Logic Design in Flipped Classroom

CH1 Fundamentals of Hardware Description Language

Lecturer: ENDpj(周敬堯)

Outline

- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Simulation & Verification

Hardware Description Language

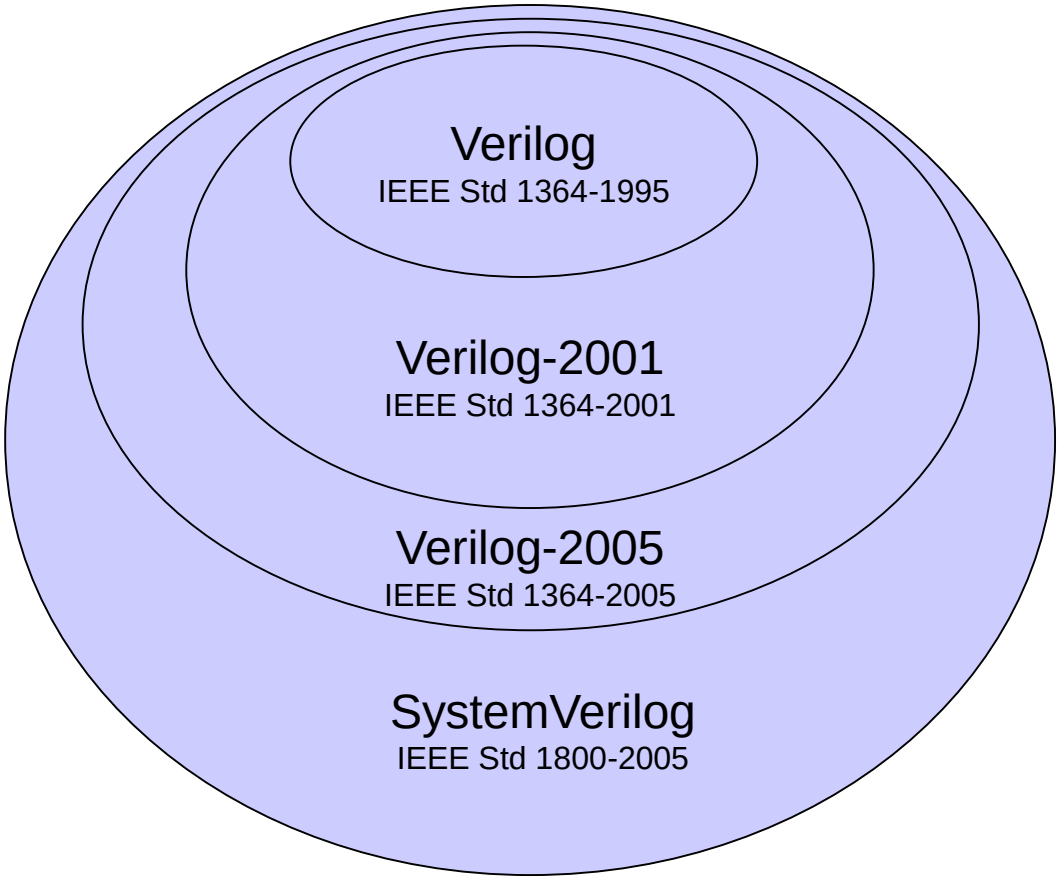
From Wikipedia

- **H**ardware **D**escription **L**anguage (HDL) is any language from a class of computer languages and/or programming languages for formal description of electronic circuits, and more specifically, digital logic.
- HDL can
  - Describe the circuit's operation, design, organization
  - Verify its operation by means of simulation.
- Now HDLs usually merge Hardware Verification Language, which is used to verify the described circuits.
- Supporting discrete-event (digital) or continuous-time (analog) modeling, e.g.:
  - SPICE, Verilog HDL, VHDL, SystemC

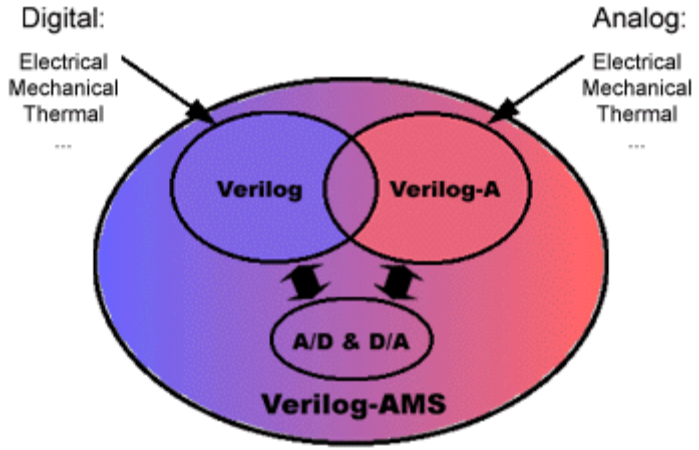
High-Level Programming Language

- It's possible to describe a hardware (operation, structure, timing, and testing methods) in C/C++/Java, why do we use HDL?
- The efficiency (to model/verify) does matter.
  - Native support to concurrency
  - Native support to the simulation of the progress of time
  - Native support to simulate the model of system
- The required level of detail determines the language we use.

History/Branch of Verilog



Digital-signal HDL



Mixed-signal/Multi-domain Hardware Discription Language

Outline

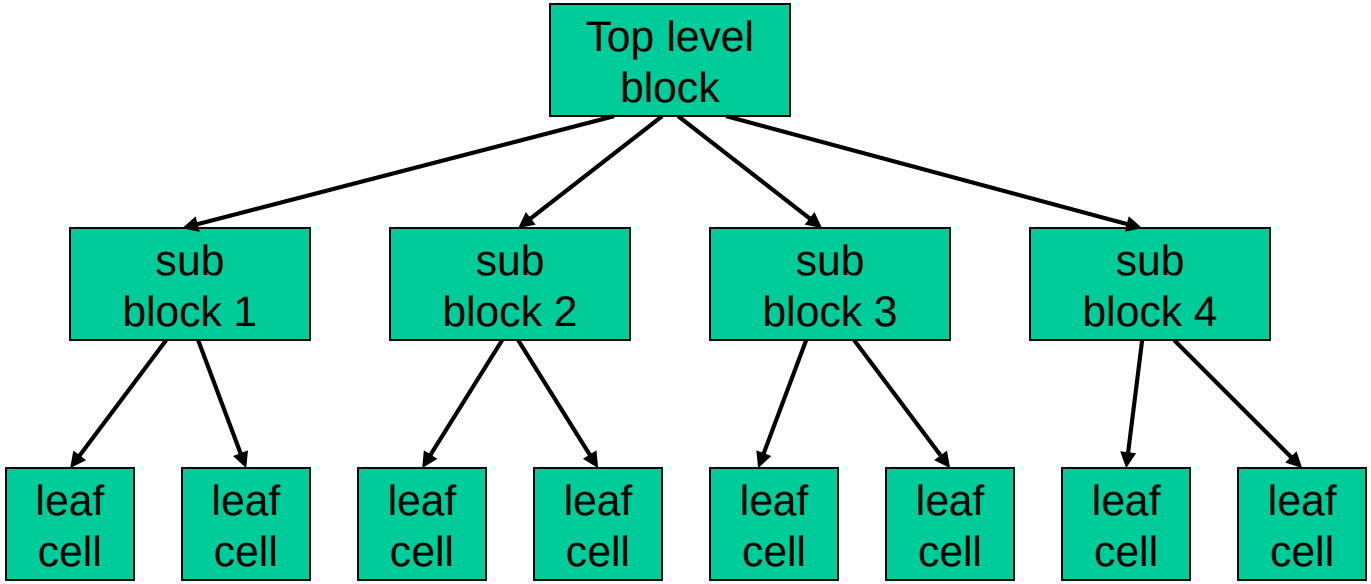
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification

Hierarchical Modeling Concept

- Introduce *top-down* and *bottom-up* design methodologies
- Introduce *module* concept and encapsulation for hierarchical modeling
- Explain differences between modules and module instances in Verilog

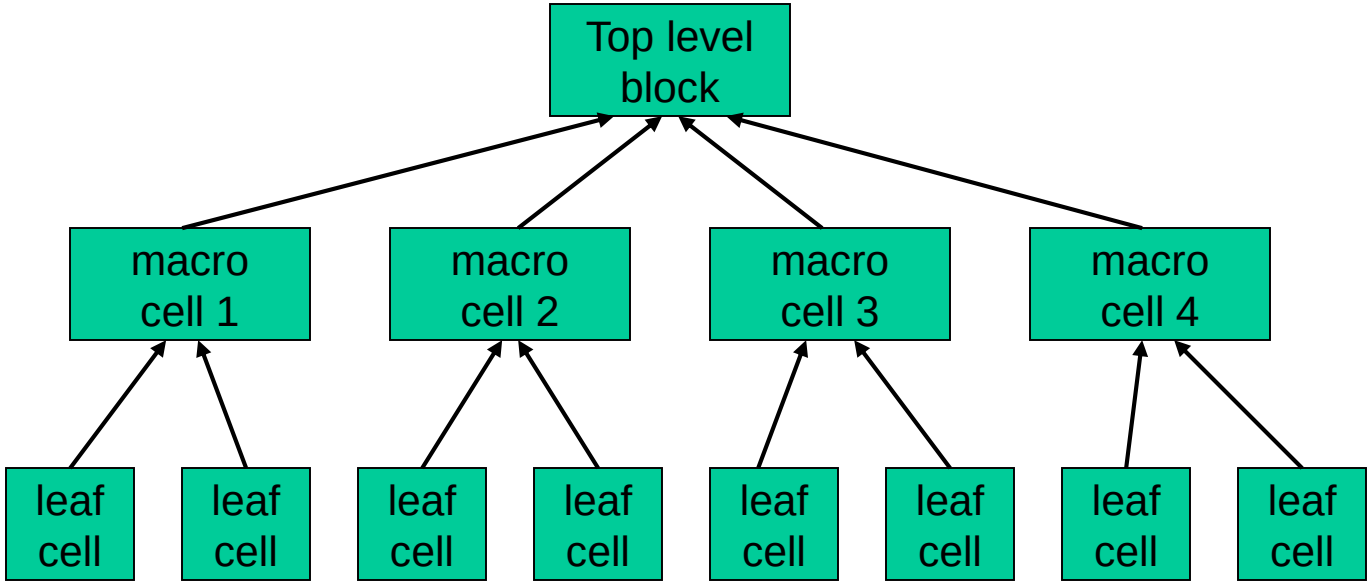
Top-down Design Methodology

- We define the top-level block and identify the sub-blocks necessary to build the top-level block.
- We further subdivide the sub-blocks until we come to leaf cells, which are the cells that cannot further be divided.

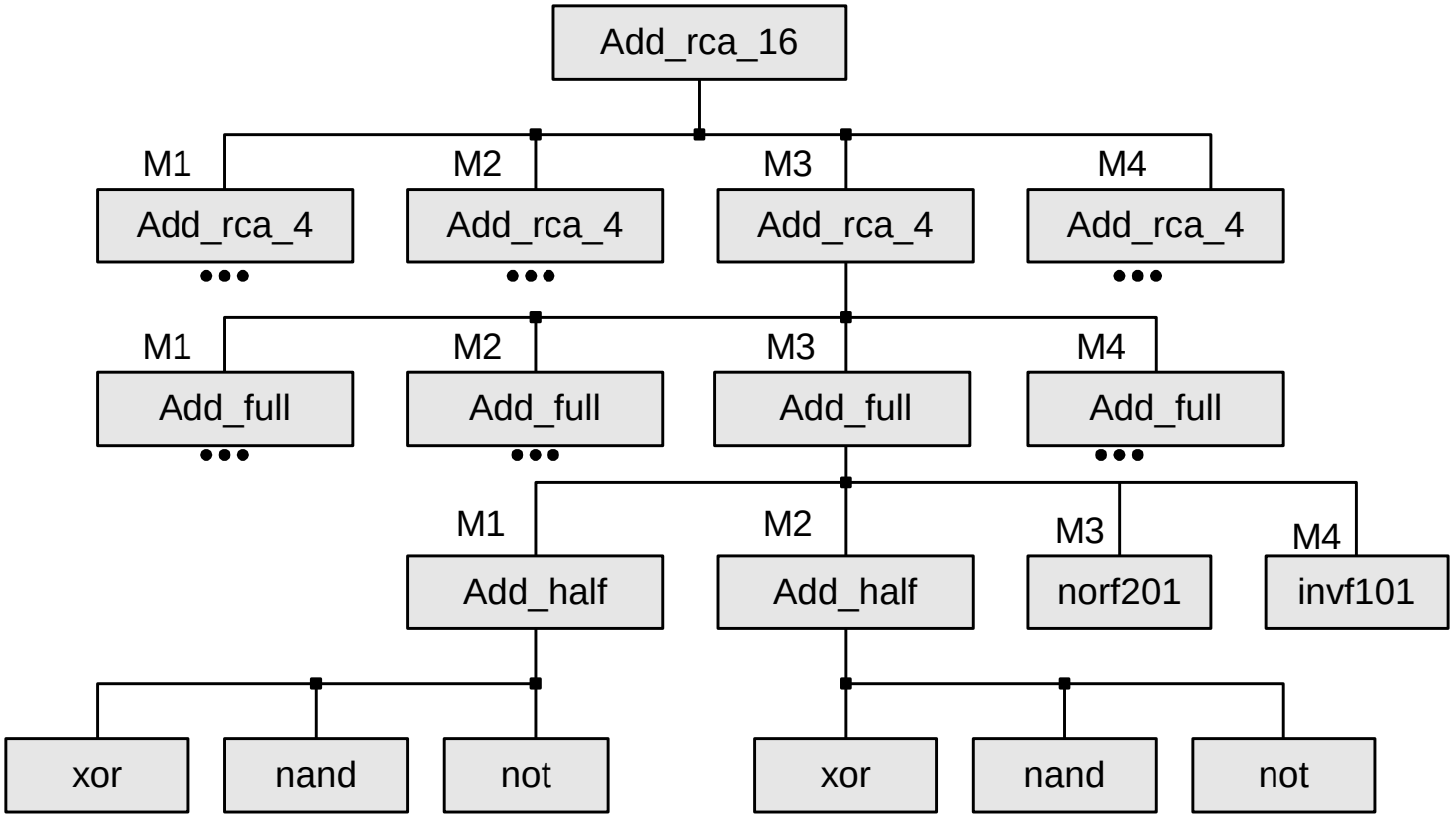


Bottom-up Design Methodology

- We first identify the building block that are available to us.
- We build bigger cells, using these building blocks.
- These cells are then used for higher-level blocks until we build the top-level block in the design.

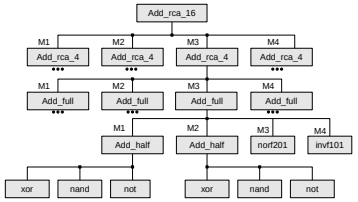


Example: 16-bit Adder



Hierarchical Modeling in Verilog

- A Verilog design consists of a hierarchy of modules.
- Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports.
- Internally, a module can contain any combination of the following
  - net/variable declarations (wire, reg, integer, etc.)
  - concurrent and sequential statement blocks
  - instances of other modules (sub-hierarchies).



Module

- Basic building block in Verilog.
- Module
  1. Created by “declaration” (can’t be nested)
  2. Used by “instantiation“

- Interface is defined by ports
- May contain instances of other modules
- All modules run concurrently

Design Encapsulation

- Encapsulate structural and functional details in a module

```
module <Module Name> (<PortName List>);

 // Structural part
      <List of Ports>
      <Lists of Nets and Registers>
      <SubModule List>  <SubModule Connections>

// Behavior part
      <Timing Control Statements>
            <Parameter/Value Assignments>
      <Stimuli>
      <System Task>
endmodule
```

- Encapsulation makes the model available for instantiation in other modules

Instances

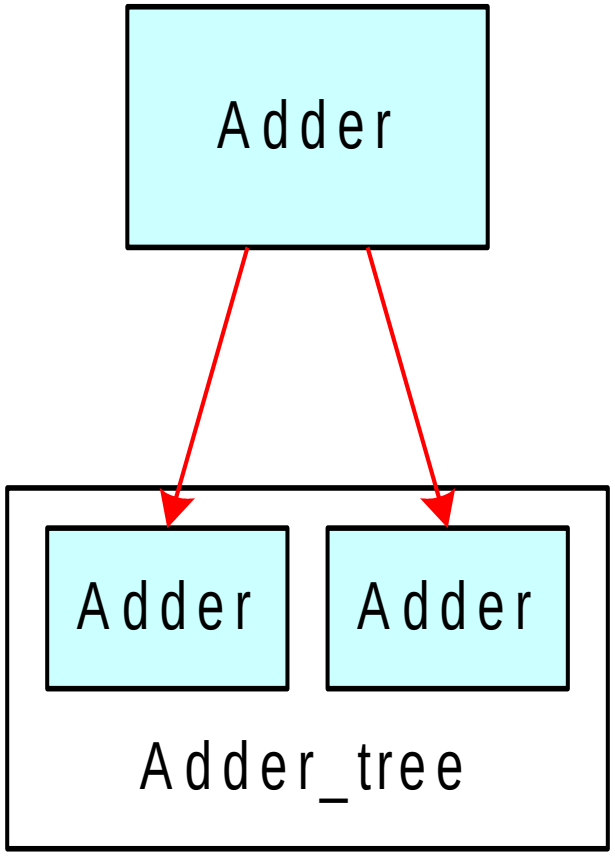
- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template.
- Each object has its own name, variables, parameters and I/O interface.

Module Instantiation

```
module adder(out,in1,in2);
    output  out;
    input   in1,in2,sel;

    assign  out=in1 + in2;
endmodule
```

instance example

```
module adder_tree (out0,out1,in1,in2,in3,in4);
    output  out0,out1;
    input   in1,in2,in3,in4;

    adder   add_0 (out0,in1,in2);
    adder   add_1 (out1,in3,in4);
endmodule
```



Analogy: module ↔ class

As ***module*** is to ***Verilog HDL***, so ***class*** is to ***C++ programming language***.

Syntax	module *m_Name*(IO list);
	...
	endmodule
		class *c_Name* {
		...
		};

Instantiation	*m_Name* **ins_name** (port connection list);	*c_Name* **obj_name**;
Member	**ins_name**.member_signal	**obj_name**.member_data
Hierachy	instance.sub_instance.member_signal	object.sub_object.member_data

Analogy: module ↔ class

```
class c_AND_gate {
 bool in_a;
 bool in_b;
 bool out;
 void evalutate() { out = in_a && in_b; }
};
```

Model AND gate with C++

```
module m_AND_gate ( in_a, in_b, out );
 input  in_a;
 input  in_b;
 output out;
 assign out = in_a & in_b;
endmodule
```

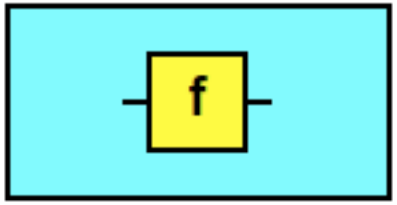
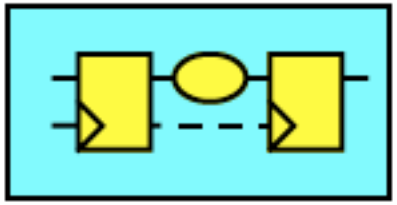
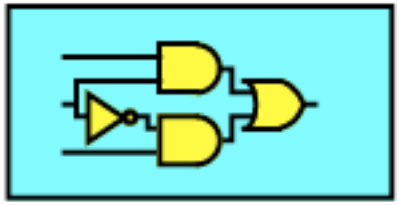
Model AND gate with Verilog HDL

- assign and evaluate() is simulated/called at each $T_{i+1} = T_i + t_{resolution}$

Outline

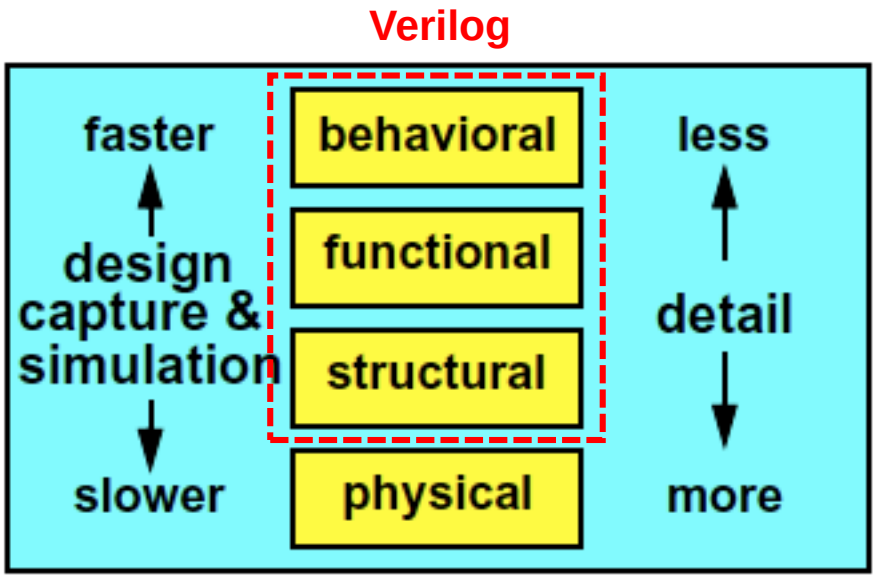
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay
- Simulation & Verification

Cell-Based Design and Levels of Modeling

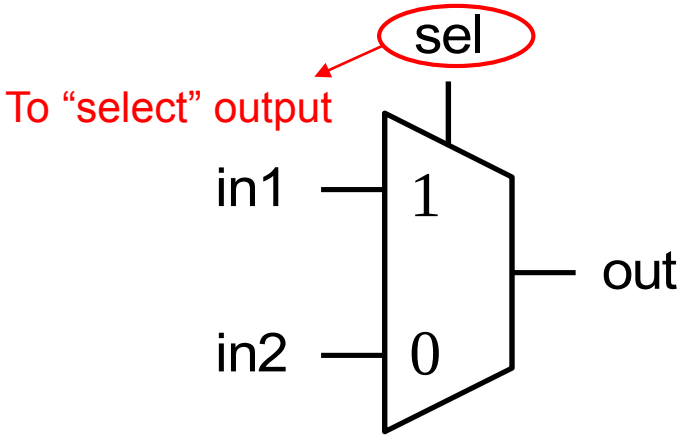
Behavioral Level		Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools
Register Transfer Level (RTL)		Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level
Structural/Gate Level		Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library
Transistor/Physical Level		Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE

Tradeoffs Among Modeling Levels

- Each level of modeling permits modeling at a higher or lower level of detail. More detail means more efforts for designers and the simulator.



An Example - 1-bit Multiplexer in Behavioral Level




```
if (sel==1)
  out = in1;
else
  out = in2;
```

```
module mux2(out,in1,in2,sel);
    output out;
    input in1,in2,sel;
    reg out;

    always@(in1 or in2 or sel)
    begin
        if(sel) out=in1;
        else    out=in2;
    end
endmodule
```

always block

Behavior: Event-driven behavior description construct

An Example - 1-bit Multiplexer in RTL Level

sel	in1	in2	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

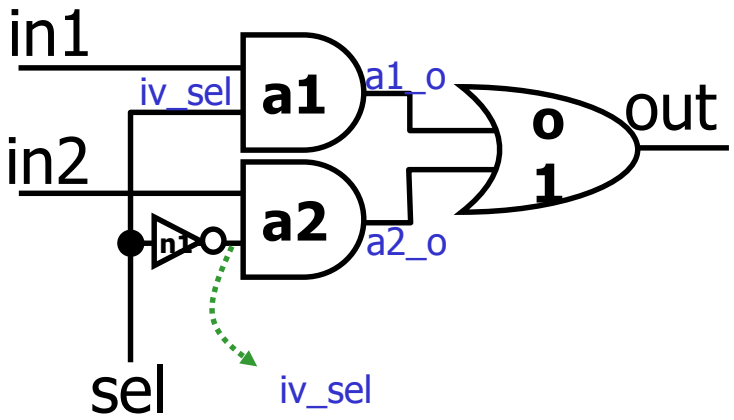
Continuous assignment

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    assign  out=sel?in1:in2;
endmodule
```

out = sel? In1 : in2

RTL: describe logic/arithmetic function between input node and output node

An Example - 1-bit Multiplexer in Gate Level



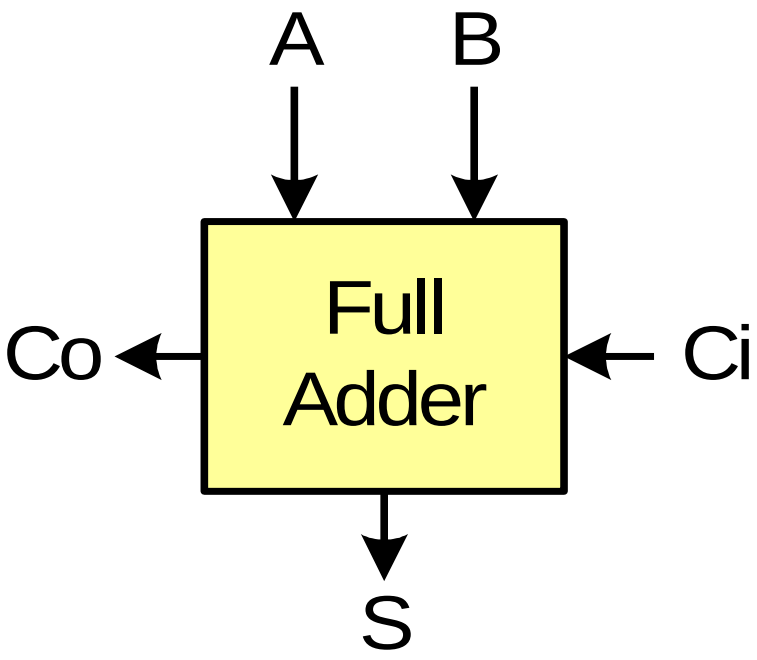

```
module mux2(out,in1,in2,sel);
    output  out;
    input   in1,in2,sel;

    and a1(a1_o,in1,sel);
    not n1(iv_sel,sel);
    and a2(a2_o,in2,iv_sel);
    or  o1(out,a1_o,a2_o);
endmodule
```

Gate Level: you see only netlist (gates and wires) in the code

Gate Level Modeling

- Steps
  - Develop the Boolean function of output
  - Draw the circuit with logic gates/primitives
  - Connect gates/primitives with net (usually wire)

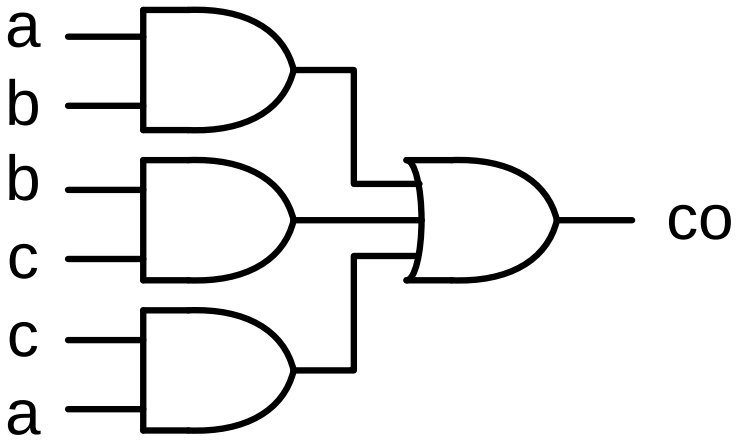
Case Study
1-bit Full Adder



Ci	A	B	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Case Study
1-bit Full Adder

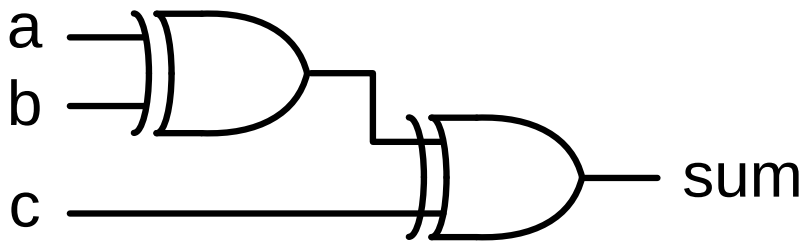
- co = (a • b) + (b • ci) + (ci • a);

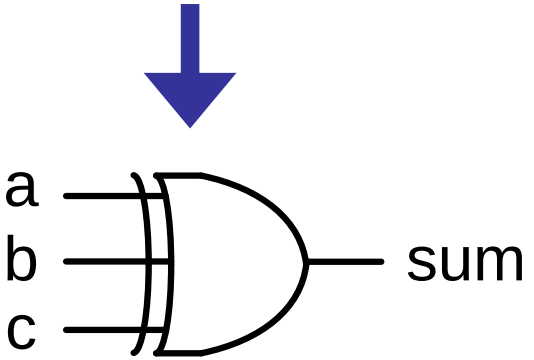



```
30
31 module FA_co ( co, a, b, ci );
32
33   input   a, b, ci;
34   output  co;
35   wire    ab, bc, ca;
36
37   and g0( ab, a, b );
38   and g1( bc, b, c );
39   and g2( ca, c, a );
40   or  g3( co, ab, bc, ca );
41
42 endmodule
43
```

Case Study
1-bit Full Adder

- sum = a ⊕ b ⊕ ci



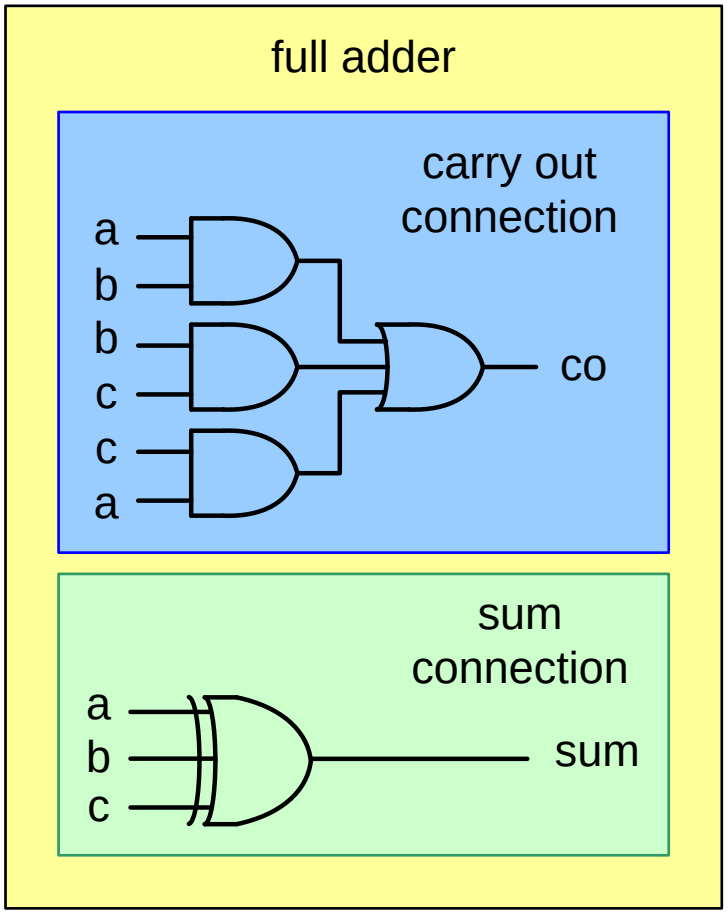



```
44 module FA_sum ( sum, a, b, ci );
45
46   input   a, b, ci;
47   output  sum, co;
48
49   xor g1( sum, a, b, ci );
50
51 endmodule
52
```

Case Study
1-bit Full Adder

- Full Adder Connection
  - Instance *ins_c* from FA_co
  - Instance *ins_s* from FA_sum

```
20
21 module FA_gatelevel( sum, co, a, b, ci);
22
23   input    a, b, ci;
24   output   sum, co;
25
26   FA_co    ins_c( co, a, b, ci );
27   FA_sum   ins_s( sum, a, b, ci );
28
29 endmodule
30
```

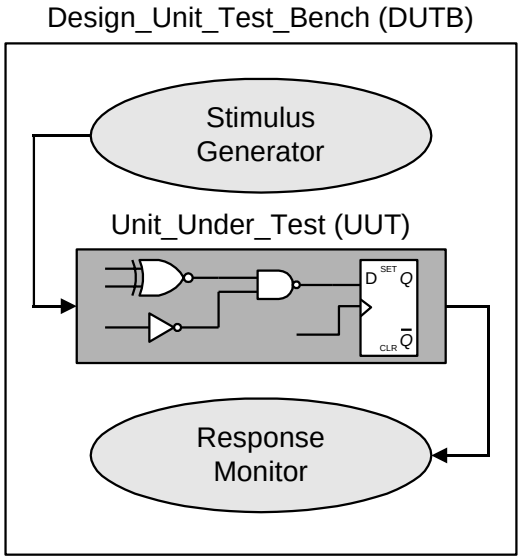


Outline

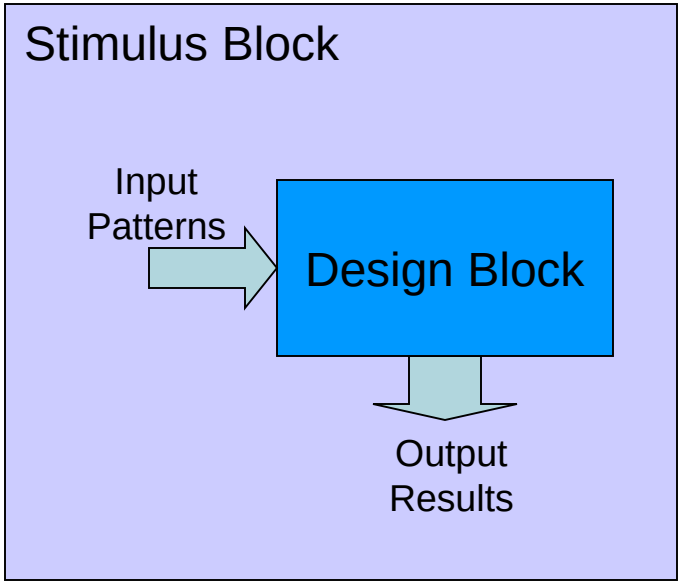
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Simulation & Verification

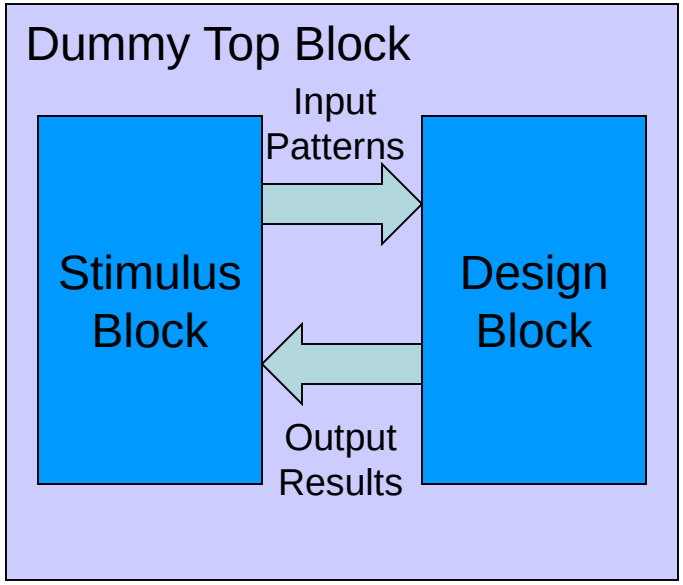
Verification Methodology

- Task: systematically verify the functionality of a model.
- Approaches: Simulation and/or formal verification
- Simulation:
  (1) detect syntax violations in source code
  (2) simulate behavior
  (3) monitor results



Components of a Simulation

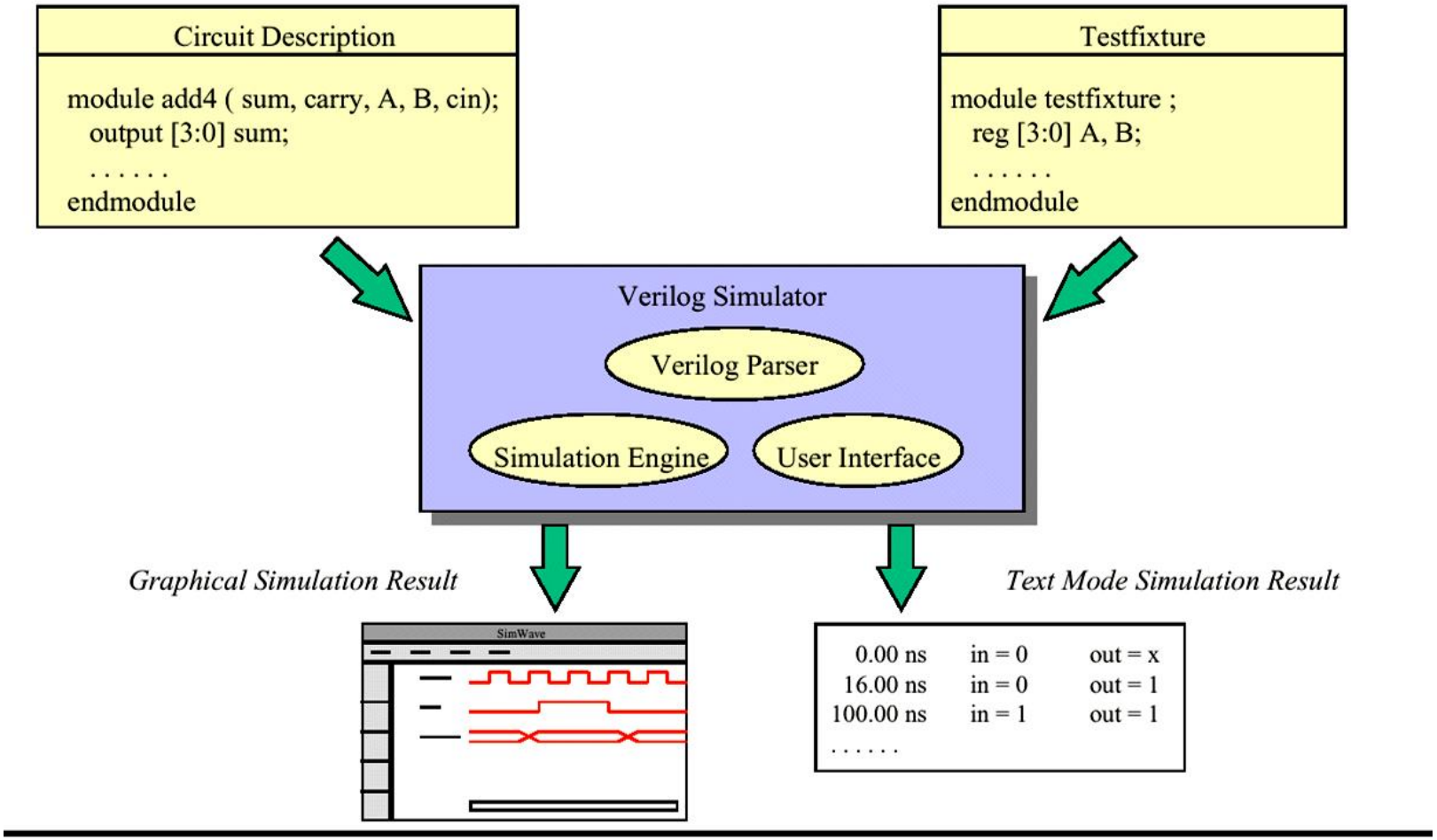




The output results are verified by console/waveform viewer

The output results are verified by testbench or stimulus block

Verilog Simulator



Testbench Template

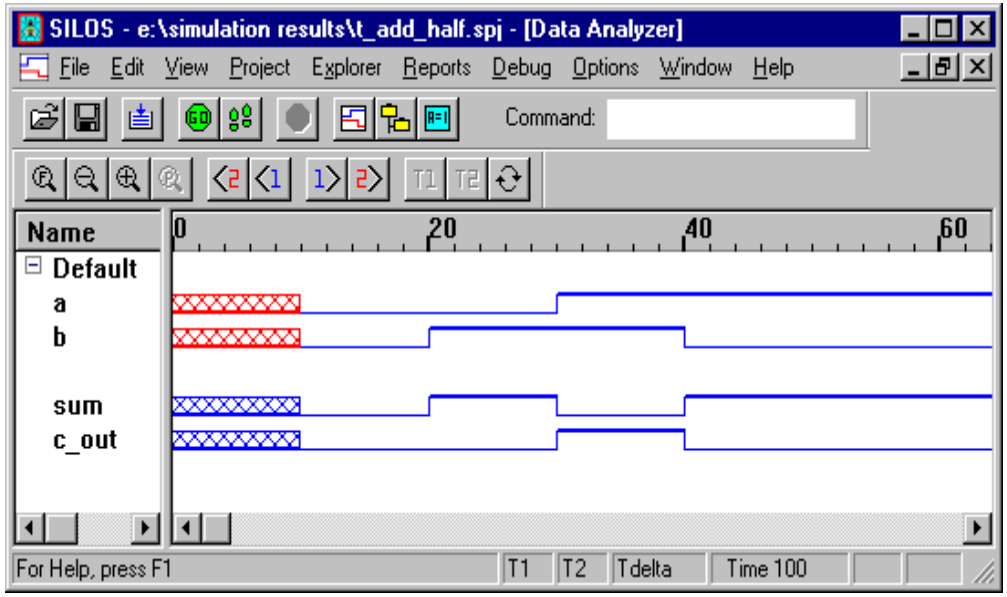
- Consider the following template as a guide for simple testbenches:

```
module t_DUTB_name ();    // substitute the name of the UUT
  reg …;                  // Declaration of register variables for primary inputs of the UUT
  wire …;                 // Declaration of primary outputs of the UUT
  parameter       time_out = // Provide a value

  UUT_name M1_instance_name ( UUT ports go here);

  initial $monitor (  );  // Specification of signals to be monitored and displayed as text

  initial #time_out $stop;  // (Also $finish) Stopwatch to assure termination of simulation

  initial                 // Develop one or more behaviors for pattern generation and/or
                          // error detection
    begin
                          // Behavioral statements generating waveforms
                          // to the input ports, and comments documenting
                          //  the test.  Use the full repertoire of behavioral
                          // constructs for loops and conditionals.

    end
endmodule
```

Example: Testbench

```
module t_Add_half();
  wire          sum, c_out;
  reg           a, b;                    // Variable for stimulus waveforms

  Add_half_0_delay M1 (sum, c_out, a, b);  //UUT

   initial begin                         // Time Out
    #100 $finish;                        // Stopwatch
    end

  initial begin                          // Stimulus patterns
    #10 a = 0; b = 0;                    // Statements execute in sequence
    #10 b = 1;
    #10 a = 1;
    #10 b = 0;
  end
endmodule
```

Simulation Results



MODELING TIP

A Verilog simulator assigns an *initial* value of **x** to all variables.

Propagation Delay

- Gate propagation delay specifies the time between an input change and the resulting output change
- Transport delay describes the time-of-flight of a signal transition
- Verilog uses an inertial delay model for gates and transport delay for nets

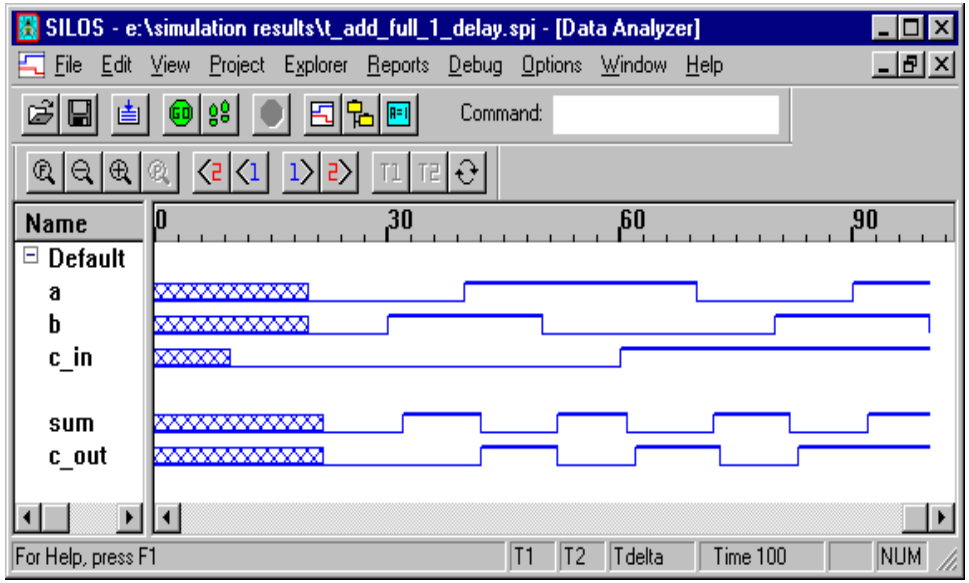
MODELING TIP

All primitives and nets have a default propagation delay of 0.

Example: Propagation Delay

- Unit-delay simulation reveals the chain of events

```
module Add_full (sum, c_out, a, b, c_in);
  output          sum, c_out;
  input           a, b, c_in;
  wire            w1, w2, w3;

  Add_half          M1 (w1, w2, a, b);
  Add_half          M2 (sum, w3, w1, c_in);
  or              #1 M3 (c_out, w2, w3);
endmodule

module Add_half (sum, c_out, a, b);
  output          sum, c_out;
  input           a, b;

  xor             #1 M1 (sum, a, b);
  and             #1 M2 (c_out, a, b);
endmodule
```



Simulation Schemes

- There are 3 categories of simulation schemes
  - Time-based: Simulation on real time scale, used by SPICE simulators
  - Event-based: Simulation on events of signal transition, used by Verilog simulators. Note each event must occurs on discrete time specified by the testbench.
  - Cycle-based: Used by system/platform level verification, less used in cell-based IC designing.

Logic Design in Flipped Classroom

Chap.2-1 Logic Design at Register-Transfer Level

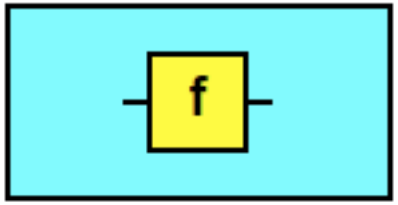
Lecturer: ENDpj(周敬堯)

Outline

- Definition of Register-Transfer Level (RTL)
- Verilog Syntax of RTL
  - Operands
  - Operators
- Continuous Assignments for Combinational Circuits

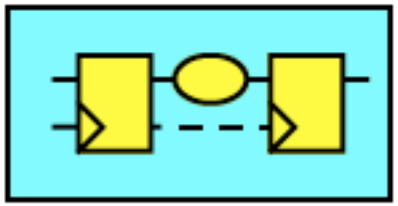
Brief Review of Modeling Levels

Behavioral Level



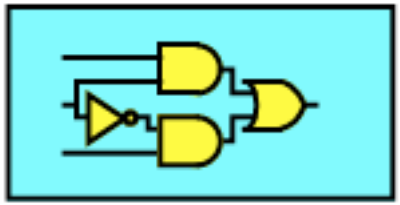
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

Structural/Gate Level



Usually generated by synthesis tool by using a front-end cell library, can be simulated by EDA tools. A gate is mapped to a cell in library

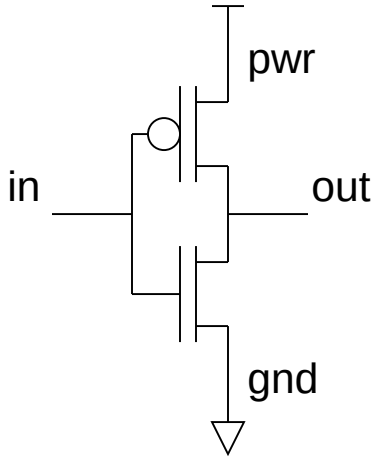
Transistor/Physical Level

Usually generated by synthesis tool by using a back-end cell library, can be simulated by SPICE

Various Abstraction of Verilog (1/4)

- Transistor level Verilog description of a inverter

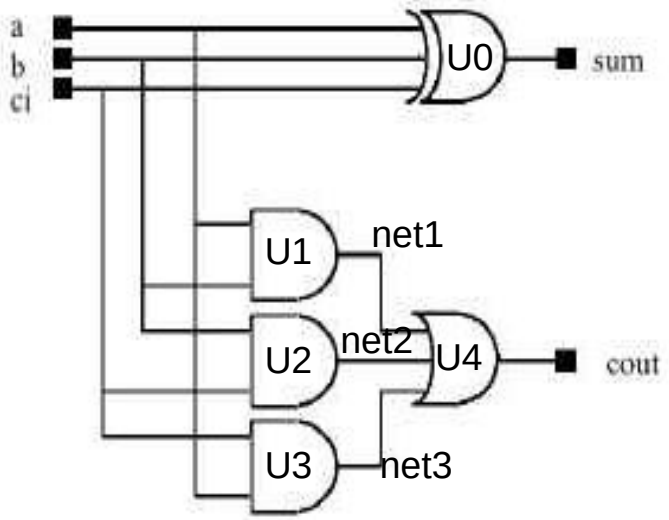
```
module inv(out, in);
// port declaration
  output  out;
  input    in;
// declare power and ground
  supply1  pwr;
  supply0  gnd;
// Switch level description
  pmos  S0(out, pwr, in);
  nmos  S1(out, gnd, in);
endmodule
```



Various Abstraction of Verilog (2/4)

- Gate level Verilog description of a full-adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  wire     net1, net2, net3;
// Netlist description
  xor  U0(sum,a,b,cin);
  and U1(net1, a, b);
  and U2(net2, b, cin);
  and U3(net3, cin, a);
  or    U4(cout, net1, net2, net3;
endmodule
```



Various Abstraction of Verilog (3/4)

- RT-Level (RTL) Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  wire     cout, sum;

// RTL description
  assign  sum = a^b^cin;
  assign  cout = (a&b)|(b&cin)|(cin&a);

endmodule
```

Whenever a or b or c changes its logic state, evaluate *sum* and *cout* by using the equation

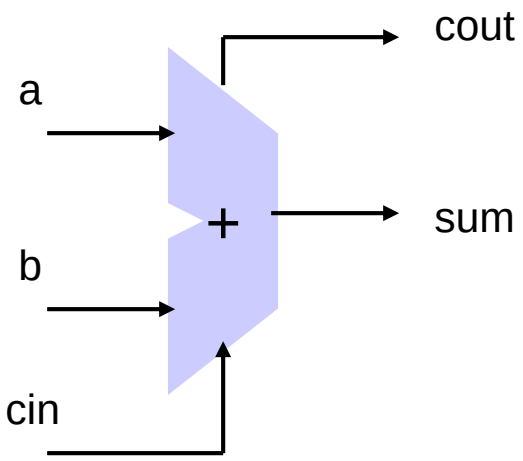
$sum = a \oplus b \oplus ci$

$cout = ab + bc + ca$

Various Abstraction of Verilog (4/4)

- Behavioral level Verilog description of a full adder

```
module fadder(cout, sum, a, b, cin);
// port declaration
  output  cout, sum;
  input    a, b, cin;
  reg       cout, sum;
// behavior description
  always @(a or b or cin)
    begin
       {cout,sum} = a + b + cin;
    end
endmodule
```



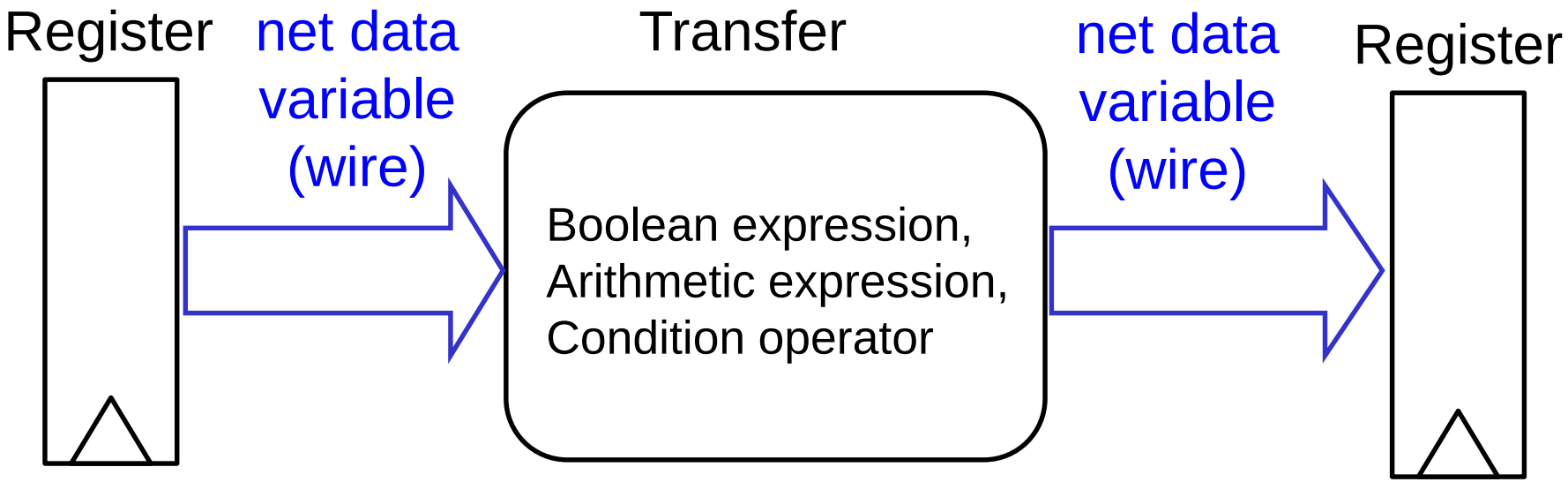
What is Register Transfer Level?

- In integrated circuit design, Register Transfer Level (RTL) description is a way of describing the operation of a synchronous digital circuit.

- In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between synchronous registers, and the logical operations performed on those signals.

Description at RT-Level

- Register (memory, usually D Flip-Flops)
- Transfer (operation, **combinational**)



Procedures to Design at RT-Level

- Design partition
  - Sequential circuit (register part)
  - Combinational circuit(transfer part)
- Net declaration
  - I/O ports
  - Net variable
- Transfer Circuit design
  - Logical operator
  - Arithmetical operator
  - Conditional operator

Example

```
wire [7:0] in1;
wire [7:0] in2;
wire [7:0] in3;              //input signals of transfer part

wire [7:0] out1;
wire [7:0] out2;
wire [7:0] out3;             //output signals of transfer part

assign out1 = in1 & in2;
assign out2 = in1 + in3;
assign out3 = in1[0]==1'b1 ? in2 : in3;    //transfer description
```

Outline

- Definition of Register-Transfer Level (RTL)
- Verilog Syntax of RTL
  - Operands
  - Operators
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits
- User-Defined Primitives

Operands

- Data types **`wire`** & **`reg`** are used for operands in RTL/Behavioral Level description
- Since all metal wires in circuits and only represent 0 or 1, we should consider all the variables in binary!
  - Unsigned: Ordinary binary
  - Signed: 2's compliment binary
  - Since there's only 0/1 for each bit, whether the variable is unsigned or signed is up to your recognition!
  - Simulators do all operations in binary (i.e. 0/1). If you check all the operations in binary, you'll never get wrong arithmetic/logic results!

Operators

Concatenation and replications	{,}
Negation	! , ~
Unary reduction	& , | , ^ , ^~ …
Arithmetic	+ , - , * , / , %
Shift	>> , <<
Relational	< , <= , > , >=
Equality	== , != , **===** , **!==**
Bitwise	~ , & , | , ^
Logical	&& , ||
Conditional	? :

Example (Arithmetic)

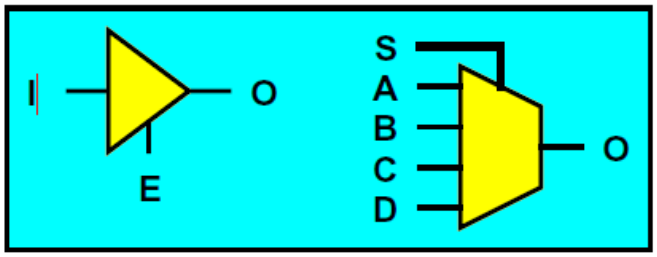
```
module DUT (sum,diff1,diff2,negA,A,B);
  output [4:0] sum,diff1,diff2,negA;
  input   [3:0] A , B

  assign sum=A+B;
  assign diff1=A-B;
  assign diff2=B-A;
  assign neg=-A;
endmodule
```

```
module test ();
  reg [3:0] A,B;
  wire [4:0]   sum,diff1,diff2,negA;
  DUT mydesign (.sum(sum) , .diff1(diff1) ,
        .diff2(diff2) , .negA(A) , .A(A) , .B(B));
 initial
  begin
    #5 A=5;B=2;
    $display("t_sim A B sum diff1 diff2 negA");
    $monitor($time,"%d%d%d%d%d%d",A,B,sum,
                diff1,diff2,neg);
    #10 $monitoroff;
    $monitor($time,"%b%b%b%b%b%b",A,B,sum,
                diff1,diff2,neg);
endmodule
```

```
t_sim A     B     sum   diff1 diff2  negA
5     5     2     7      3     29    27
15    0101  0010  00111 00011 11101 11011
```

Conditional Operators

- Conditional Operator
  - Usage: ***conditional_expression ? true_expression: false_expression;***




```
module driver(O,I,E);
output O; input I,E;
  assign O = E ? I : 'bz;
endmodule

module mux41(O,S,A,B,C,D);
output O;
input A,B,C,D; input [1:0] S;
  assign O = (S == 2'h0) ? A :
             (S == 2'h1) ? B :
             (S == 2'h2) ? C : D;
endmodule
```

Outline

- Definition of Register-Transfer Level (RTL)
- Verilog Syntax of RTL
  - Operands
  - Operators
- Continuous Assignments for Combinational Circuits
- Building Blocks of Sequential Circuits
- User-Defined Primitives

Assignments

- Continuous assignment

```
module holiday_1(sat, sun, weekend);
      input sat, sun; output weekend;
      assign weekend = sat | sun;      // outside a procedure
endmodule
```

- Procedural assignment

```
module holiday_2(sat, sun, weekend);
      input sat, sun; output weekend; reg weekend;
      always #1 weekend = sat | sun;      // inside a procedure
endmodule
```

```
module assignments
        // continuous assignments go here
always begin
        // procedural assignments go here
end
endmodule
```

Continuous Assignments (1/2)

- Convenient for logical or datapath specifications

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes

Continuous Assignments (1/2)

- Continuous assignments provide a way to model combinational logic

continuous assignment

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  assign out=~in;
endmodule
```

gate-level modeling

```
module inv_array(out,in);
  output [31:0] out;
  input [31:0] in;
  not U1(out[0],in[0]);
  not U2(out[1],in[1]);
  ..
  not U31(out[31],in[31]);
endmodule
```

Assignment Delays

- Regular Assignment Delay

  wire out;

  assign #10 out = in1 & in2; // delay in a continuous assign

- Implicit Continuous Assignment Delay

  wire #10 out = in1 & in2;

- Net Declaration Delay

  wire # 10 out; // net delays

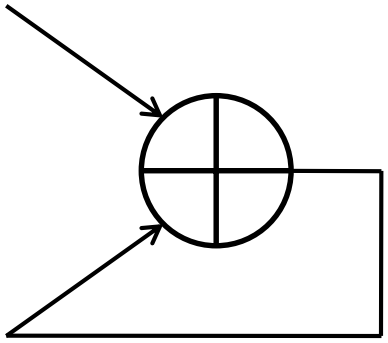
  assign out = in1 & in2;

- Remind again, non-synthesizable!
  For Testbench/Behavior model usage

Avoiding Combinational Loops

- Avoid combinational loops (or logic loops)
  - HDL Compiler and Design Compiler will automatically open up asynchronous combinational loops
  - Without disabling the combinational feedback loop, the static timing analyzer can't resolve
  - Example

```
wire [3:0] a;
wire [3:0] b;

assign a = b + a;
```



Logic Design in Flipped Classroom

Chap.2-2 Logic Design at Behavioral Level

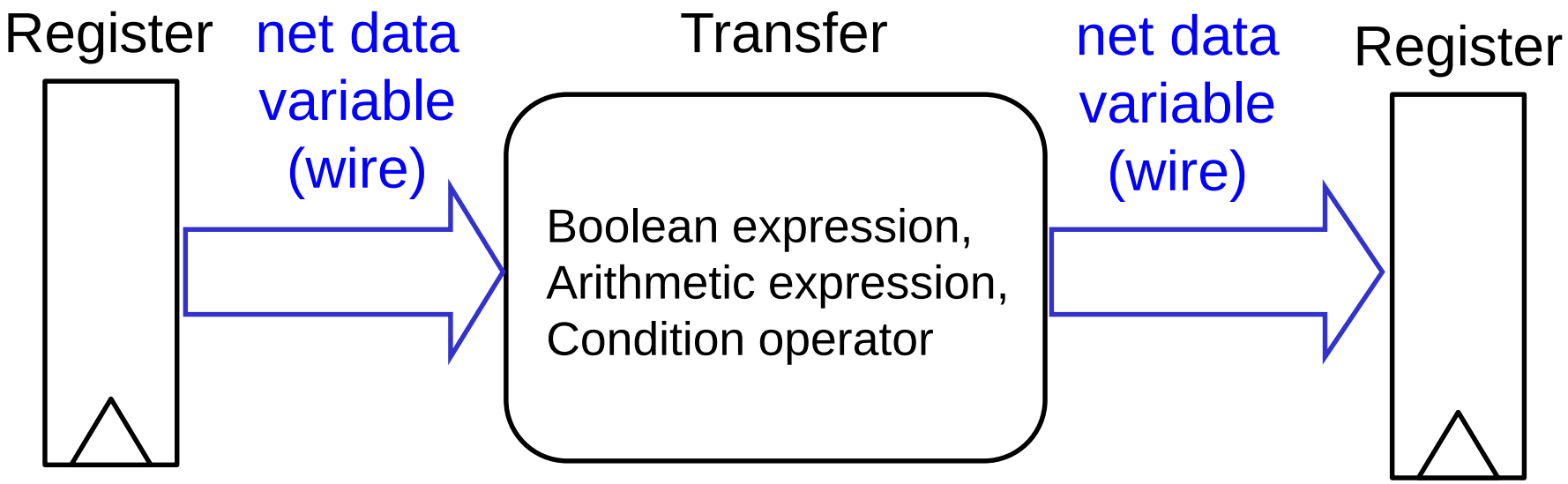
Lecturer: ENDpj(周敬堯)

Outline

- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - `initial` block
  - `always` block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Review of RTL

- Describe sequential elements (register) and combinational elements (transfer) of a synchronous circuit structurally in different parallel constructs.

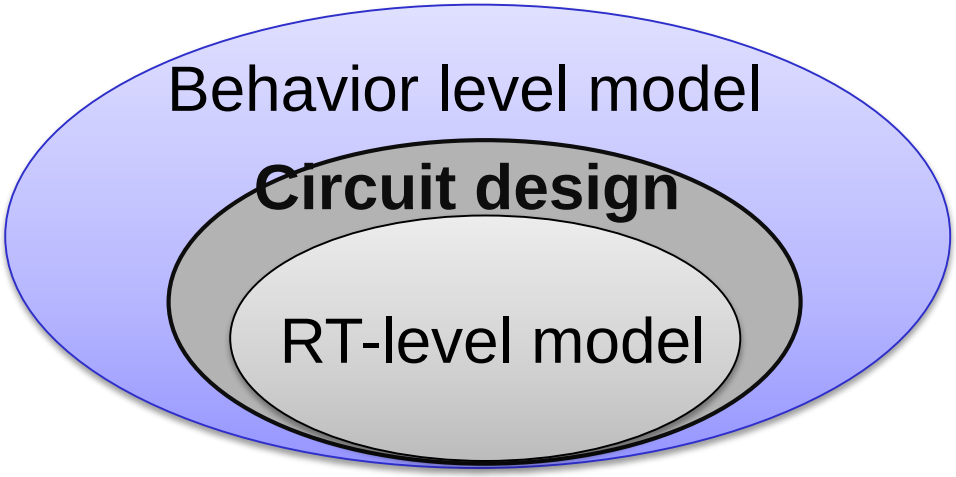


What is Behavioral Level

- Literally, modeling circuits using “behavioral” descriptions and events
- Description of an RTL model
  - Structure: constructs are separated for combinational and sequential circuits
  - Signal: continuous evaluate-update, pin accurate
  - Timing: cycle accurate
- Description of a behavioral level model
  - Structure: construct can be combinational, sequential, or hybrid circuits.
  - Signal: event-driven, timing, arithmetic (floating point or integer, … to pin accurate)
  - Timing: untimed (ordered), approximate-timed (with delay notification), cycle accurate
- **Note: Only a small part of behavioral level syntax is used for describing circuits, others are widely used for verification (testbench)**

Relationship between RTL and Behavioral Level

- Behavior level model
  - Hardware modeling
  - Implicit structure description for modeling
  - Flexible



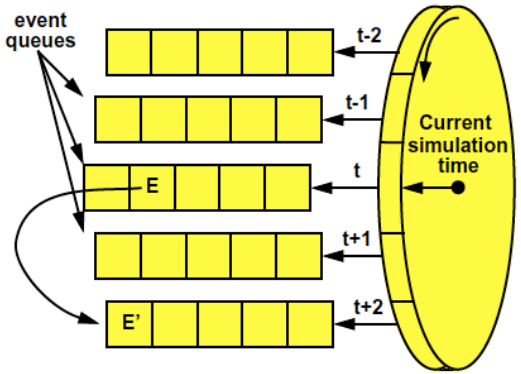
- RTL level model
  - Hardware modeling
  - Explicit structure description for modeling
  - Accurate

Event

- A data type has state and timing description
  - state: level change (1→0, 0→1, 1→X, 0→Z, etc.), edge (defined logic-level transition: e.g. 0→1)
  - timing: virtual time in simulator
  - example: signal s changed from 0 to 1 at 3ns

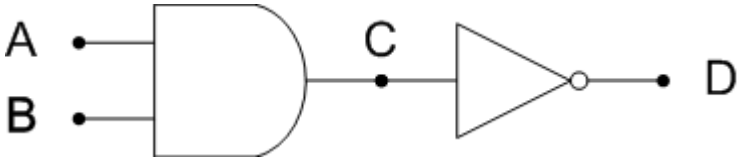
Event-Based Simulation

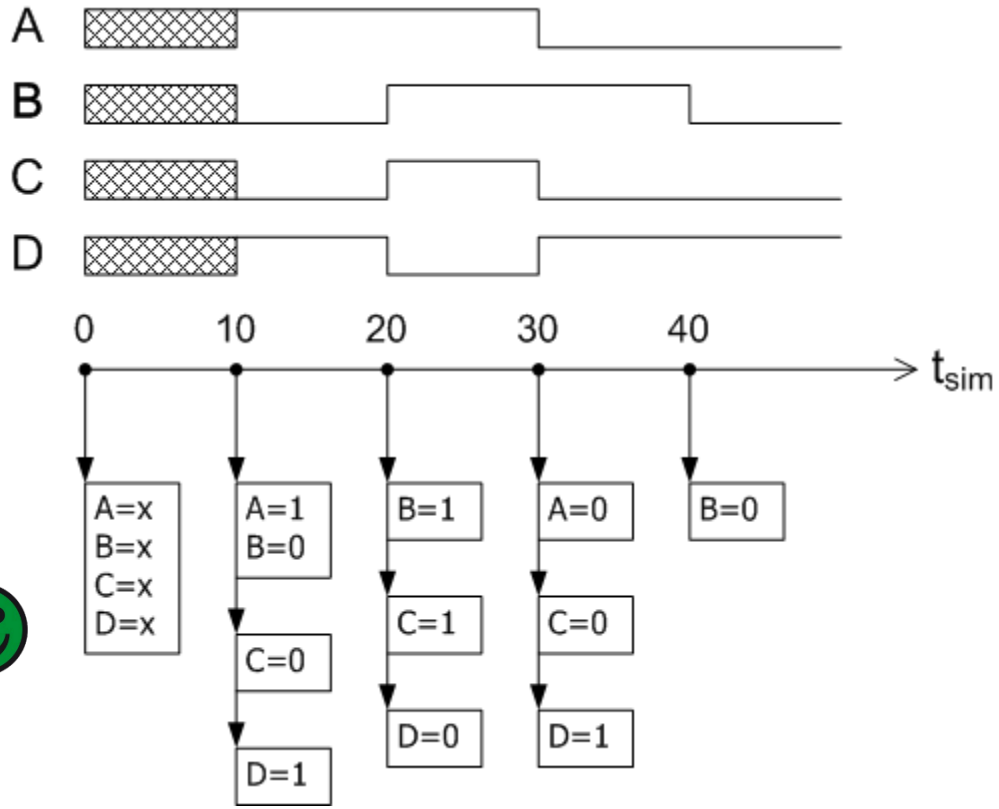
- Execute statement (evaluate expression and update variable) when defined event occurs
  - input transition cause an event on circuit
  - simulation is on the OR-ed occurrence of sensitive events
- Benefits
  - Accelerate simulation speed: Only evaluate expression when variables on RHS change
  - Allow high-level description (behavior) of a constructor



A current event scheduling another event

Example of Event-Based Simulation (1/)



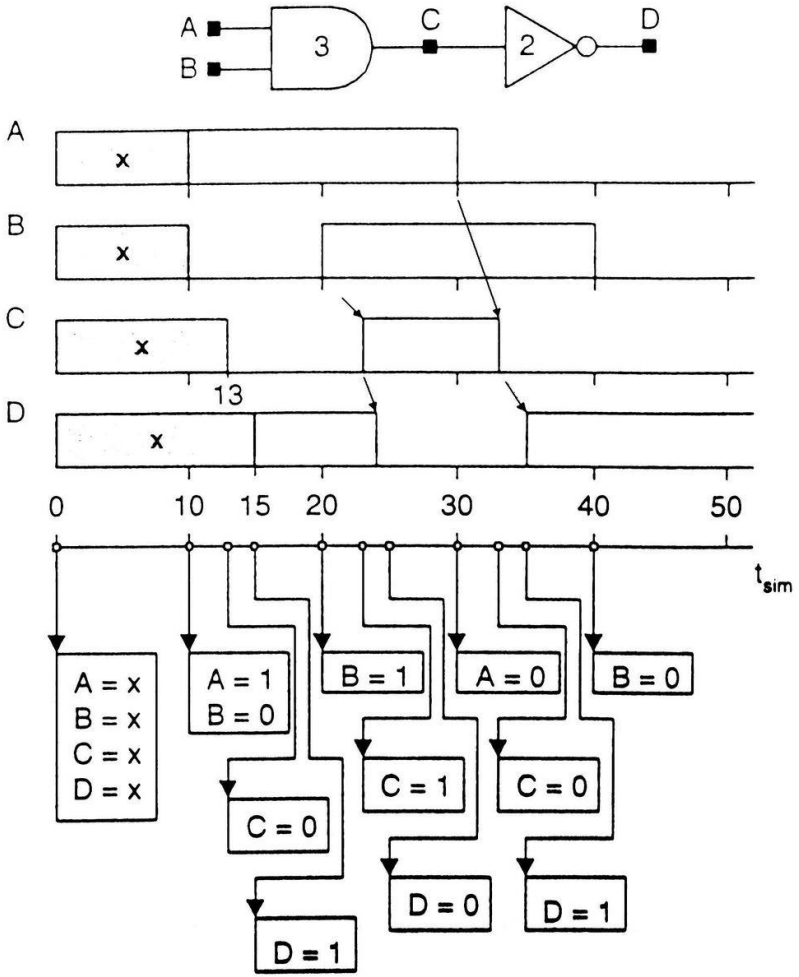


`timescale 1ns/10ps

Event-driven statement
execution number:
4+(2+1+1)+(1+1+1)+(1+1+1)+1 = 15

Continuous assignment
execution number:
(40ns / 10ps)+1 = 4001

Example of Event-Based Simulation (2/2)



Outline

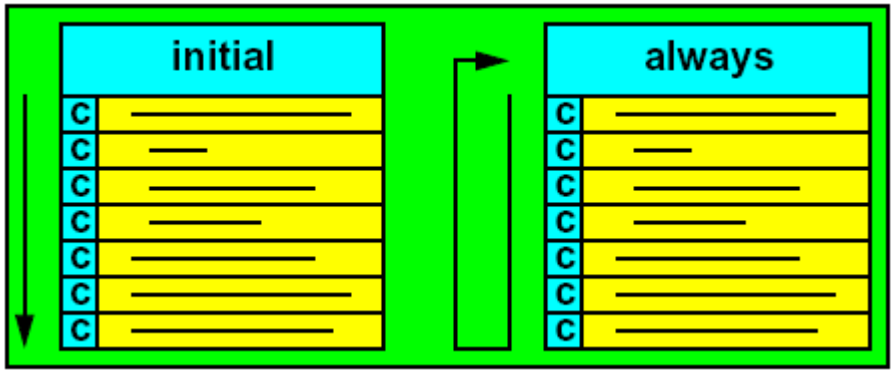
- Behavioral Level Modeling and Event-Based Simulation
- Procedural Construct and Assignment
  - **`initial`** block
  - **`always`** block
  - procedural assignment
- Data Path Modeling
  - Data Path
  - Timing Parameters
- Finite State Machine (FSM)
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Syntax Rules of Procedural Constructs

- 2 Types of Constructs
  - `initial` block
  - `always` block
- Only event-driven variable (`reg`) can be LHS in event-driven construct
  - It's syntax. Not relevant to whether it's a flip-flop or a metal wire!
  - RHS is not restricted.
- Event-driven variable is updated by *procedural assignment.*
  - Blocking *assignment*
  - Non-blocking *assignment*

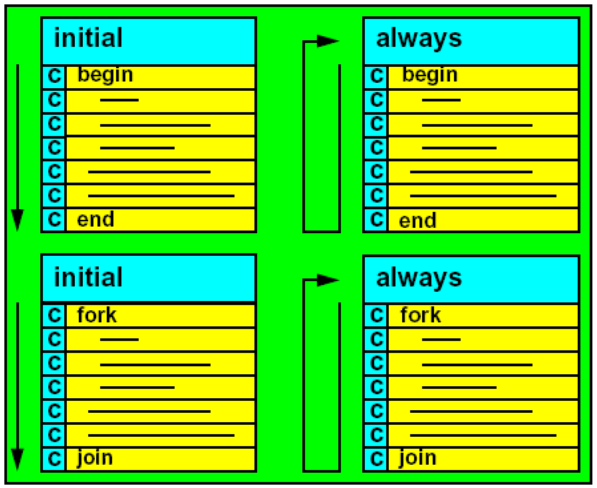
Procedural Blocks (1/2)

- 2 Types of blocks: **`initial`**, **`always`**
- Trigger conditions
  - **`initial`**: at the beginning of simulation (**NON**-SYNTHESIZABLE)
  - **`always`**: depends on the sensitivity list
- Components
  - Timing controls (#) (**NON**-SYNTHESIZABLE)
  - Conditional construct (if, else, case, …)
  - Procedural assignments



Procedural Blocks (2/2)

- Sequential block: **`begin`** and **`end`** group sequential procedural statements
- Concurrent block: **`fork`** and **`join`** group concurrent procedural statements (**NON**-SYNTHESIZABLE, not for modeling circuits)
- If the procedural block contains only one assignment, then both **`begin-end`** and **`for-join`** can be omitted
- Two types of blocks differ only if there's timing control inside



Two blocks do the same operations!

```
begin          fork
  #5 a = 1;      #5  a = 1;
  #5 a = 2;      #15 a = 3;
  #5 a = 3;      #10 a = 2;
end            join
```

Example

```
module assignment_test;
  reg [3:0] r1,r2;
  reg [4:0] sum2;
  reg [4:0] sum1;

  always @(r1 or r2)
    sum1 = r1 + r2;

  initial
  begin
      r1 = 4'b0010; r2=4'b1001;
      sum2 = r1+r2;
      $display(" r1   r2   sum1   sum2");
      $monitorb(r1, r2, sum1, sum2);

      #10 r1 = 4'b0011;
  end

endmodule
```

Result

r1	r2	sum1	sum2
0010	1001	01011	01011
0011	1001	01100	01011

Sensitivity List (1/2)

```
always@ ( a or b or cin)
  begin
    {cout, sum} = a + b + cin;
  end
```

- Edge-sensitive control (@)
  - The sensitivity list is described after "**always @**"
  - This means if any signals inside change (have an edge in waveform), the **always** block is triggered.
  - Keywords **or** are used to separate multiple signals
  - Keywords **posedge** & **negedge** is used when the **always** block should be triggered by positive or negative edge transition of the signal
  - Missing signals in sensitivity list may lead to wrong results!

WRONG!

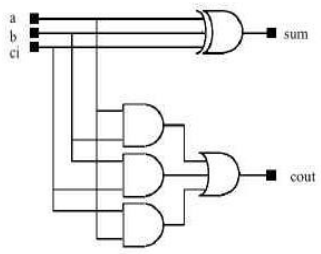
```
// initial a=0, b=0, x=0,  y=1
always@(a or b) begin  // 1. b changes to 1
    y = ~x;            // 2. y remains 1
    x = a | b;         // 3. x changes to 1
end
```

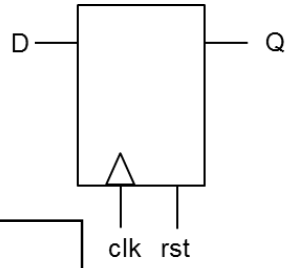
CORRECT!

```
// initial a=0, b=0, x=0,  y=1
always@(a or b or x) begin // 1. b changes to 1
    y = ~x;                // 2. y remains 1
                           // 4. y changes to 0
    x = a | b;             // 3. x changes to 1,
                                  trigger again!
                           // 5. x remains 1
end
```

Sensitivity List (2/2)

- Easy way to use sensitivity list
  - For combinational circuit
    - Pure logic circuit
    - Use always @ (*)
  - For sequential circuit
    - D-FF circuit
    - Edge trigger of clock or reset signal






```
module counter (cnt , data , ld , en , clk , rst);
output [7:0] cnt;
reg    [7:0] cnt;
input  [7:0] data;
input        ld , en , clk , rst;
reg    [7:0] ncnt;

always @ (*) begin
  if (rst)      ncnt = 0;
  else if (ld)  ncnt = data;
  else if (en)  ncnt = cnt+1;
  else          ncnt = cnt;
end

always @ (posedge clk or posedge rst) begin
  if (rst)  cnt <= 0;
  else      cnt <= ncnt
end
endmodule
```

Combinational part

Sequential part

Blocking & Non-blocking Assignments

- There are two types of procedural assignment statements: blocking (=) and non-blocking (<=)
- Blocking assignment (**=**)
  - Evaluate the RHS and pass to the LHS
  - Suitable for model or design the **combinational** circuit
  - **Difficult to model the concurrency**
- Non-blocking assignment (**<=**)
  - Evaluate the RHS, but schedule the LHS
  - Update LHS only after evaluate all RHS
  - Greatly simplify modeling concurrency

Blocking or Non-Blocking?

- Blocking assignment
  - Evaluation and assignment are immediate

```
always @ (a or b or c)
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
```

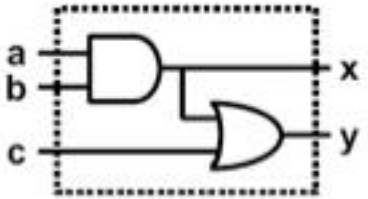
- Nonblocking assignment
  - All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

```
always @ (a or b or c)
begin
   x <= a | b;         1. Evaluate a | b but defer assignment of x
   y <= a ^ b ^ c;     2. Evaluate a^b^c but defer assignment of y
   z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                    4. Assign x, y, and z with their new values
```

Blocking for Combinational Logic

- Both synthesizable, but both correctly simulated?
- Non-blocking assignment do not reflect the intrinsic behavior of multi-stage combinational logic

Blocking Behavior	a b c x y
(Given) Initial Condition	1 1 0 1 1
a changes; always block triggered	0 1 0 1 1
x = a & b;	0 1 0 0 1
y = x | c;	0 1 0 0 0




```
module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c or x)
  begin
    x = a & b;
    y = x | c;
  end

endmodule
```

Nonblocking Behavior	a b c x y	Deferred
(Given) Initial Condition	1 1 0 1 1	
a changes; always block triggered	0 1 0 1 1	
x <= a & b;	0 1 0 1 1	x<=0
y <= x | c;	0 1 0 1 1	x<=0, y<=1
Assignment completion	0 1 0 0 1	

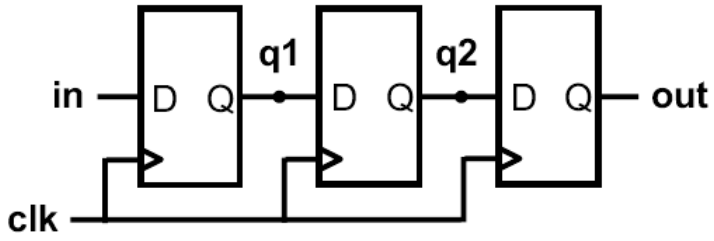
```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  always @ (a or b or c)
  begin
    x <= a & b;
    y <= x | c;
  end

endmodule
```

Non-Blocking for Sequential Logic

- Blocking assignment do not reflect the intrinsic behavior of multi-stage sequential logic

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

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."




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

"At each rising clock edge, *q1* = *in*.
After that, *q2* = *q1* = *in*.
After that, *out* = *q2* = *q1* = *in*.
Therefore *out* = *in*."