90 nm	65 nm	45 nm/40 nm	32 nm/28 nm	22 nm/20 nm	16 nm/14 nm	10 nm	7 nm
									5 nm

Review

- Design metrics:
 - Functionality and robustness:
 - Cost
 - Performance
 - Power and Energy
- Verilog
 - Hardware description language: describe hardware!
 - Logic synthesis: Verilog -> Gate-level netlists
 - Used in both ASIC and Verilog

Verilog Introduction

- A module definition describes a component in a circuit
- Two ways to describe module contents:
 - **Structural Verilog**
 - List of sub-components and how they are connected
 - Just like schematics, but using text
 - tedious to write, hard to decode
 - You get precise control over circuit details
 - May be necessary to map to special resources of the FPGA/ASIC
 - **Behavioral Verilog**
 - Describe what a component does, not how it does it
 - Synthesized into a circuit that has this behavior
 - Result is only as good as the tools
- Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.

```
name          port list
module addr_cell (a, b, cin, s, cout);
  input    a, b, cin;
  output   s, cout;           port declarations (input,
                                output, or inout)
                                module body
endmodule

module adder (A, B, S);
  addr_cell ac1 ( ... connections ... );
endmodule
```

keywords

name

port list

port declarations (input, output, or inout)

Instance of `addr_cell`

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.

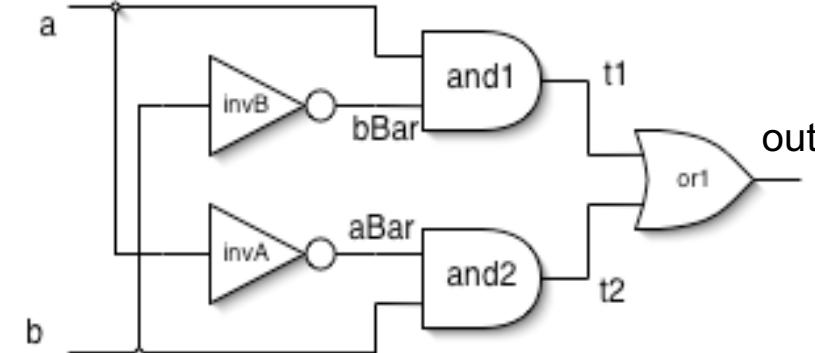
Structural Model - XOR example

```
module xor_gate( out, a, b );
    input a, b;
    output out;
    wire aBar, bBar, t1, t2;
    not invA (aBar, a);      instances
    not invB (bBar, b);
    and and1 (t1, a, bBar);
    and and2 (t2, b, aBar);
    or  or1 (out, t1, t2);

endmodule
```

module name
port list
port declarations
internal signal declarations

Built-in gates
instances
Instance name
Interconnections (note output is first)



- Notes:
 - The instantiated gates are not “executed”. They are active always.
 - xor gate already exists as a built-in (so really no need to define it).
 - Undeclared variables assumed to be wires. Don’t let this happen to you!

Simple Behavioral Model

```
module foo (out, in1, in2);
    input      in1, in2;
    output     out;
    assign out = in1 & in2;
endmodule
```

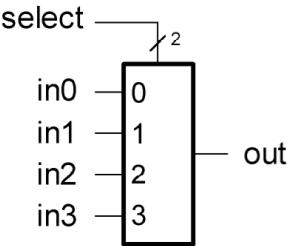
“continuous assignment”
Connects out to be the logical “and” of in1 and in2.

Short-hand for explicit instantiation of bit-wise “and” gate (in this case).

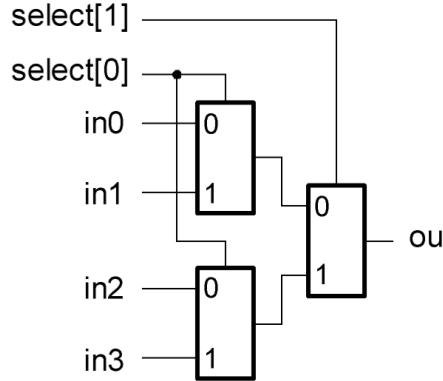
The assignment **continuously happens**, therefore **any change on the rhs is reflected in out immediately** (except for the small delay associated with the implementation of the practical &).

Not like an assignment in C that takes place when the program counter gets to that place in the program.

Instantiation, Signal Array, Named ports



a) 4-input mux symbol



b) 4-input mux implemented with 2-input muxes

```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0,in1,select;
    output out;
    wire s0,w0,w1;
    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or  (out, w0, w1);
endmodule // mux2
```

```
module mux4 (in0, in1, in2, in3, select, out);
    input in0,in1,in2,in3;
    input [1:0] select;           ----- Signal array. Declares select[1], select[0]
    output out;
    wire w0,w1;

    mux2
        m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
        m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
        m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

----- Signal array. Declares select[1], select[0]

----- Named ports. Highly recommended.

Example - Ripple Adder

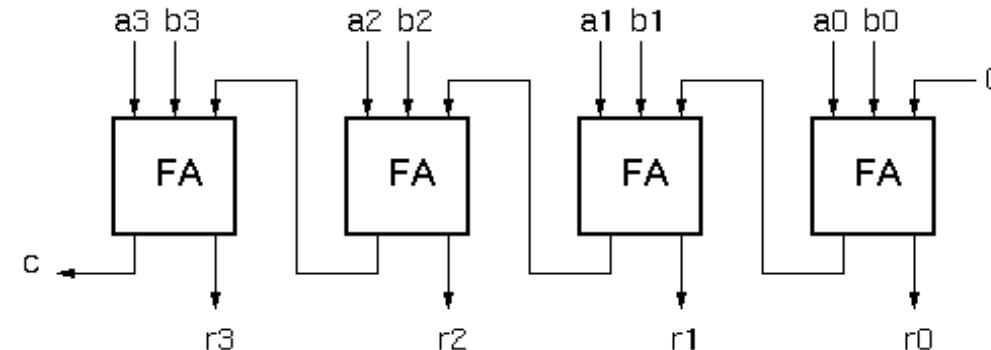
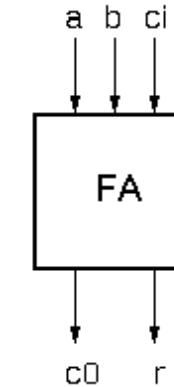
```
module FullAdder(a, b, ci, r, co);
    input a, b, ci;
    output r, co;

    assign r = a ^ b ^ ci;
    assign co = a&ci | a&b | b&ci;

endmodule
```

```
module Adder(A, B, R);
    input [3:0] A;
    input [3:0] B;
    output [4:0] R;

    wire c1, c2, c3;
    FullAdder
    add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
    add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
    add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
    add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
endmodule
```



Verilog Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
	reduction OR	Reduction
~&	reduction NAND	Reduction
~	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{}	concatenation	Concatenation
{ { }}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift

>	greater than	Relational
>=	greater than or equal to	Relational
<	less than	Relational
<=	less than or equal to	Relational
==	logical equality	Equality
!=	logical inequality	Equality
==	case equality	Equality
!=	case inequality	Equality
&	bit-wise AND	Bit-wise
^	bit-wise XOR bit-wise XNOR	Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

Continuous Assignment Examples

Assign values
whenever there is a
change in the RHS.

Model combinational
logic without
specifying an
interconnection of
gates.

```
wire [3:0] A, X,Y,R,Z;  
wire [7:0] P;  
wire r, a, cout, cin;
```

```
assign R = X | (Y & ~Z);           ← use of bit-wise Boolean operators  
assign r = &X;                     ← example reduction operator  
assign R = (a == 1'b0) ? X : Y;   ← conditional operator  
assign P = 8'hff;                 ← example constants  
assign P = X * Y;                ← arithmetic operators (use with care!)  
assign P[7:0] = {4{X[3]}, X[3:0]}; ← (ex: sign-extension)  
assign {cout, R} = X + Y + cin;   ← bit field concatenation  
assign Y = A << 2;              ← bit shift operator  
assign Y = {A[1], A[0], 1'b0, 1'b0}; ← equivalent bit shift
```

Non-Continuous Assignments

A bit unusual from a hardware specification point of view.

Shows off Verilog roots as a simulation language.

“always” block example:

```
module and_or_gate (out, in1, in2, in3);
    input      in1, in2, in3;
    output     out;
    reg        out;          “reg” type declaration. Not really a register in
                            this case. Just a Verilog idiosyncrasy.

    always @{in1 or in2 or in3} begin
        out = (in1 & in2) | in3;
    end
endmodule
```

“sensitivity” list, triggers the action in the body.

brackets multiple statements (not necessary in this example).

Isn’t this just: `assign out = (in1 & in2) | in3;`?
Why bother?

Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

case statement example:

```
module mux4 (in0, in1, in2, in3, select, out);
    input in0,in1,in2,in3;
    input [1:0] select;
    output      out;
    reg         out;

    always @ (in0 in1 in2 in3 select)
        case (select)
            2'b00: out=in0;
            2'b01: out=in1;
            2'b10: out=in2;
            2'b11: out=in3;
        endcase
    endmodule // mux4
```

keyword The statement(s) corresponding to
 whichever constant matches
 "select" get applied.

Couldn't we just do this with nested "if"s?

Well yes and no!

Always Blocks

Nested if-else example:

```
module mux4 (in0, in1, in2, in3, select, out);
    input in0,in1,in2,in3;
    input [1:0] select;
    output      out;
    reg         out;

    always @ (in0 in1 in2 in3 select)
        if (select == 2'b00)
            out=in0;
        else if (select == 2'b01)
            out=in1;
        else if (select == 2'b10)
            out=in2;
        else
            out=in3;
endmodule // mux4
```

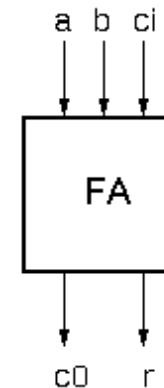
Nested if structure leads to “priority logic” structure:
with different delays for different inputs
(in3 to out delay > than in0 to out delay).
Case version treats all inputs the same.

Generate: Ripple-Carry Adder Example

```
module FullAdder(a, b, ci, r, co);
    input a, b, ci;
    output r, co;

    assign r = a ^ b ^ ci;
    assign co = a&ci + a&b + b&ci;

endmodule
```



```
module Adder(A, B, R);
    input [3:0] A;
    input [3:0] B;
    output [4:0] R;

    wire c1, c2, c3;
    FullAdder
        add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
        add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
        add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
        add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );
endmodule
```

```
graph LR
    FA0[FA] -- a0 --> Ia0
    FA0 -- b0 --> Ib0
    FA0 -- ci0 --> Ici0
    Ia0 --> FA0
    Ib0 --> FA0
    Ici0 --> FA0
    FA0 -- r0 --> Or0
    FA0 -- co0 --> C1
    FA1[FA] -- a1 --> Ia1
    FA1 -- b1 --> Ib1
    FA1 -- ci1 --> Ici1
    Ia1 --> FA1
    Ib1 --> FA1
    Ici1 --> FA1
    FA1 -- r1 --> Or1
    FA1 -- co1 --> C2
    FA2[FA] -- a2 --> Ia2
    FA2 -- b2 --> Ib2
    FA2 -- ci2 --> Ici2
    Ia2 --> FA2
    Ib2 --> FA2
    Ici2 --> FA2
    FA2 -- r2 --> Or2
    FA2 -- co2 --> C3
    FA3[FA] -- a3 --> Ia3
    FA3 -- b3 --> Ib3
    FA3 -- ci3 --> Ici3
    Ia3 --> FA3
    Ib3 --> FA3
    Ici3 --> FA3
    FA3 -- r3 --> Or3
    C1 --> FA1
    C2 --> FA2
    C3 --> FA3
    Or0 --> OR[0]
    Or1 --> OR[1]
    Or2 --> OR[2]
    Or3 --> OR[3]
    OR[4] --> R[4]
```

Example - Ripple Adder Generator

Parameters give us a way to generalize our designs.

```
module Adder(A, B, R);  
    parameter N = 4;  
    input [N-1:0] A;  
    input [N-1:0] B;  
    output [N:0] R;  
    wire [N:0] C;  
  
    genvar i;  
  
    generate  
        for (i=0; i<N; i=i+1) begin:bit  
            FullAdder add(.a(A[i]), .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));  
        end  
    endgenerate  
  
    assign C[0] = 1'b0;  
    assign R[N] = C[N];  
endmodule
```

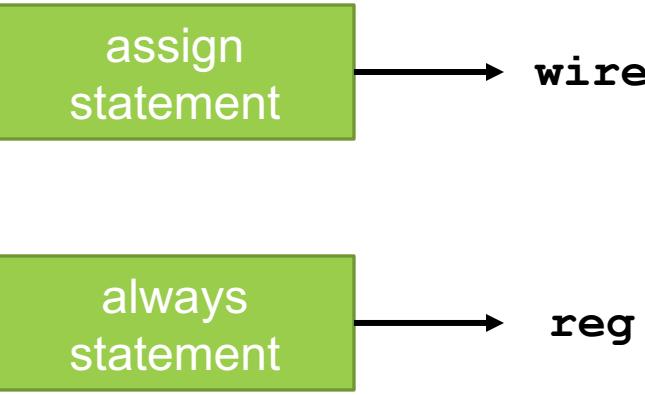
Declare a parameter with default value.
Note: this is not a port. Acts like a “synthesis-time” constant.
Replace all occurrences of “4” with “N”.
variable exists only in the specification - not in the final circuit.
Keyword that denotes synthesis-time operations
For-loop creates instances (with unique names)

```
Adder #(N(64)) adder64 ( ... );  
Overwrite parameter N  
at instantiation.
```

Simplified Verilog Guidelines

- Combinational logic:
 - Continuous Assignment:
`assign a = b & c;`
 - Always block with @(*)
`always @(*) begin`
`a = b & c; // blocking statement`

`end`



Are these combinational circuits correct?

```
// Example A: 3-Input Adder
always @(a or b) begin
    out = a + b + c;
end
```

```
// Example B:
assign out = in & out;
```

```
// Example C:
always @(*) begin
    case (select)
        2'b00: out=in0;
        2'b01: out=in1;
        2'b10: out=in2;
    endcase
end
```

Administrivia

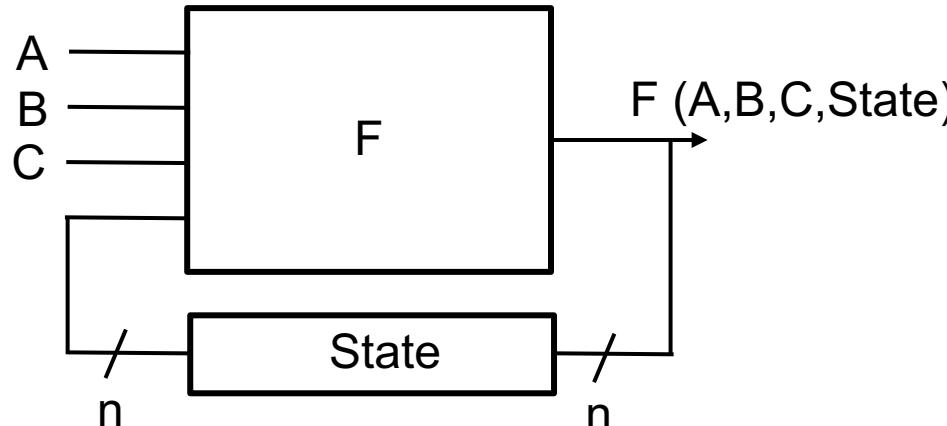
- Homework 1 out last week.
- All discussions start this week.
- Lab 2 start this week.
 - Please attend your assigned session.



- **Verilog!**
 - Combinational Circuits
 - Assign statement
 - Always blocks
 - Case statement
 - Generator
 - Sequential Circuits
 - Latches and FlipFlops
 - Always block
 - Sensitivity list
 - Blocking vs Nonblocking

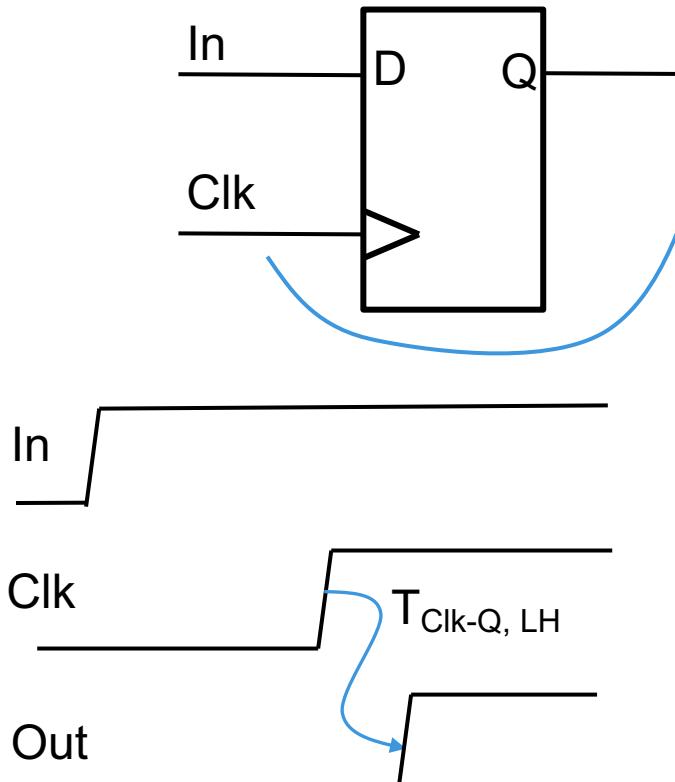
Sequential Logic

- Output is a function of both the current inputs and the state.
- State represents the memory.
- State is a function of previous inputs.
- In synchronous digital systems, state is updated on each clock tick.



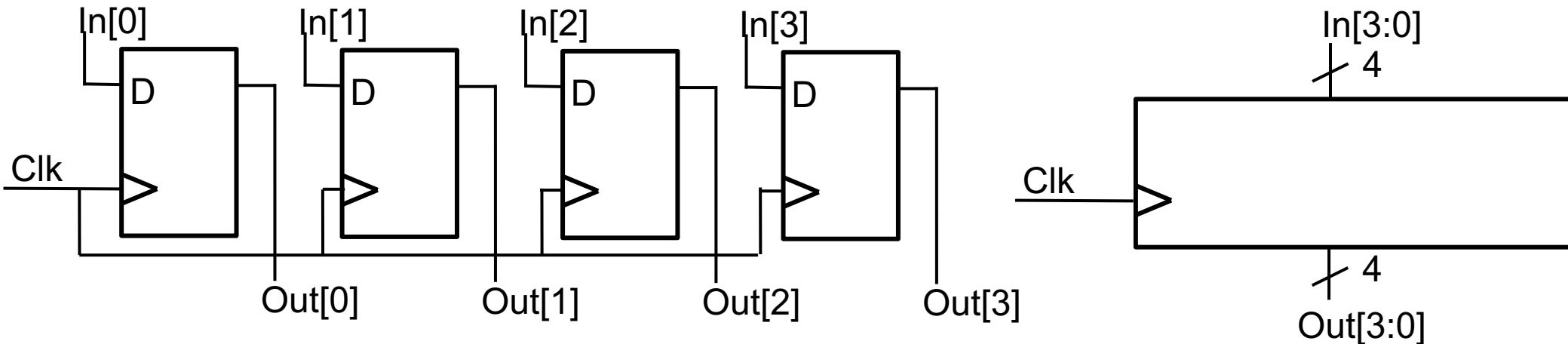
Timing

- Flip-flop timing
(latch timing will be covered later)

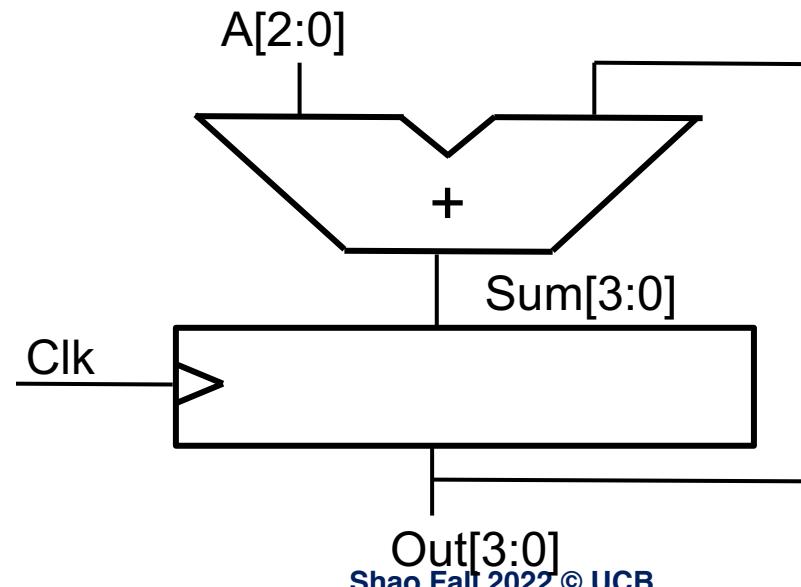


Register

- 4-bit register



- Accumulator



State Elements in Verilog

Always blocks are the only way to specify the “behavior” of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

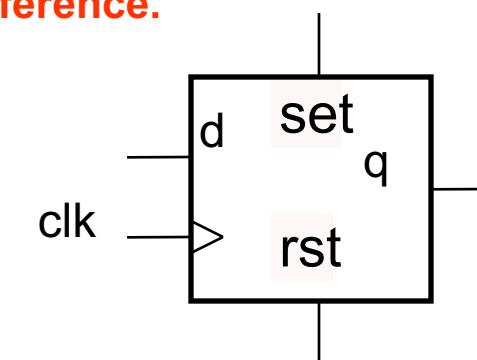
```
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @ (posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```

keyword

“always @ (posedge clk)” is key
to flip-flop inference.

This gives priority to
reset over set and set
over d.



The Sequential always Block

Combinational

```
module comb(input a, b, sel,
            output reg out);
    always @(*) begin
        if (sel) out = b;
        else out = a;
    end
endmodule
```

Sequential

```
module seq(input a, b, sel, clk,
            output reg out);
    always @ (posedge clk) begin
        if (sel) out <= b;
        else out <= a;
    end
endmodule
```

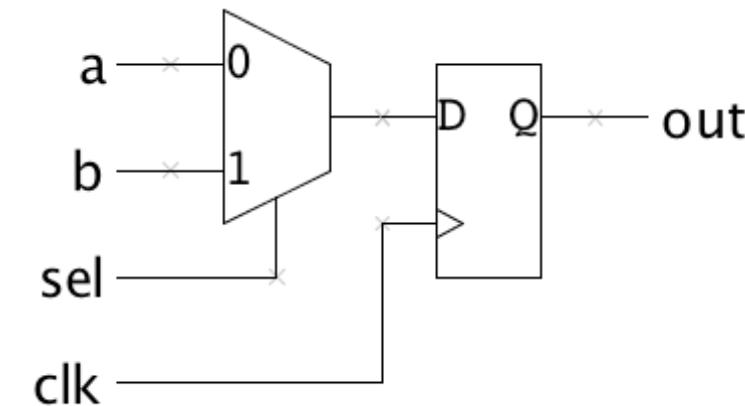
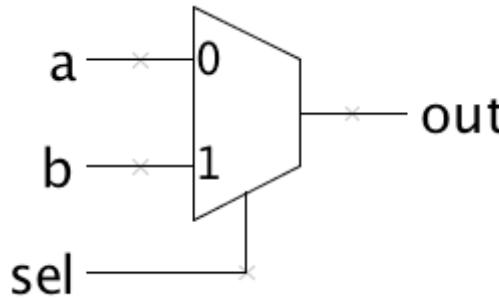
The Sequential always Block

Combinational

```
module comb(input a, b, sel,
             output reg out);
  always @(*) begin
    if (sel) out = b;
    else out = a;
  end
endmodule
```

Sequential

```
module seq(input a, b, sel, clk,
             output reg out);
  always @ (posedge clk) begin
    if (sel) out <= b;
    else out <= a;
  end
endmodule
```



Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
 - Blocking assignment (`=`): evaluation and assignment are immediate

```
always @(*) begin
    x = a | b;      // 1. evaluate a|b, assign result to x
    y = a ^ b ^ c; // 2. evaluate a^b^c, assign result to y
    z = b & ~c;    // 3. evaluate b&(~c), assign result to z
end
```

- Nonblocking assignment (`<=`): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active `always` blocks)

```
always @(*) begin
    x <= a | b;      // 1. evaluate a|b, but defer assignment to x
    y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
    z <= b & ~c;    // 3. evaluate b&(~c), but defer assignment to z
    // 4. end of time step: assign new values to x, y and z
end
```

Assignment Styles for Sequential Logic

```
module blocking(
    input in, clk,
    output reg out
);
    reg q1, q2;
    always @ (posedge clk) begin
        q1 = in;
        q2 = q1;
        out = q2;
    end
endmodule
```

```
module nonblocking(
    input in, clk,
    output reg out
);
    reg q1, q2;
    always @ (posedge clk) begin
        q1 <= in;
        q2 <= q1;
        out <= q2;
    end
endmodule
```

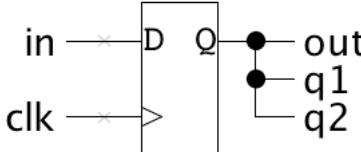
Use Nonblocking for Sequential Logic

```
always @ (posedge clk) begin  
    q1 = in;  
    q2 = q1; // uses new q1  
    out = q2; // uses new q2  
end
```

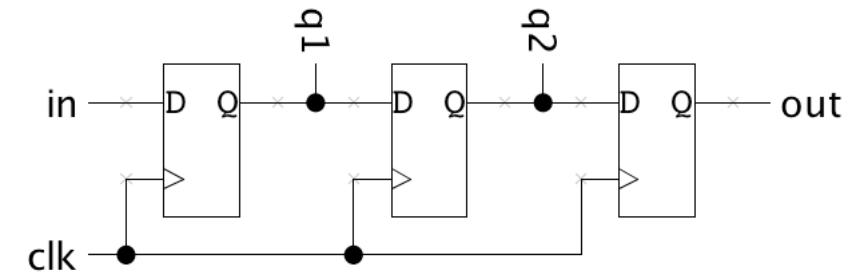
```
always @ (posedge clk) begin  
    q1 <= in;  
    q2 <= q1; // uses old q1  
    out <= q2; // uses old q2  
end
```

(“old” means value before clock edge, “new” means the value after most recent assignment)

“At each rising clock edge, $q_1 = \text{in}$.
After that, $q_2 = q_1$.
After that, $\text{out} = q_2$.
Therefore $\text{out} = \text{in}$.”



“At each rising clock edge, q_1 , q_2 , and out simultaneously receive the old values of in , q_1 , and q_2 .”

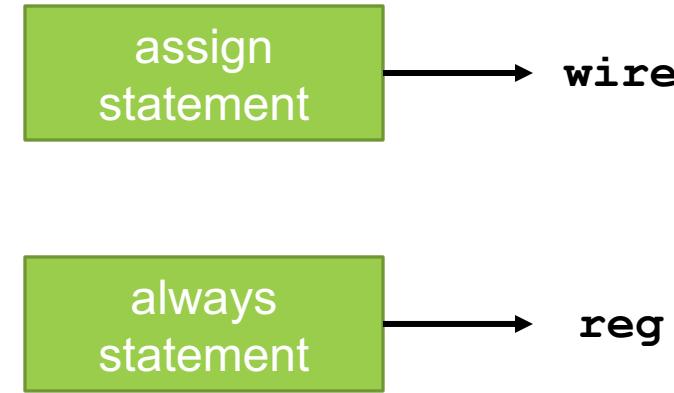


- Blocking assignments **do not** reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use **nonblocking** assignments for sequential always blocks

Simplified Verilog Guidelines

- Combinational logic:
 - Continuous Assignment:
`assign a = b & c;`
 - Always block with @(*)
`always @(*) begin`
`a = b & c; // blocking statement`
`end`

- Sequential logic:
 - Always block with @(posedge clk)
`always @(posedge clk) begin`
`a <= b & c; // nonblocking statement`
`end`



Verilog in EECS 151/251A

- We use behavioral modeling at the bottom of the hierarchy
- Use instantiation to:
 - 1) build hierarchy and,
 - 2) map to FPGA and ASIC resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
 - No other alternative: ex: state elements, case
 - Helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
 - Harris & Harris book chapter 4 is a good source.
 - ***Be careful of what you read on the web.*** Many bad examples out there.
 - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final Thoughts on Verilog Examples

A large part of digital design is knowing how to write Verilog that gets you the desired circuit.

First understand the circuit you want then figure out how to code it in Verilog.

If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

Review

- Verilog
 - Hardware description language: describe hardware!
 - Logic synthesis: Verilog -> Gate-level netlists
 - Used in both ASIC and Verilog
 - Modules
 - Structural vs behavioral
 - Operators and logic values
 - Combinational Circuits
 - Assign statement
 - Always blocks
 - Sequential Circuits
 - Always blocks
 - Blocking vs NonBlocking