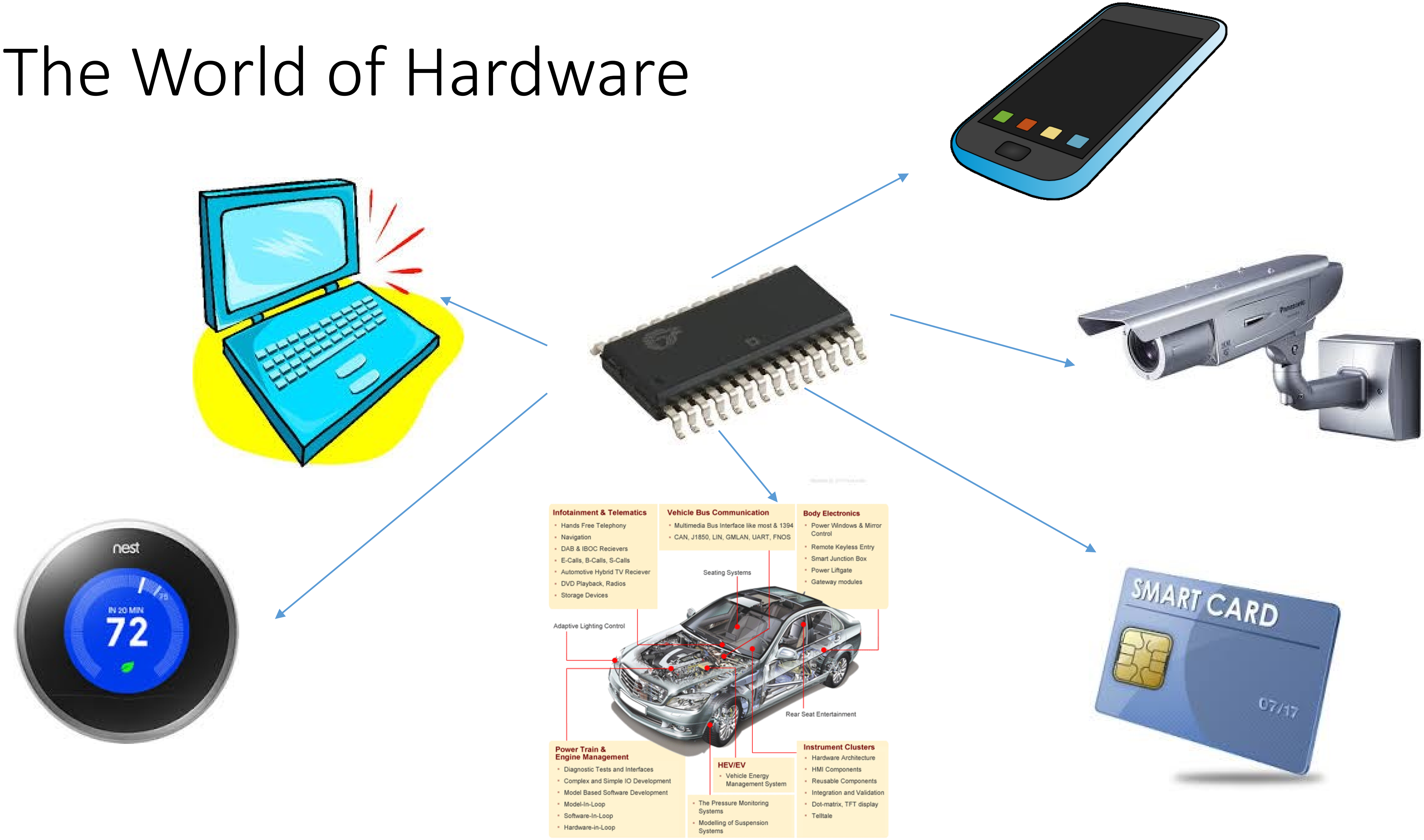
Lecture 1: Introduction and Basic Verilog

EL GY 6463: Advanced Hardware Design

Instructor: Siddharth Garg

The World of Hardware



Infotainment & Telematics

- Hands Free Telephony
- Navigation
- DAB & IBOC Recievers
- E-Calls, B-Calls, S-Calls
- Automotive Hybrid TV Reciever
- DVD Playback, Radios
- Storage Devices

Vehicle Bus Communication

- Multimedia Bus Interface like most & 1394
- CAN, J1850, LIN, GMLAN, UART, FNOS

Body Electronics

- Power Windows & Mirror Control
- Remote Keyless Entry
- Smart Junction Box
- Power Liftgate
- Gateway modules

Power Train & Engine Management

- Diagnostic Tests and Interfaces
- Complex and Simple IO Development
- Model Based Software Development
- Model-In-Loop
- Software-In-Loop
- Hardware-in-Loop

HEV/EV

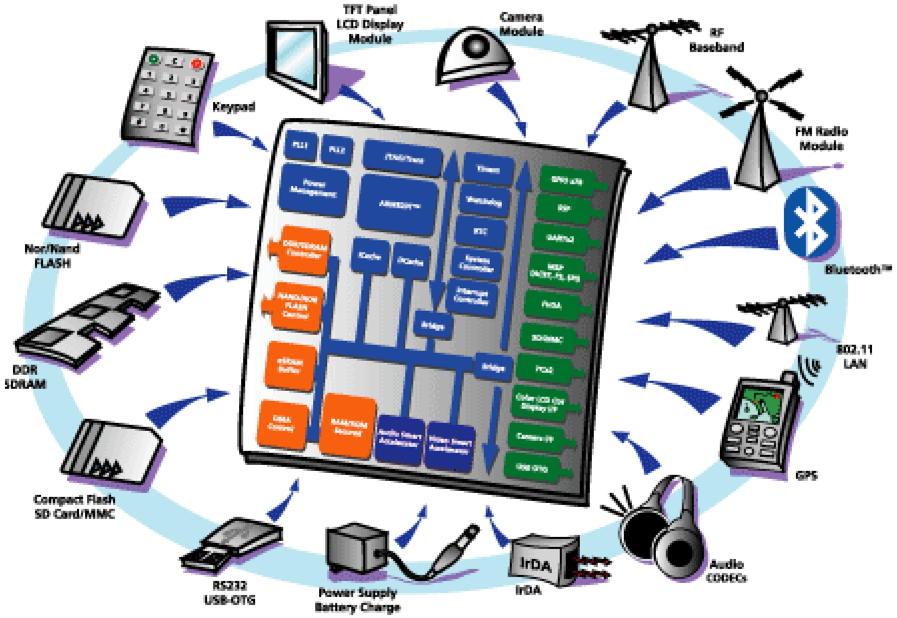
- Vehicle Energy Management System

- The Pressure Monitoring Systems
- Modelling of Suspension Systems

Instrument Clusters

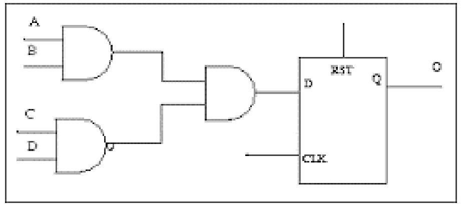
- Hardware Architecture
- HMI Components
- Reusable Components
- Integration and Validation
- Dot-matrix, TFT display
- Telltale

Hardware “Stack”



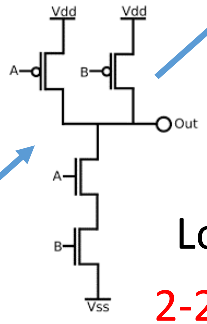
System-on-chip (SoC)

1M to > 1 Billion transistors!



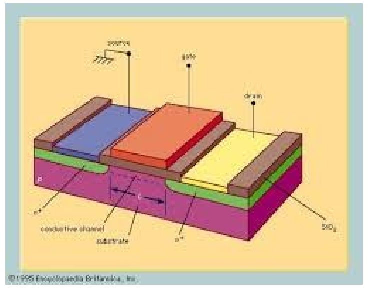
Logic Circuit/Netlist

1k-100K transistors



Logic gate

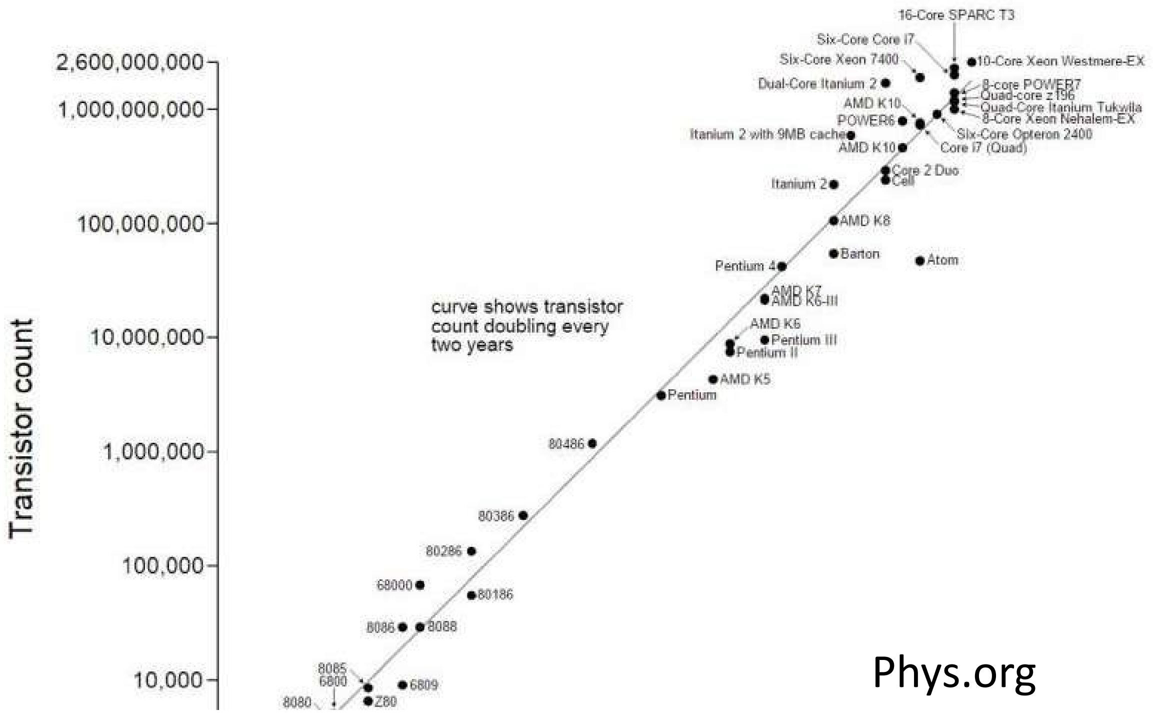
2-20 transistors



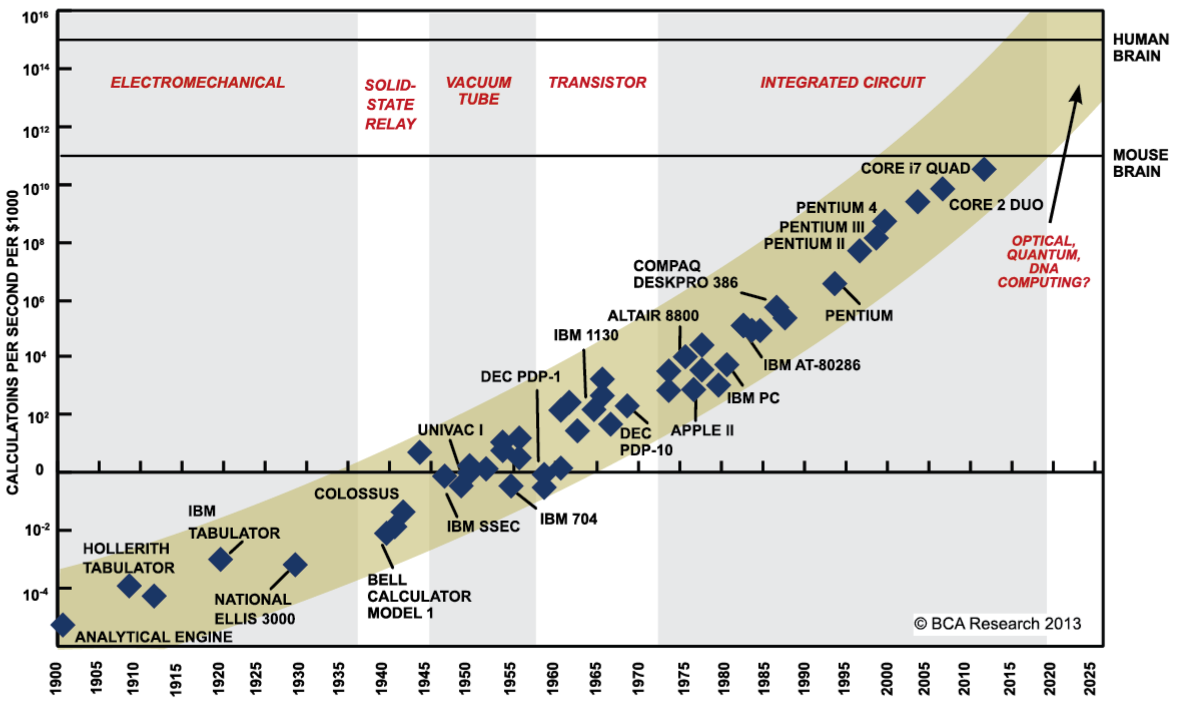
CMOS Transistor

How did we get here? Moore’s Law

- Gordon Moore predicted in 1965 that the number of transistors that could be “crammed” in a chip would double every 2 years
  - Updated to double every 1.5 years



Phys.org



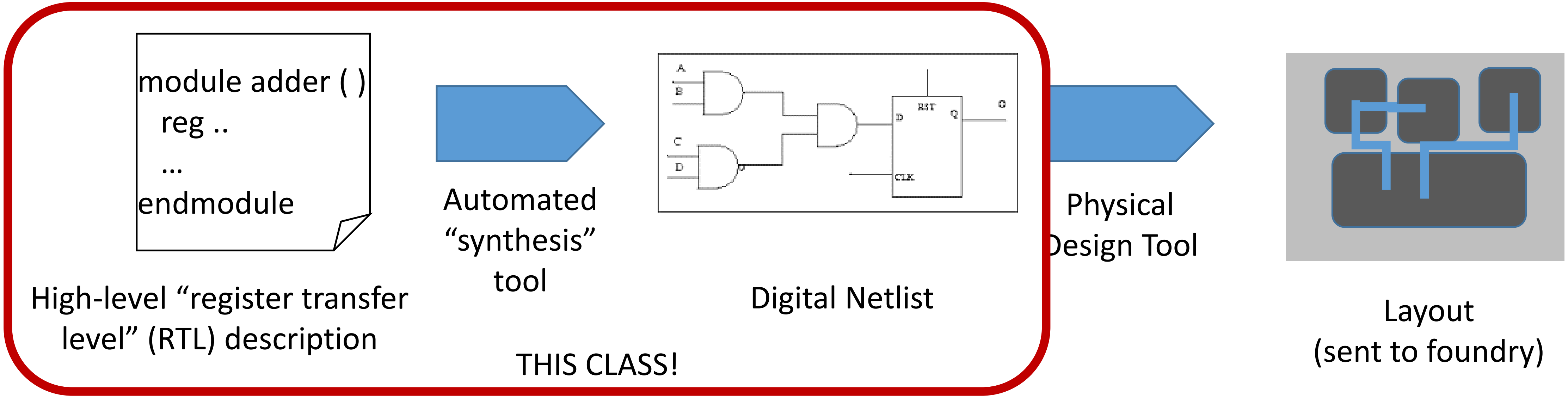
SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

How Are ICs (chips) Designed

Chips typically have two parts: logic (compute) and memory (store). This class focuses on logic design

At first, chips were designed by hand – custom design

But, as the number of transistors on a chip grew, designing by hand became tedious and error-prone – automated design flow and tools



```
module adder ( )
  reg ..
  …
endmodule
```

High-level “register transfer level” (RTL) description

Automated “synthesis” tool

Digital Netlist

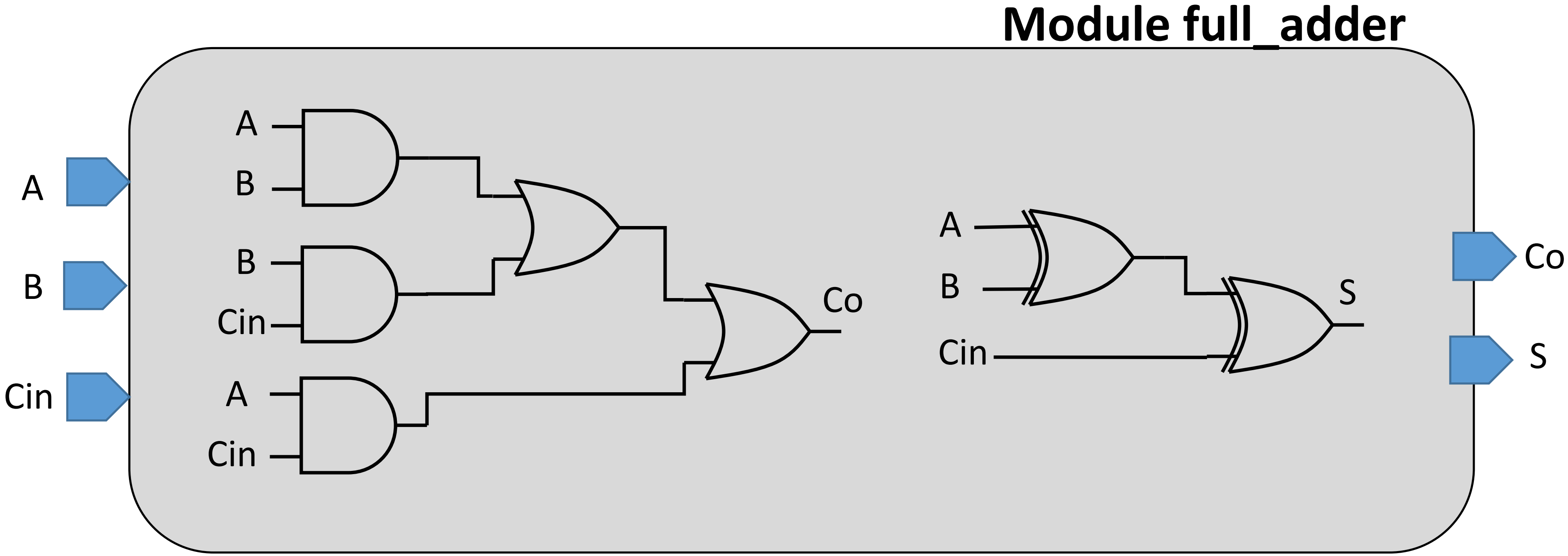
Physical Design Tool

Layout (sent to foundry)

THIS CLASS!

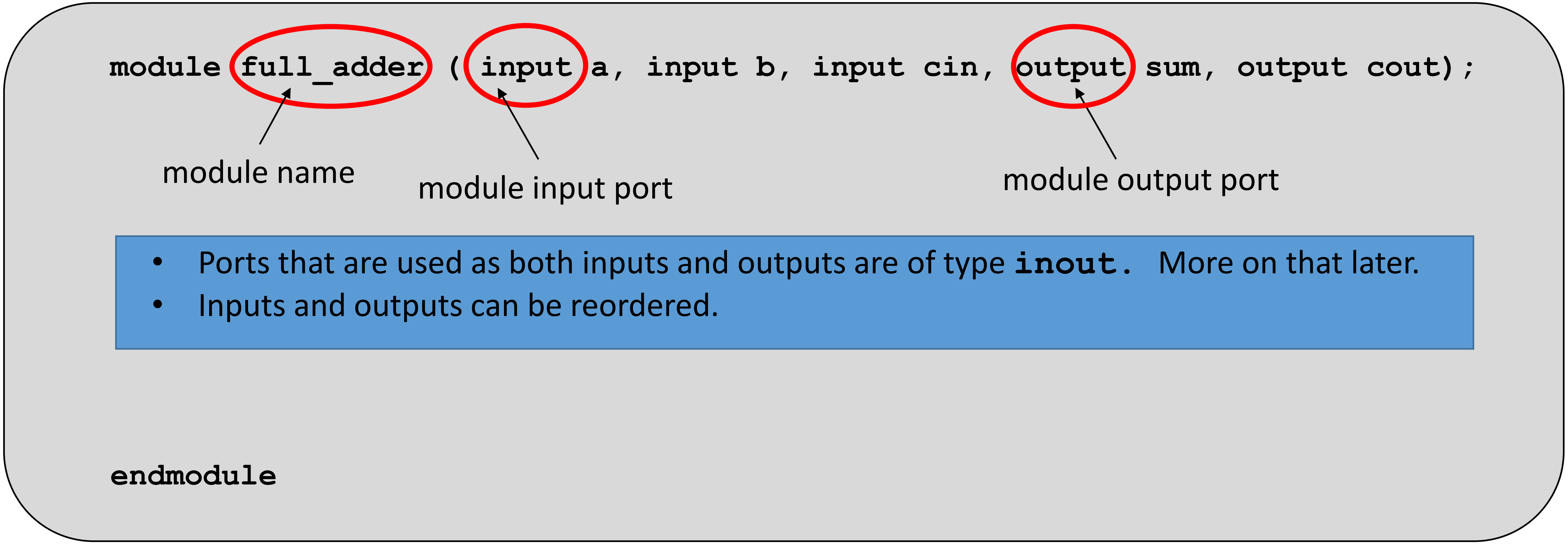
Our First Example: Full Adder

- Three inputs (A, B, Cin) and two outputs (S, Co)
  - Co = A&B | B&C | C&A (& = and ; | = or)
  - S = A ^ B ^ C (Note: ^ = xor)



Verilog Modules

- Building blocks of Verilog designs.
  - Code reuse (like functions in C/C++)
  - Hierarchical structure: modules can be instantiated within modules



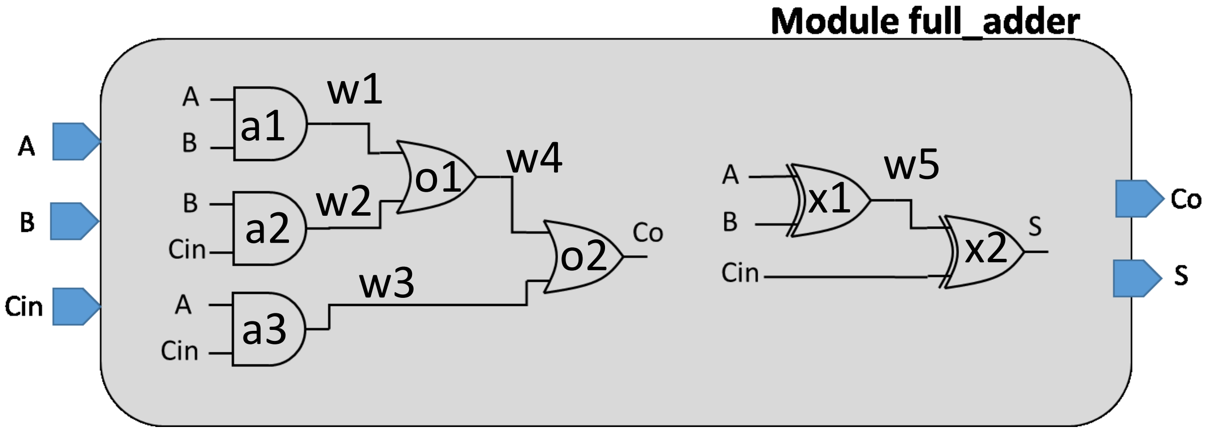
```
module full_adder ( input a, input b, input cin, output sum, output cout);
```

- Ports that are used as both inputs and outputs are of type **`inout`.** More on that later.
- Inputs and outputs can be reordered.

```
endmodule
```

Structural Descriptions

- Describes the gate-level structure of a combinational circuit



```
module full_adder ( input a, input b, input cin, output sum, output cout);

wire w1, w2, w3, w4, w5;

and a1 (w1, a, b),
    a2 (w2, b, cin),
    a3 (w3, a, cin);

or  o1 (w4, w1, w2),
    o2 (cout, w4, w3);

xor x1 (w5, a, b),
    x2 (sum, cin, w5);

endmodule
```

Built-in data type

Built-in Verilog modules

Instance name

output

input

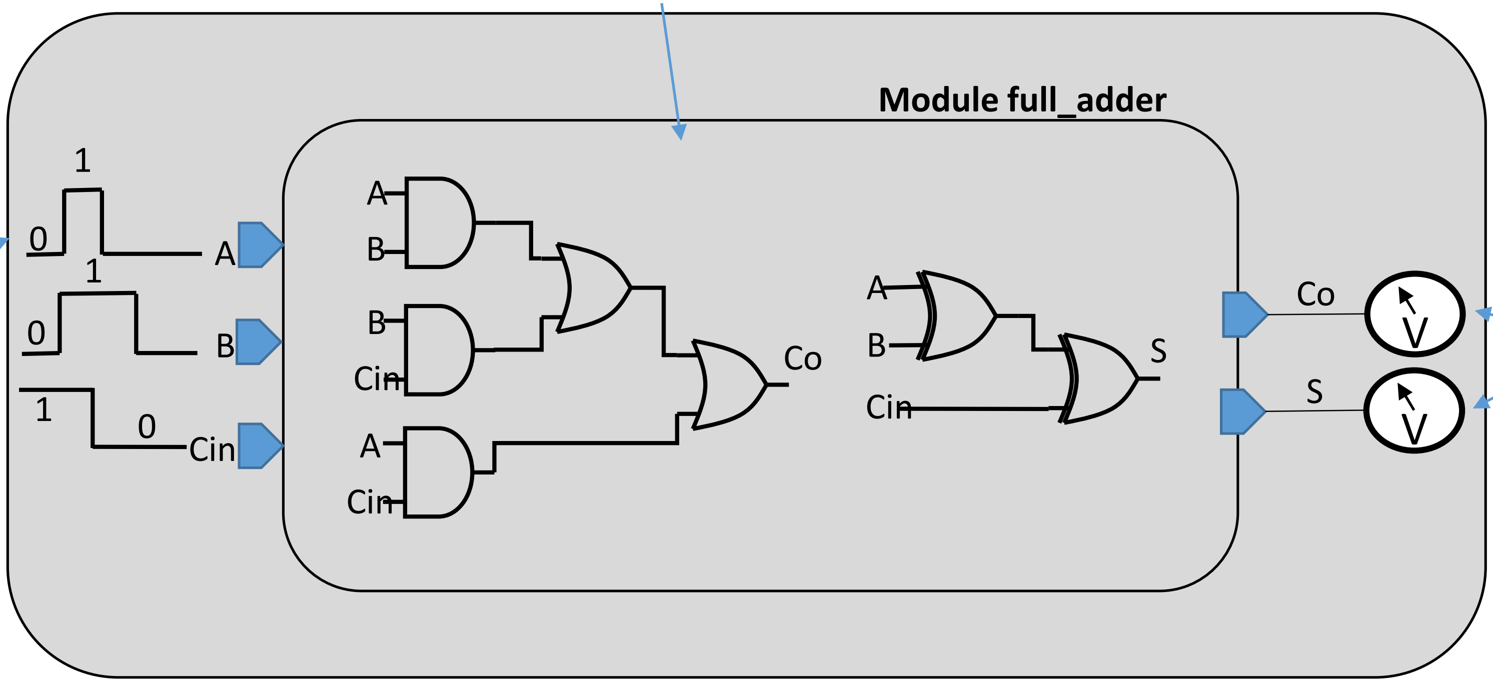
- Wire data type
  - used to connect modules
  - Inputs and outputs are of type wire by default
- In-built modules corresponding to major Boolean logic gates
  - and, or, xor, nand, nor, xnor, inv ….
  - See pp. 161, Table 6.1 of T&M for more

Simulating the Full Adder

- How do you check if the full adder works correctly?
    - Simulate full adder with different inputs
    - Design a module that provides inputs (stimuli) and displays outputs from the full adder. This is commonly called a testbench.

Top level module. Usually has no inputs/outputs.

“Design under Test” (DUT)



Apply stimuli in the form of Boolean waveforms

Monitor and displays the outputs

Testbenches

	wire	reg
Drive Strength	Passive. Driven by reg.	Store info. Used to drive wires
Connect to module inputs	Yes	Yes
Connect to module outputs	Yes	No
LHS in initial block	No	Yes

```
module test_bench;

reg a_tb, b_tb, cin_tb;

wire sum_tb, cout_tb;

full_adder fa(a_tb, b_tb, cin_tb, sum_tb, cout_tb);
// full_adder fa(.a(a_tb), .b(b_tb), .cin(cin_tb),
                 .sum(sum_tb), .cout(cout_tb));
initial
  begin
    $monitor ($time, “A=%b, B=%b, Cin=%b, Sum=%b, Cout=%b”,
                        a_tb, b_tb ,cin_tb, sum_tb, cout_tb);

      a_tb = 0; b_tb =0; cin_tb = 1;
    #1 a_tb = 1;
    #2 cin_tb = 0;
end
endmodule
```

Verilog datatype. Used to “drive” wires.

Inputs to DUT

Outputs from DUT

DUT

Comment

Alternative declaration

Initial block is executed ***once*** at t=0

Like printf

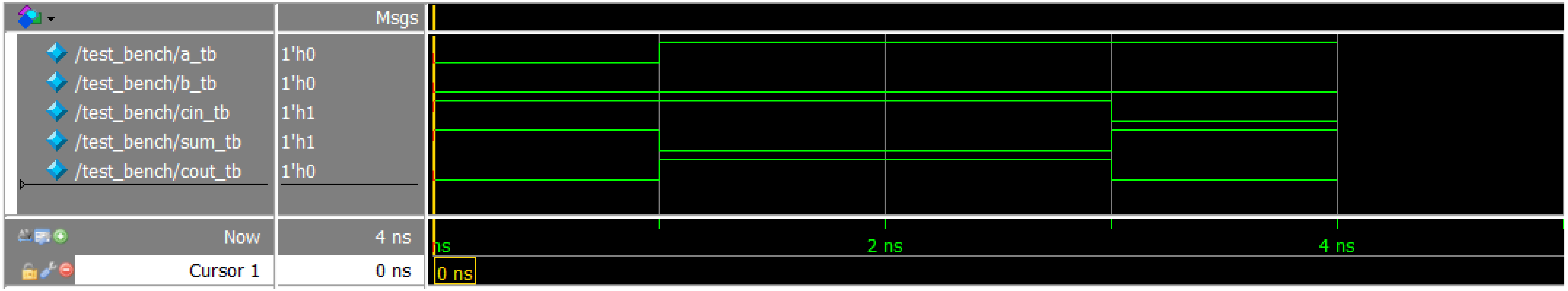
Wait for 1 unit of *simulated* time

Must be of type reg

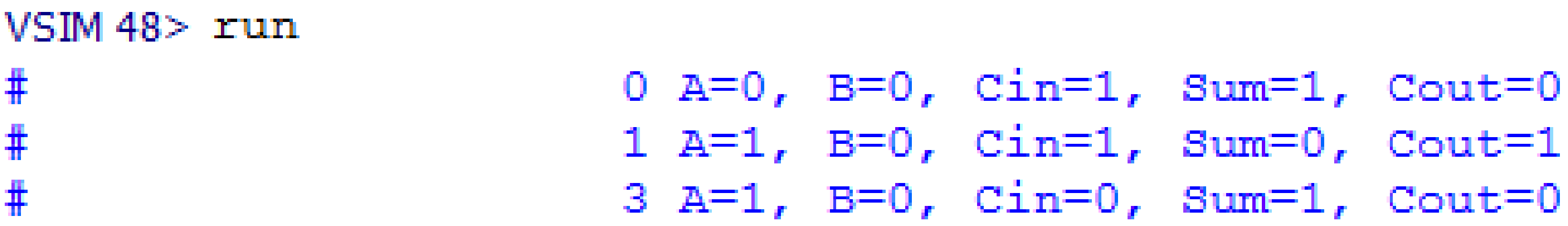
“blocking” assignment operator. More on blocking vs. non-blocking later.

Note: use outside initial block illegal! Try it.

Simulation Output



	Msgs
/test_bench/a_tb	1'h0
/test_bench/b_tb	1'h0
/test_bench/cin_tb	1'h1
/test_bench/sum_tb	1'h1
/test_bench/cout_tb	1'h0
Now	4 ns
Cursor 1	0 ns



```
VSIM 48> run
#                          0 A=0, B=0, Cin=1, Sum=1, Cout=0
#                          1 A=1, B=0, Cin=1, Sum=0, Cout=1
#                          3 A=1, B=0, Cin=0, Sum=1, Cout=0
```

Checkpoint

```
module test_bench;

~~reg~~ wire a_tb, b_tb, cin_tb; //will this work?

~~wire~~ reg sum_tb, cout_tb;   //will this work?

full_adder fa(a_tb, b_tb, cin_tb, sum_tb, cout_tb);
// full_adder fa(.a(a_tb), .b(b_tb), .cin(cin_tb),
                 .sum(sum_tb), .cout(cout_tb));
initial
  begin
    $monitor ($time, “A=%b, B=%b, Cin=%b, Sum=%b, Cout=%b”,
                      a_tb, b_tb ,cin_tb, sum_tb, cout_tb);

       a_tb = 0; b_tb =0; cin_tb = 1;
    #1 a_tb = 1;
    #2 cin_tb = 0;
end
endmodule
```

	wire	reg
Drive Strength	Passive. Driven by reg.	Store info. Used to drive wires
Connect to module inputs	Yes	Yes
Connect to module outputs	Yes	No
LHS in initial block	No	Yes

In-class Exercise #1

- Problem 1.16 (modified) from T&M.

*It is worth buying hotdogs (**Hot_H**) a weekday (**Week_H**) whenever it is lunchtime (**Lunch_H**) or after midnight (**Midnight_H**), but only if none of your professors are swimming in the pool **(No_Profs_H).** You always buy hotdogs on weekends!*

***Hot_H:** asserted if you should buy hotdogs*

***Week_H:** asserted if it is a weekday*

***Lunch_H:** asserted during lunch hour*

***Midnight_H:** asserted if after midnight*

***No_Profs_H:** asserted if no profs swimming*

*Write a Verilog module **buy_hotdog** that helps you decide whether to buy a hotdog or not.*

My Solution

```
module buy_hotdog ( input week_h, input lunch_h,
                    input midnight_h, input no_profs_h, output hot_h);

wire lch_mnght_h, lch_mnght_noprofs_h, buy_wkday, not_wkday;

or  o1 (lch_mnght_h, lunch_h, midnight_h);
and a1 (lch_mnght_noprofs_h, lch_mnght_h, no_profs_h);
and a2 (buy_wkday, lch_mnght_noprofs_h, week_h);
not n1 (not_wkday, week_h);
or  o2 (hot_h, buy_wkday, not_wkday);

endmodule
```

Other gate types: nand, nor, xor, xnor

Behavioral Description

- Alternatively, we can specify the Boolean logic equations that represent the combinational circuit

	wire	**reg**
Drive Strength	Passive. Driven by reg.	Store info. Used to drive wires
Connect to module inputs	Yes	Yes
Connect to module outputs	Yes	No
LHS in *initial* block	No	Yes
LHS in *assign*	Yes	No

```
module full_adder_bhv ( input a, input b, input cin, output sum, output cout);
```

Continuous assign updates the value of LHS *any time* the RHS variables change.

LHS can *only* be of type *wire*. Illegal to use inside *initial* block.

```
assign sum = a ^ b ^ cin;
```

Boolean **xor** operator

```
assign cout = (a&b) | (b&cin) | (cin&a);

endmodule
```

Bitwise **or** operator

Bitwise **and** operator

Testbench for Behavioral Description

- Behavioral description can be swapped in seamlessly!

```
module test_bench;

reg a_tb, b_tb, cin_tb;

wire sum_tb, cout_tb;

full_adder_bhv fa(a_tb, b_tb, cin_tb, sum_tb, cout_tb);

initial
  begin
    $monitor ($time, “A=%b, B=%b, Cin=%b, Sum=%b, Cout=%b”,
                    a_tb, b_tb ,cin_tb, sum_tb, cout_tb);

       a_tb = 0; b_tb =0; cin_tb = 1;
    #1 a_tb = 1;
    #2 cin_tb = 0;
end
endmodule
```

In-class Exercise #2

- Problem 1.16 (modified) from T&M.

*It is worth buying hotdogs (**Hot_H**) on a weekday (**Week_H**) whenever it is lunchtime (**Lunch_H**) or after midnight (**Midnight_H**), but only if none of your professors are swimming in the pool **(No_Profs_H).** You always buy hotdogs on weekends!*

***Hot_H:** asserted if you should buy hotdogs*

***Week_H:** asserted if it is a weekday*

***Lunch_H:** asserted during lunch hour*

***Midnight_H:** asserted if after midnight*

***No_Profs_H:** asserted if no profs swimming*

*Write a behavioral Verilog module **buy_hotdog** that helps you decide whether to buy a hotdog or not. How many lines of code did you save!?*

My Solution

```
module buy_hotdog ( input week_h, input lunch_h,
                    input midnight_h, input no_profs_h, output hot_h);
assign hot_h = (week_h & ((lunch_h | midnight_h) & (~no_profs_h)))
               | (~week_h);

endmodule
```

Precedence rules in book, but note that it’s always better to use ().

Procedural Description

- Allows the functionality using a step-by-step procedure
  - Like C code

```
module full_adder ( input a, input b, input cin, output reg sum, output reg
cout);

always @ (a, b, cin) begin
  sum  = 0;
  cout = 0;

  if (a^b^cin) begin
    sum = 1;
  end

  if ((a&b) | (b&cin) | (cin&a)) begin
    cout = 1;
  end
end
endmodule
```

Sensitivity list

Called every time a variable in the sensitivity list changes

Sum is set to 1 only if a^b^cin = TRUE, remains at 0 otherwise

LHS must be reg.

Blocking assignment operator. Can be used in both *initial* and *always* blocks.

	wire	**reg**
Drive Strength	Passive. Driven by reg.	Store info. Used to drive wires
Connect to module inputs	Yes	Yes
Connect to module outputs	Yes	No
LHS in *initial* and *always* block	No	Yes
LHS in *assign*	Yes	No
Module output type	Yes (default)	Yes
Module input type	Yes (default)	No

But remember….

- When using an always @ block to specify combinational logic, check for the following:

1) Rule 1: All inputs that the output depends on must be specified. In the previous example, the following will not work

```
                    always @ (a, b)
always @ (*) // instead use this. automatically
                 determines inputs!
```

2) Rule 2: always make sure the outputs are set to a value for every possible combination of inputs

Rule 2: Example

```
always @ (a, b) begin
  if (a==0 && b==0) begin
    sum = 1;
  end
  if (a==0 && b==1) begin
    sum = 0;
  end
end
```

But what happens when a=1 and b=1? Since sum is not assigned a value and is of type reg, it will “remember” the value from the last time it was assigned. That will depend on the previous values of a and b. But combinational logic just depends on current values!

In-class Exercise #3

- Problem 1.16 (modified) from T&M.

*It is worth buying hotdogs (**Hot_H**) on a weekday (**Week_H**) whenever it is lunchtime (**Lunch_H**) or after midnight (**Midnight_H**), but only if none of your professors are swimming in the pool **(No_Profs_H).** You always buy hotdogs on weekends!*

***Hot_H:** asserted if you should buy hotdogs*

***Week_H:** asserted if it is a weekday*

***Lunch_H:** asserted during lunch hour*

***Midnight_H:** asserted if after midnight*

***No_Profs_H:** asserted if no profs swimming*

*Write a procedural Verilog module **buy_hotdog** that helps you decide whether to buy a hotdog or not. How many lines of code did you save!?*

My Solution

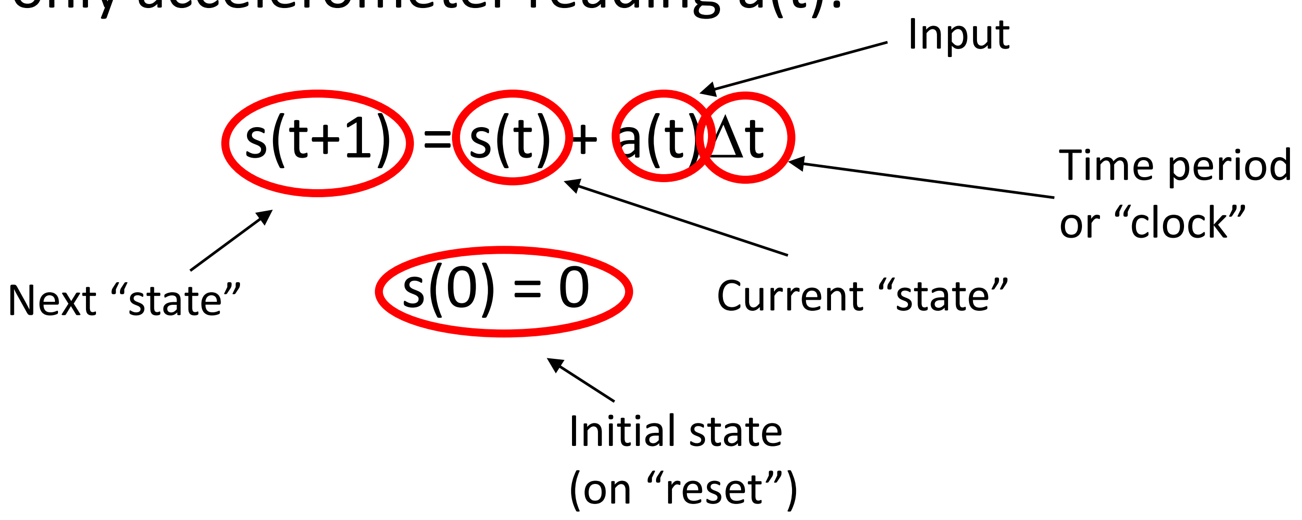
```
module buy_hotdog ( input week_h, input lunch_h,
                    input midnight_h, input no_profs_h, output reg hot_h);


always @(*) begin
   hot_h = 0;
   if (week_h == 0) begin
        if ((midnight_h | lunch_h) & (~no_profs_h))
                hot_h=1;
   end
   else
        hot_h = 1;
end
endmodule
```

Finite State Machines: Introduction

- Combinational logic: output depends on the current input only
- Finite State Machine: output depends on input *and* current state

Example: what is my current speed?

- Given only speedometer reading s(t): output = s(t)
- Given only accelerometer reading a(t):



$$s(t+1) = s(t) + a(t)\Delta t$$

$$s(0) = 0$$

But is this a finite state machine? Why or why not?

FSM: Formal Model

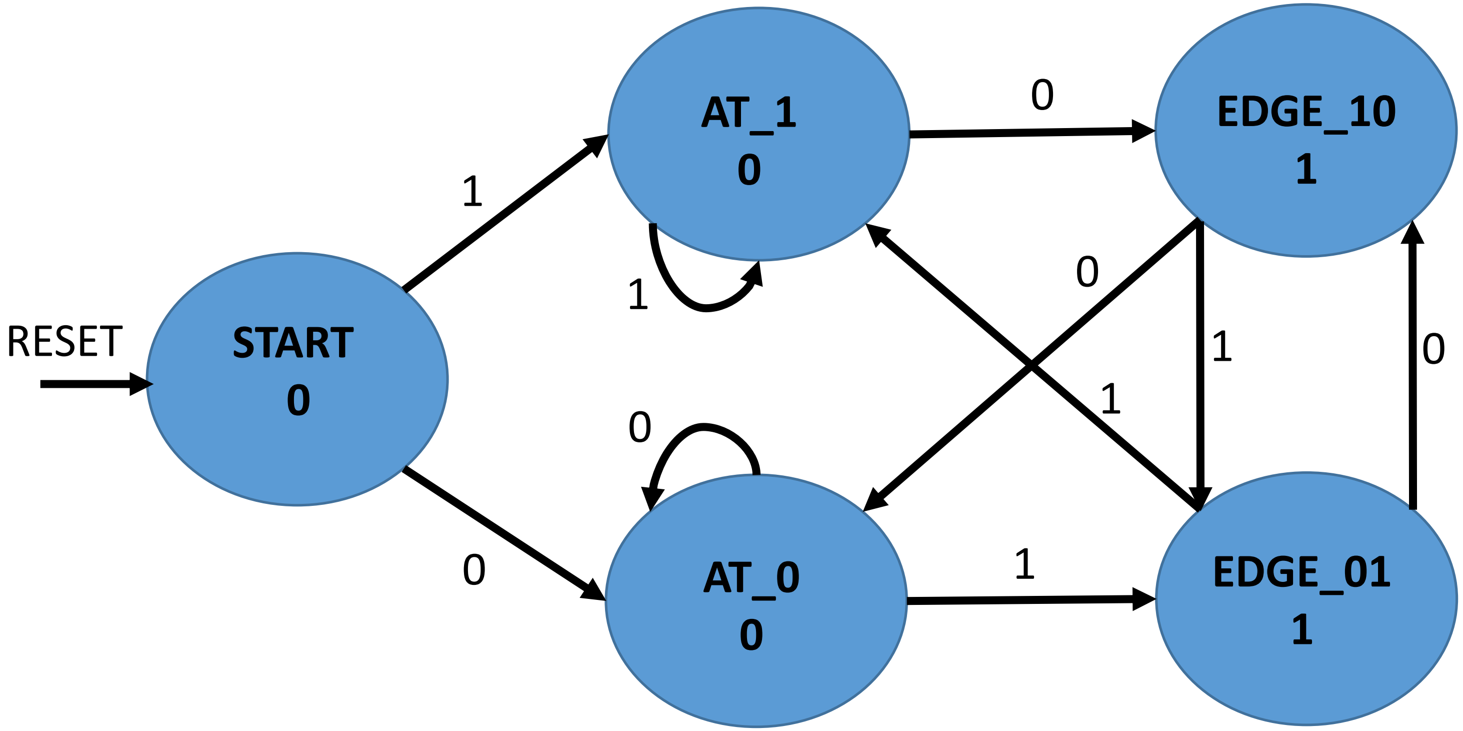
- A finite set of *states: S*
  - An initial state: $s_0 \in S$
- A finite set of *inputs:* Σ
- A finite set of *outputs:* Γ
- A state transition function $\delta : S \times \Sigma \to S$
- An output function ω
  - **Moore machine** $\omega : S \to \Gamma$ ← Output depends only on current state
  - **Mealy machine** $\omega : S \times \Sigma \to \Gamma$ ← Output depends on current state and input

FSM Example: Edge Detector

- Goal: detect 0->1 and 1->0 transitions in a sequence of inputs. Output a 1 when a transition occurs, zero otherwise.

Input: 00101111100001000

Output: 00111000010001100

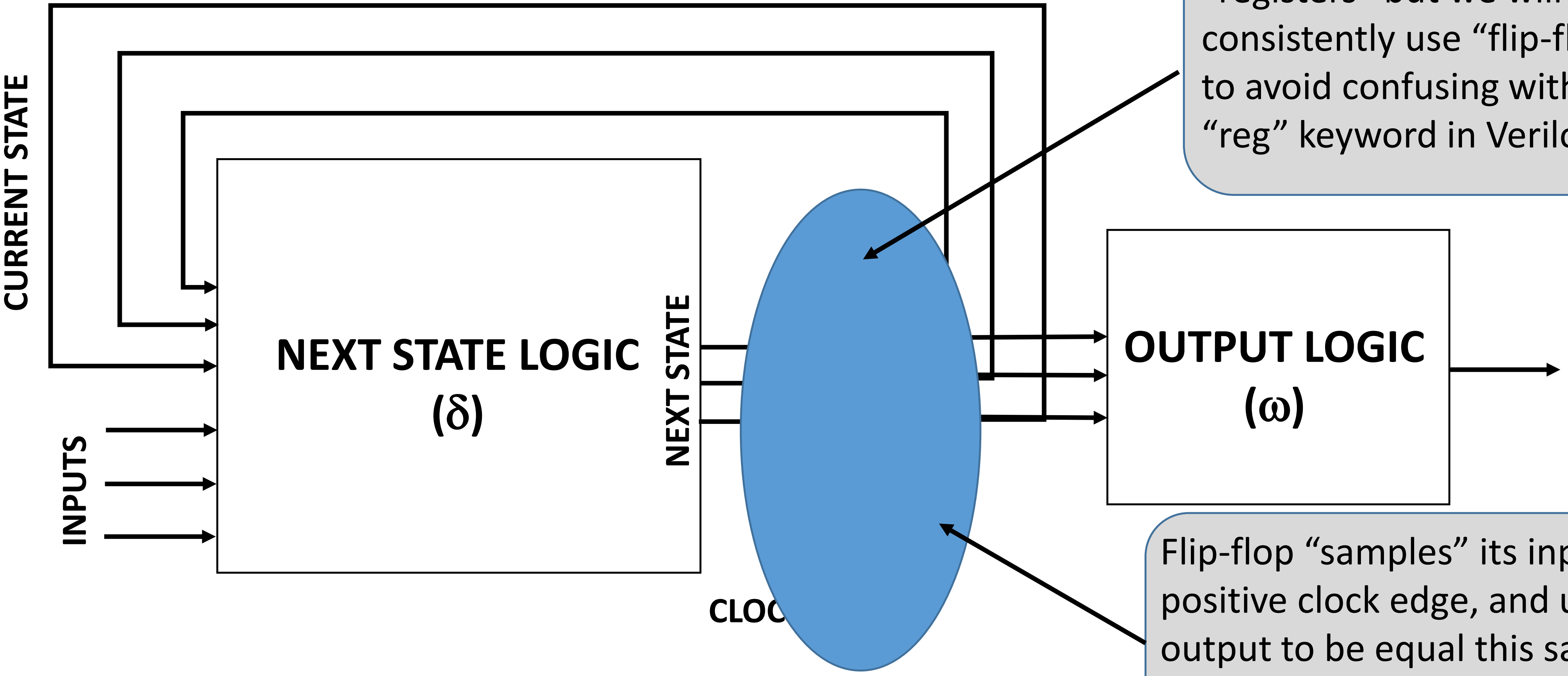


- *states S = {START, AT_1, AT_0, EDGE_10, EDGE_01}*
  - Initial state: s_0 = START
- *Inputs* $\Sigma = \{0,1\}$
- *Outputs* $\Gamma = \{0,1\}$

Is this a Mealy or a Moore machine?

http://courses.cs.washington.edu/courses/cse370/10sp/pdfs/lectures/15-VerilogIIPrint.pdf

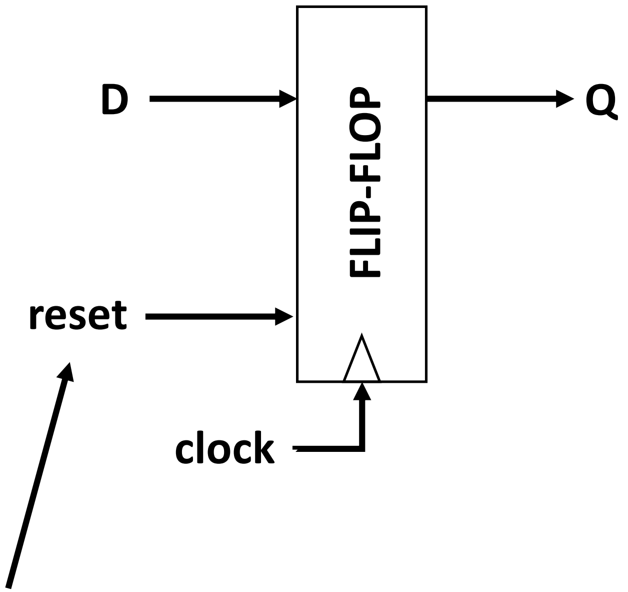
Clocked Implementation of a Moore FSM



Also referred to as "registers" but we will consistently use "flip-flops" to avoid confusing with the "reg" keyword in Verilog

Flip-flop "samples" its inputs at every positive clock edge, and updates the output to be equal this sampled value. The output does not change till the next positive edge

Flip-flops in Verilog



If reset = 1, output Q = 0.

- Asynchronous: Q = 0 as soon as reset becomes 1
- Synchronous: Q = 0 at next positive edge

D Flip-flop with Asynchronous Reset.

```
module dff ( input D, input reset,
             input clock, output reg Q);

always @ (posedge clock, posedge reset) begin
  if (reset)
     Q <= 0;
  else
     Q <= D;
end

endmodule
```

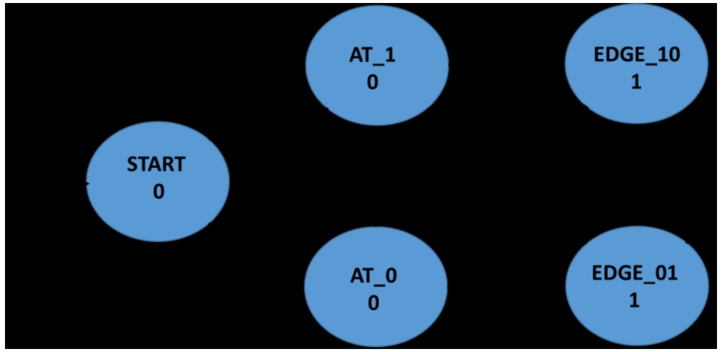
when clock transitions from 0 to 1

or reset transitions from 0 to 1

- **Non-blocking assignment** (remember that = is a blocking assignment)
- Rule of thumb: use non-blocking when always block is edge triggered. More on this later.

What about synchronous reset?

Edge Detector FSM in Verilog



- Let’s implement the **sequential portion** first
  - At positive clock edge, current state <= next state

```
module edge_detect ( input in, input clock, input reset, output reg out);


reg [2:0] current_state, next_state;

localparam START = 0, AT_1 = 1, AT_0 = 2, EDGE_10 = 3, EDGE_01 = 4;
//localparam START = 3’b000, AT_1 = 3’b001, AT_2 = 3’b010 … ;
always @ (posedge clock, posedge reset) begin
  if (reset)
      current_state <= START;
  else
      current_state <= next_state;
end

endmodule
```

Three bits. Why?

Three bits.

Constant defined only within the scope of current module

Edge Detector FSM in Verilog

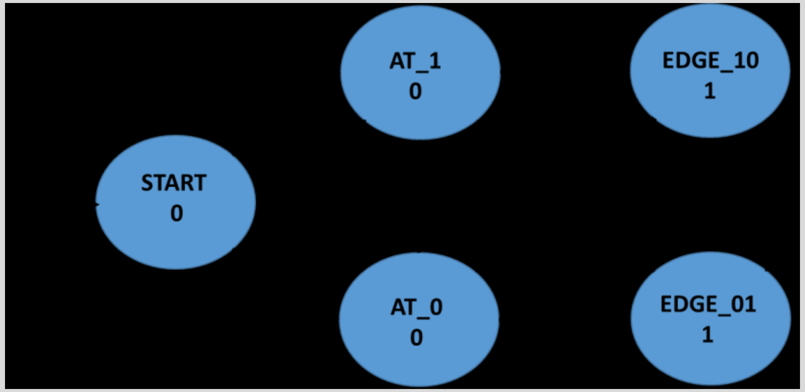
- Now let’s implement the **next state and output logic**

```
module edge_detect (input in, input clock, input reset, output reg out);
// sequential code from previous slide
always @ (current_state, in) begin //always @ (*)
  out = 0;
  next_state = current_state;
  case (current_state)
    START: if (in) next_state = AT_1;
           else next_state = AT_0;
    AT_1:  if (~in) next_state = EDGE_10;
    AT_0:  if (in) next_state = EDGE_01;
    EDGE_10: begin
               out = 1;
               if (in) next_state = EDGE_01;
                else next_state = AT_0;
             end
    EDGE_01: //DO THIS
    default: begin out = 1’bX; next_state = 3’bX; end

  endcase
end
```

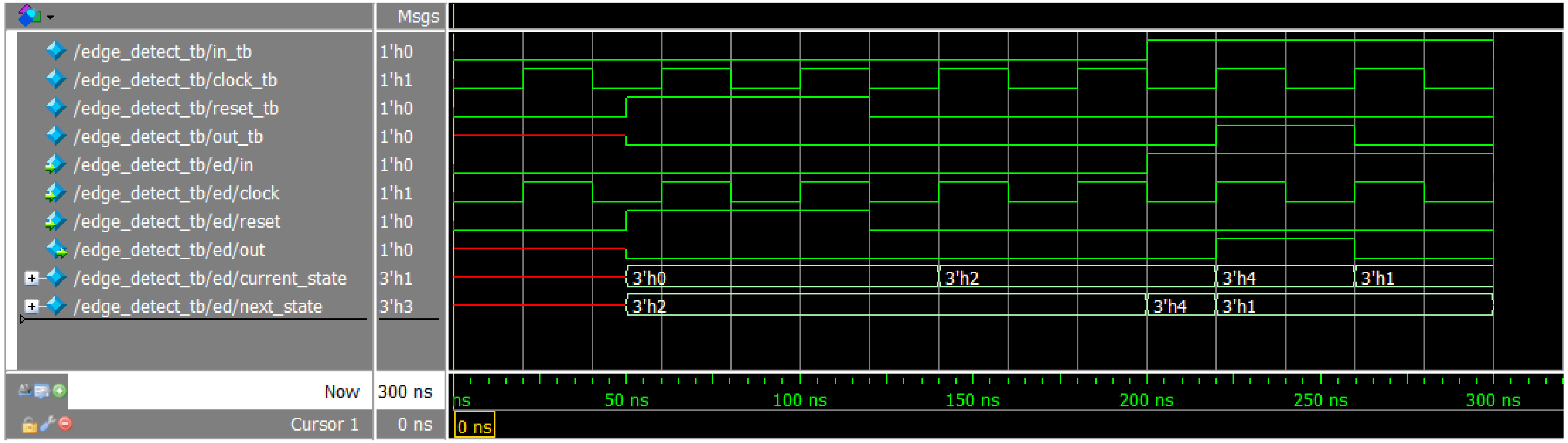
Similar to C case statement

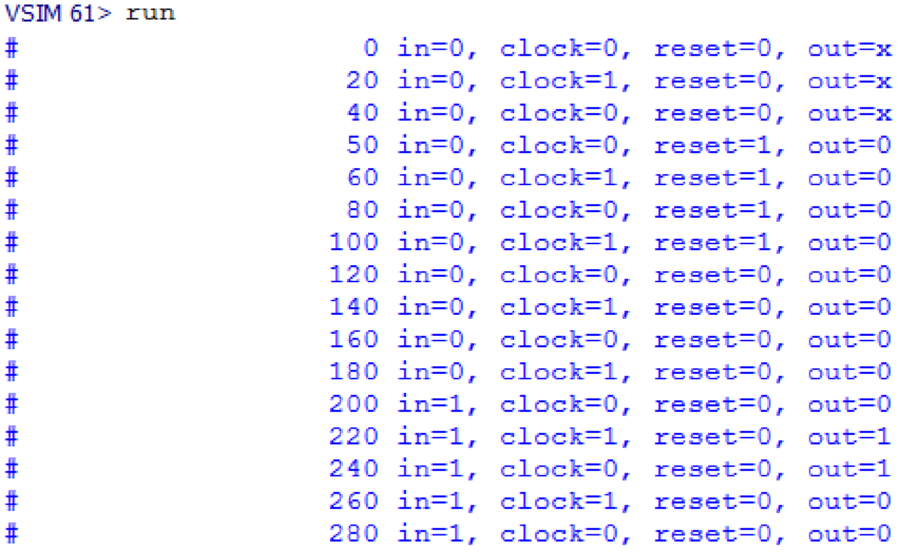
Current_state = 5,6,7

X: don’t care



Simulation Output

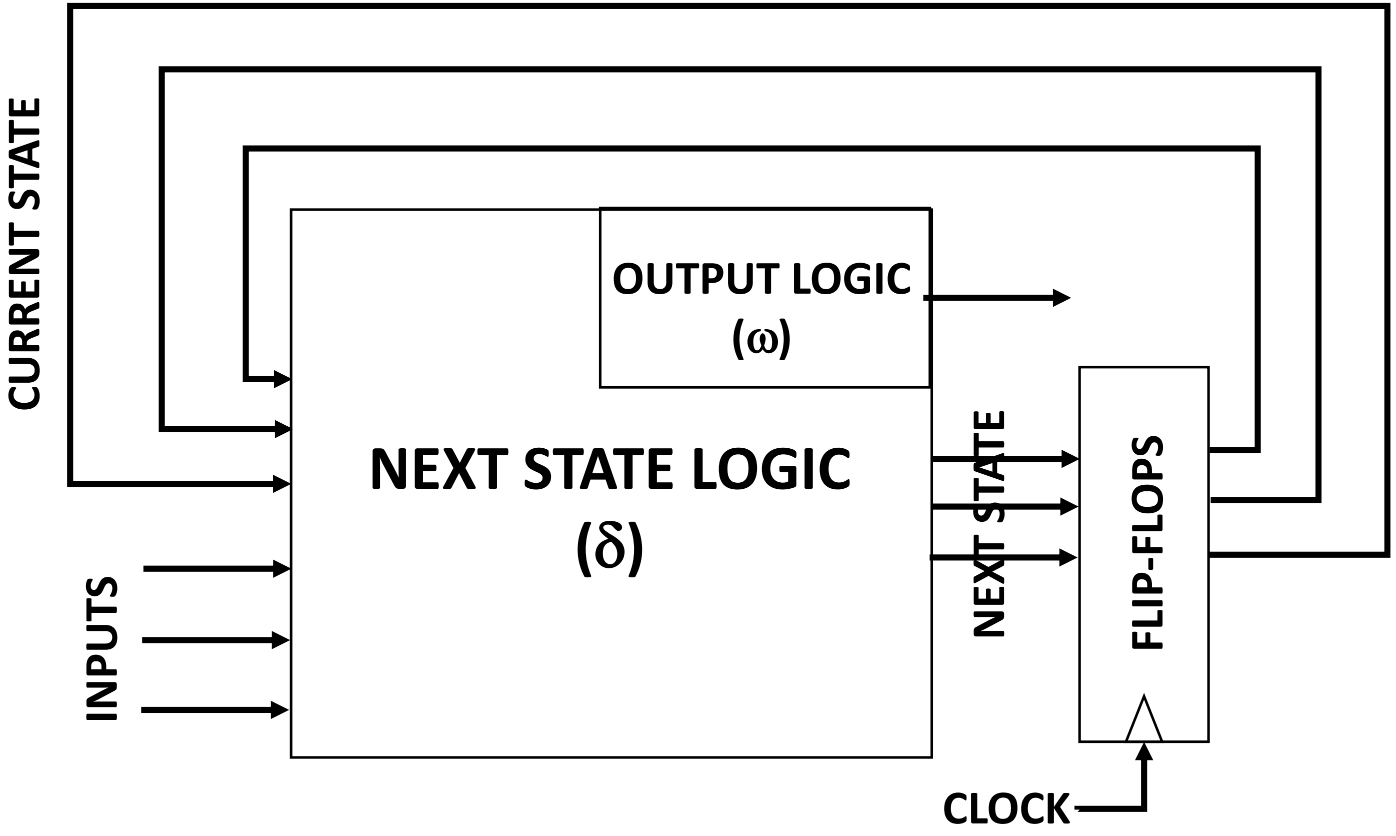

Msgs
/edge_detect_tb/in_tb 1'h0
/edge_detect_tb/clock_tb 1'h1
/edge_detect_tb/reset_tb 1'h0
/edge_detect_tb/out_tb 1'h0
/edge_detect_tb/ed/in 1'h0
/edge_detect_tb/ed/clock 1'h1
/edge_detect_tb/ed/reset 1'h0
/edge_detect_tb/ed/out 1'h0
/edge_detect_tb/ed/current_state 3'h1
/edge_detect_tb/ed/next_state 3'h3
3'h0
3'h2
3'h4
3'h1
3'h2
3'h4
3'h1
Now 300 ns
Cursor 1 0 ns
ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
0 ns



```
VSIM 61> run
#                    0 in=0, clock=0, reset=0, out=x
#                   20 in=0, clock=1, reset=0, out=x
#                   40 in=0, clock=0, reset=0, out=x
#                   50 in=0, clock=0, reset=1, out=0
#                   60 in=0, clock=1, reset=1, out=0
#                   80 in=0, clock=0, reset=1, out=0
#                  100 in=0, clock=1, reset=1, out=0
#                  120 in=0, clock=0, reset=0, out=0
#                  140 in=0, clock=1, reset=0, out=0
#                  160 in=0, clock=0, reset=0, out=0
#                  180 in=0, clock=1, reset=0, out=0
#                  200 in=1, clock=0, reset=0, out=0
#                  220 in=1, clock=1, reset=0, out=1
#                  240 in=1, clock=0, reset=0, out=1
#                  260 in=1, clock=1, reset=0, out=0
#                  280 in=1, clock=0, reset=0, out=0
```

Mealy Machine

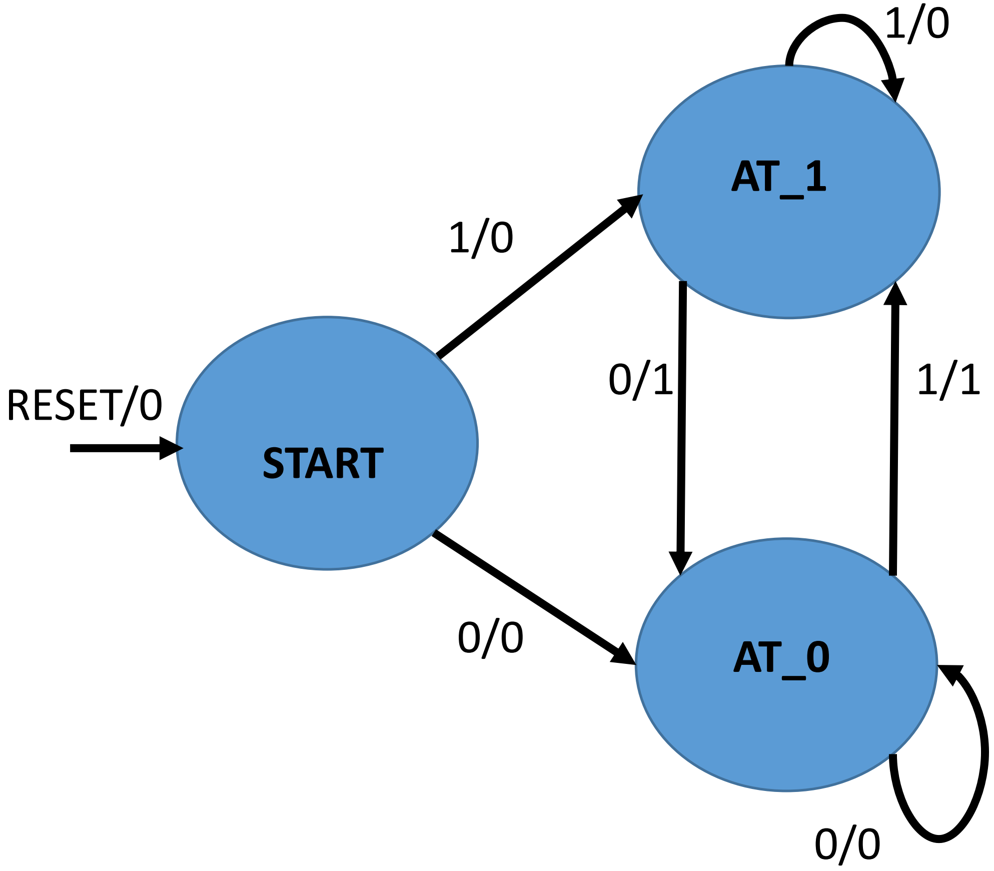
- Mealy machine: output depends on state *and* input



Direct path between inputs and outputs that does not pass through a flip-flop

- Pros:
  - Fewer states
  - Output more “responsive” to input
- Cons:
  - Unstable outputs
  - Cascading Mealy machines?

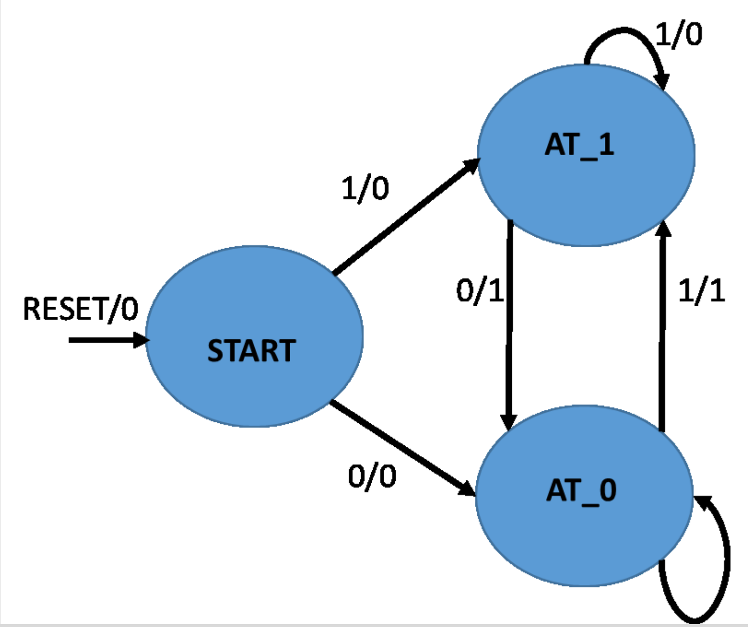
Mealy Machine: Verilog Implementation



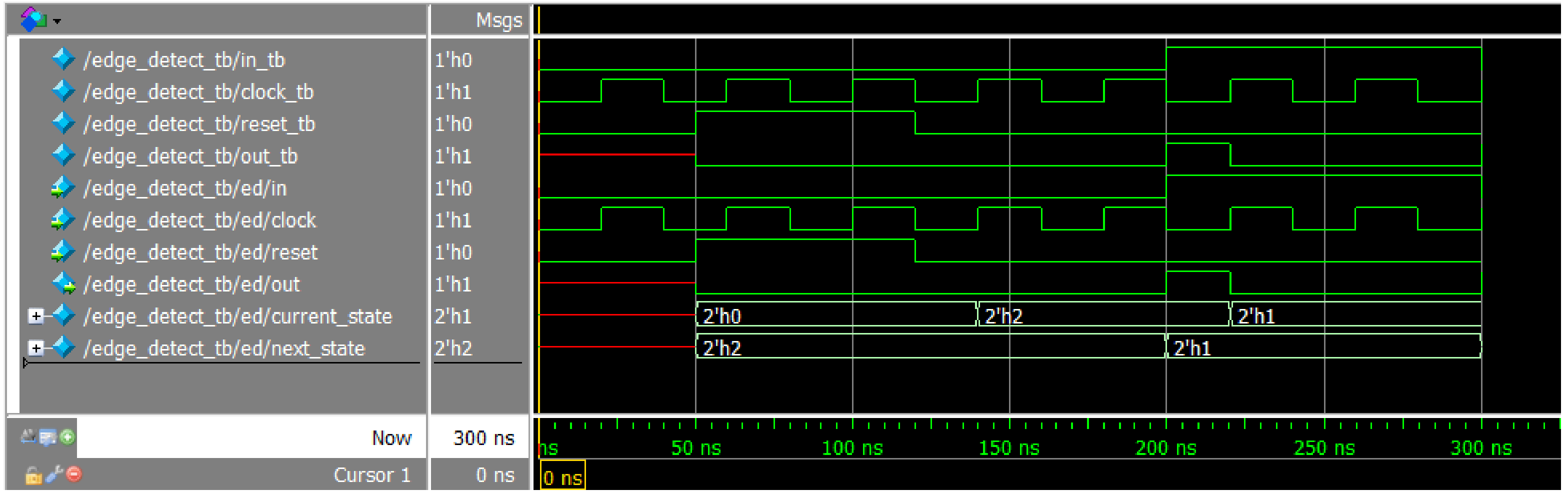
```
module edge_detect ( input in, input clock,
                     input reset, output reg
out);


reg [1:0] current_state, next_state;

localparam START = 0, AT_1 = 1, AT_0 = 2;

always @ (posedge clock, posedge reset) begin
  if (reset)
     current_state <= START;
  else
     current_state <= next_state;
end
.........
endmodule
```

Mealy Machine: Verilog Implementation

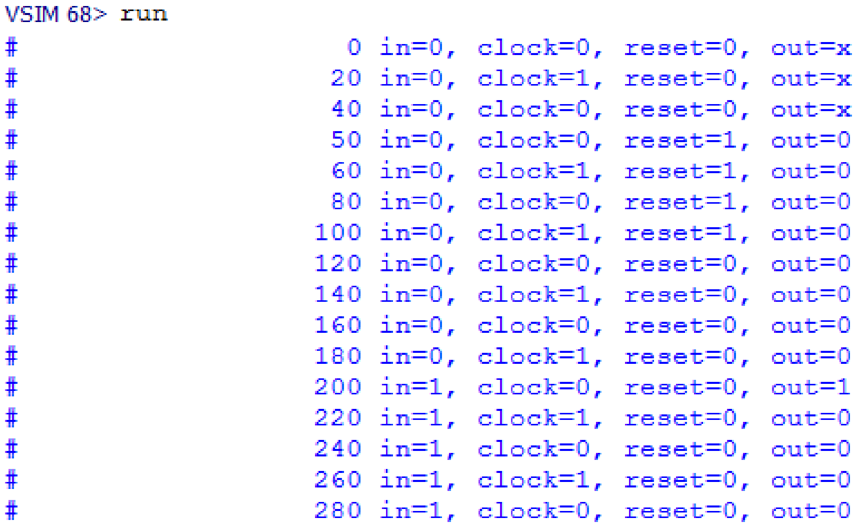
```
module edge_detect (input in, input clock, input reset, output reg out);
// sequential code from previous slide
always @ (current_state, in) begin //always @ (*)
  out = 0;
  next_state = current_state;
  case (current_state)
    START: if (in) next_state = AT_1;
           else    next_state = AT_0;
    AT_1:  begin
             if (~in) begin
                out = 1;
                next_state = AT_0;
             end
           end
   AT_0:         //DO THIS

   default:
endcase
end
endmodule
```



Simulation Output

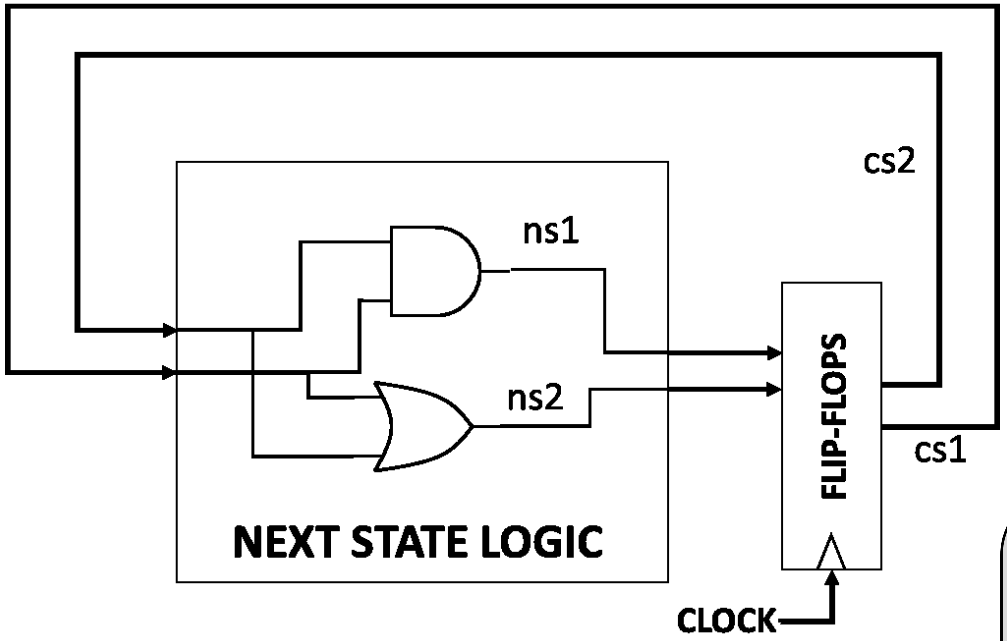


```
                                    Msgs
/edge_detect_tb/in_tb               1'h0
/edge_detect_tb/clock_tb            1'h1
/edge_detect_tb/reset_tb            1'h0
/edge_detect_tb/out_tb              1'h1
/edge_detect_tb/ed/in               1'h0
/edge_detect_tb/ed/clock            1'h1
/edge_detect_tb/ed/reset            1'h0
/edge_detect_tb/ed/out              1'h1
/edge_detect_tb/ed/current_state    2'h1
/edge_detect_tb/ed/next_state       2'h2
                           Now      300 ns
                      Cursor 1      0 ns
```



```
VSIM 68> run
#                    0 in=0, clock=0, reset=0, out=x
#                   20 in=0, clock=1, reset=0, out=x
#                   40 in=0, clock=0, reset=0, out=x
#                   50 in=0, clock=0, reset=1, out=0
#                   60 in=0, clock=1, reset=1, out=0
#                   80 in=0, clock=0, reset=1, out=0
#                  100 in=0, clock=1, reset=1, out=0
#                  120 in=0, clock=0, reset=0, out=0
#                  140 in=0, clock=1, reset=0, out=0
#                  160 in=0, clock=0, reset=0, out=0
#                  180 in=0, clock=1, reset=0, out=0
#                  200 in=1, clock=0, reset=0, out=1
#                  220 in=1, clock=1, reset=0, out=0
#                  240 in=1, clock=0, reset=0, out=0
#                  260 in=1, clock=1, reset=0, out=0
#                  280 in=1, clock=0, reset=0, out=0
```

Blocking Vs. Non-blocking Assignments



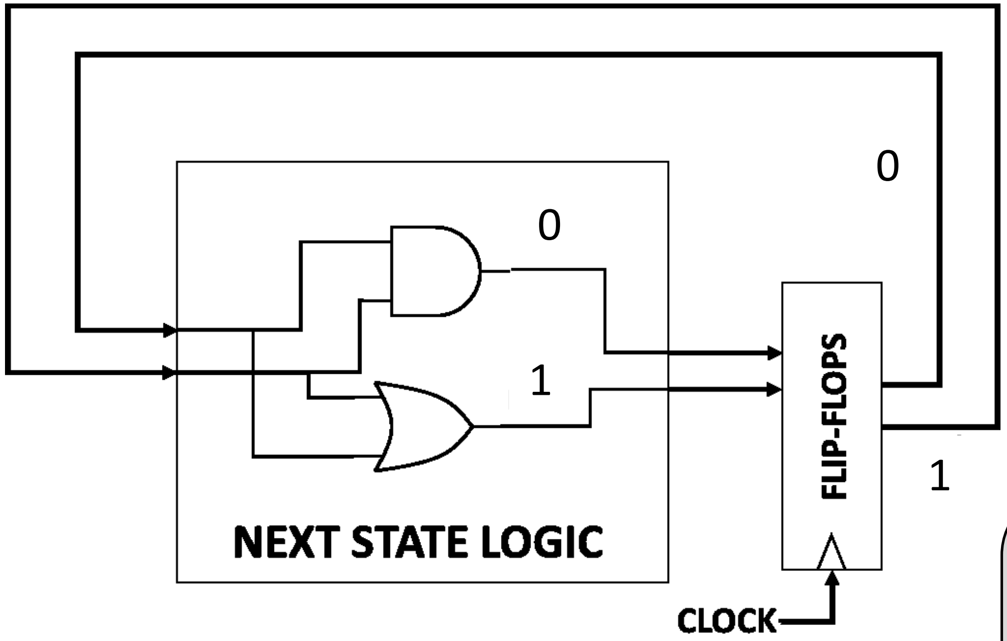
```
// combinational part
always @(cs1, cs2) begin
    ns1 = cs & cs2;
    ns2 = cs1 | cs2;
end
//sequential part
always @ (posedge clock) begin
    cs1 <= ns1;
    cs2 <= ns2;
end
```

OR

```
// sequential + combinational part
always @(posedge clcok) begin
    cs1 <= cs & cs2;
    cs2 <= cs1 | cs2
end
```

Both LHSs evaluated first; then assigned to RHSs. As if the two ops happened in parallel

Blocking Vs. Non-blocking Assignments



```
// combinational and sequential part
always @(posedge clock) begin
     cs1 = cs & cs2;
     cs2 = cs1 | cs2;
end
// Blocking assignments operate in sequence.
// So first cs1 will become 0 (1&0).
// Now both cs1 and cs2 are 0.
// The next statement executes, and cs2 remains
zero.
```

DOES NOT WORK

OR

```
// sequential + combinational part
always @(posedge clcok) begin
     cs1 <= cs & cs2;
     cs2 <= cs1 | cs2
end
```

Will operate correctly regardless of order in which these statement are written!!

Thank You

Module Hierarchy

- 4-bit adder using full adders

```
module adder ( input a, input b, input cin, output sum, output cout);

wire w1, w2, w3, w4, w5;

and a1 (w1, a, b),
    a2 (w2, b, cin),
    a3 (w3, a, cin);

or  o1 (w4, w1, w2),
    o2 (cout, w4, w3);

xor x1 (w5, a, b),
    x2 (sum, cin, w5);

endmodule
```

Blocking vs. non-blocking assignment

- Assume we want to swap the contents of registers a and b

```
module swap;

reg a, b;

initial begin
   a = 0;
   b = 1;

➡  #1 a = b;//a=1
➡  b = a;//b=1 ☹

end

endmodule
```

Blocking assignments operate in sequence

```
module swap;

reg a, b, temp;

initial begin
   a = 0;
   b = 1;

   #1 temp = a; //temp=0
   a = b; //a=1
   b = temp; //b=0 ☺

end

endmodule
```

```
module swap;

reg a, b;

initial begin
   a <= 0;
   b <= 1;

   #1 a <= b; //a=1
   b <= a; //b=0 ☺

end

endmodule
```

Works!!

Verilog Basics:

- Verilog provides several ways in which the full adder module can be specified
  - Structural: gate level structure
  - Behavioral: Boolean logic equations that describe functionality
  - Procedural: a step-by