Introduction to Verilog

TA: Chun-Yen Yao
r08943003@ntu.edu.tw
Date: Apr. 26, 2021

Fundamentals of Hardware Description Language

Materials modified from

- Computer-Aided VLSI System Design
- Introduction to VLSI Design

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

Hardware Description Language

- **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.
- Why HDL
  - Native support to concurrency
  - Native support to the simulation of the progress of time
  - Native support to simulate the model of system

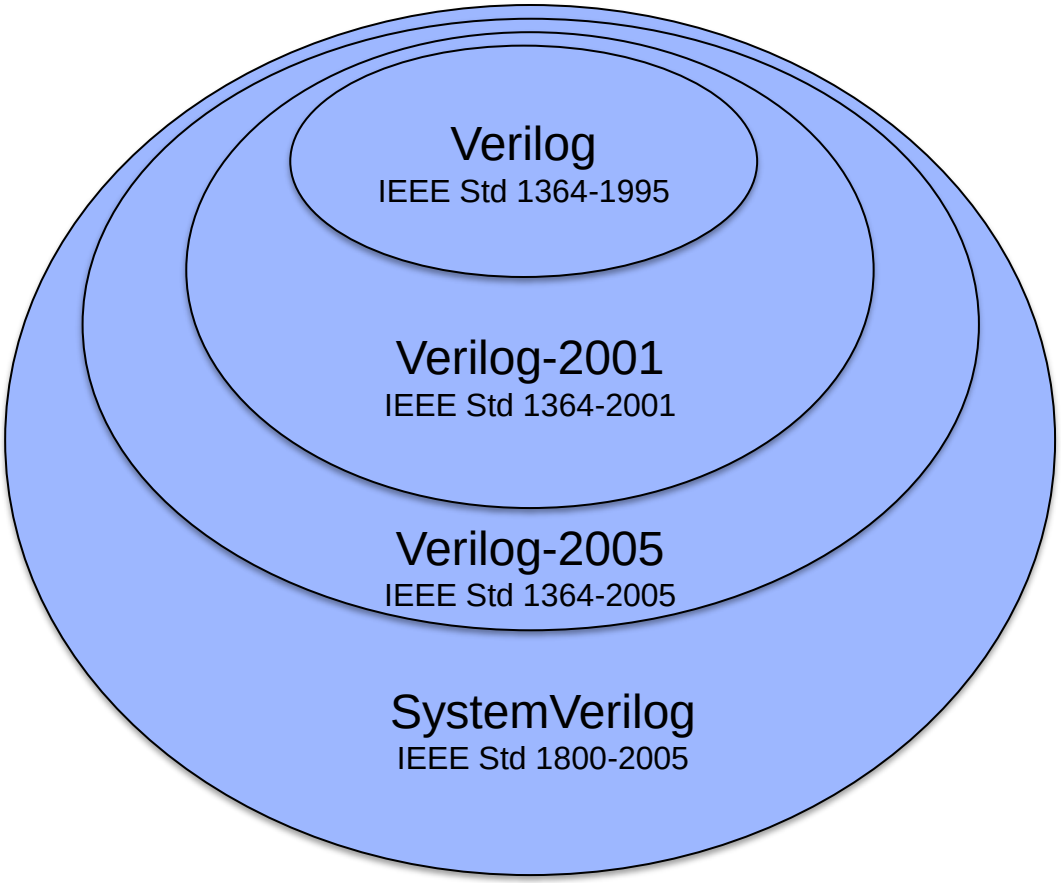
List of HDL for Digital Circuits

- **Verilog**
- **VHDL**
- Advanced Boolean Expression Language (ABEL)
- AHDL (Altera HDL, a proprietary language from Altera)
- Atom (behavioral synthesis and high-level HDL based on Haskell)
- Bluespec (high-level HDL originally based on Haskell, now with a SystemVerilog syntax)
- Confluence (a functional HDL; has been discontinued)
- CUPL (a proprietary language from Logical Devices, Inc.)
- Handel-C (a C-like design language)
- C-to-Verilog (Converts C to Verilog)
- HDCaml (based on Objective Caml)
- Hardware Join Java (based on Join Java)
- HML (based on SML)
- Hydra (based on Haskell)
- Impulse C (another C-like language)
- JHDL (based on Java)Lava (based on Haskell)
- Lola (a simple language used for teaching)
- MyHDL (based on Python)
- PALASM (for Programmable Array Logic (PAL) devices)
- Ruby (hardware description language)
- RHDL (based on the Ruby programming language) SDL based on Tcl.
- CoWareC, a C-based HDL by CoWare. Now discontinued in favor of SystemC
- SystemVerilog, a superset of Verilog, with enhancements to address system-level design and verification
- SystemC, a standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level
- SystemTCL, SDL based on Tcl.

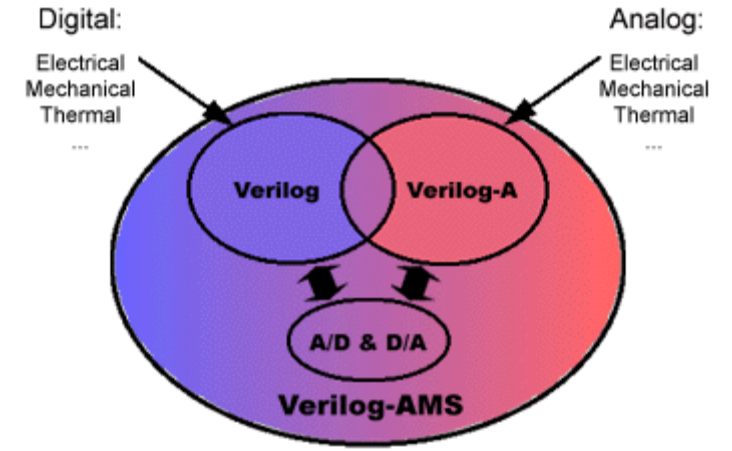
About Verilog

- Introduction on 1984 by Phil Moorby and Prabhu Goel in Automated Integrated Design System (renamed to Gateway Design Automation and bought by Cadence Design Systems)
- Open and Standardize (IEEE 1364-1995) on 1995 by Cadence because of the increasing success of VHDL (standard in 1987)
- Become popular and makes tremendous improvement on productivity
  - Syntax similar to C programming language, though the design philosophy differs greatly

History/Branch of Verilog



Digital-signal HDL



Mixed-signal/Multi-domain Hardware Discription Language

Outline

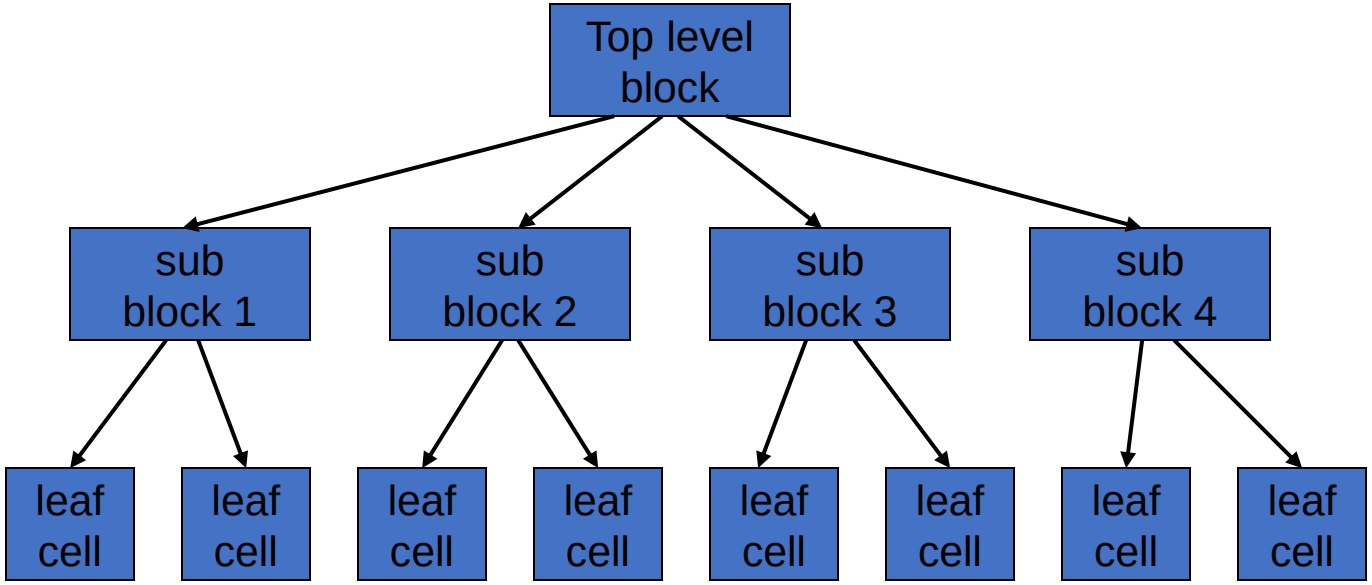
- Overview and History
- Hierarchical Design Methodology
- Levels of Modeling
  - Behavioral Level Modeling
  - Register Transfer Level (RTL) Modeling
  - Structural/Gate Level Modeling
- Language Elements
  - Logic Gates
  - Data Type
  - Timing and Delay

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

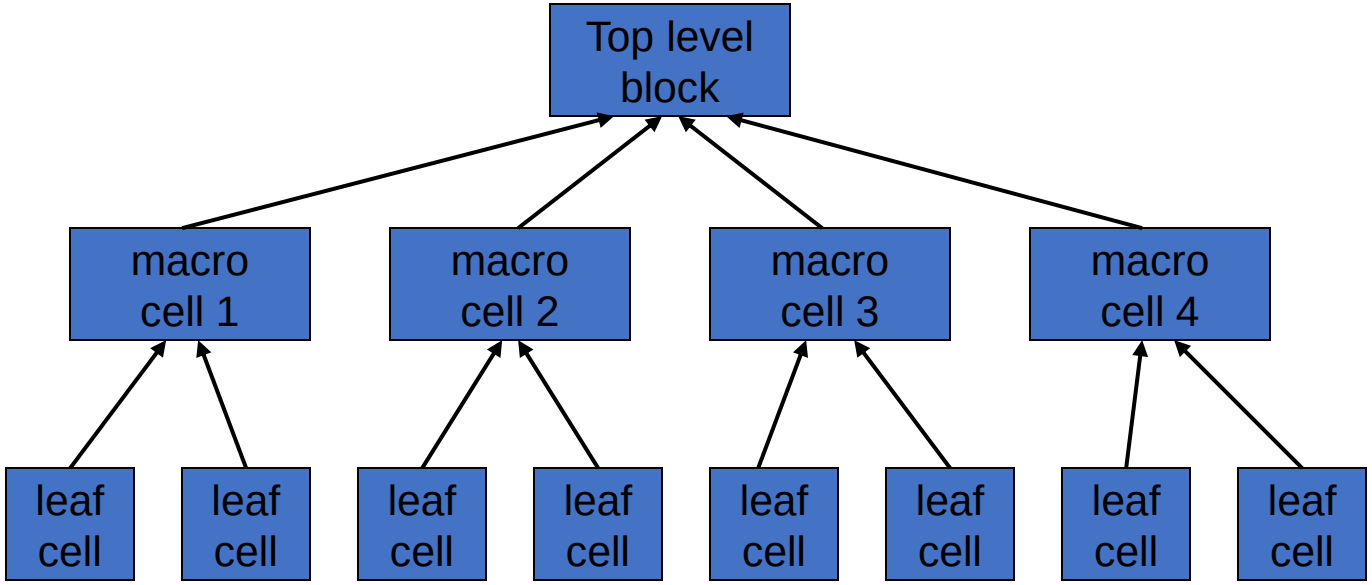
Top-down Design Methodology

- 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.

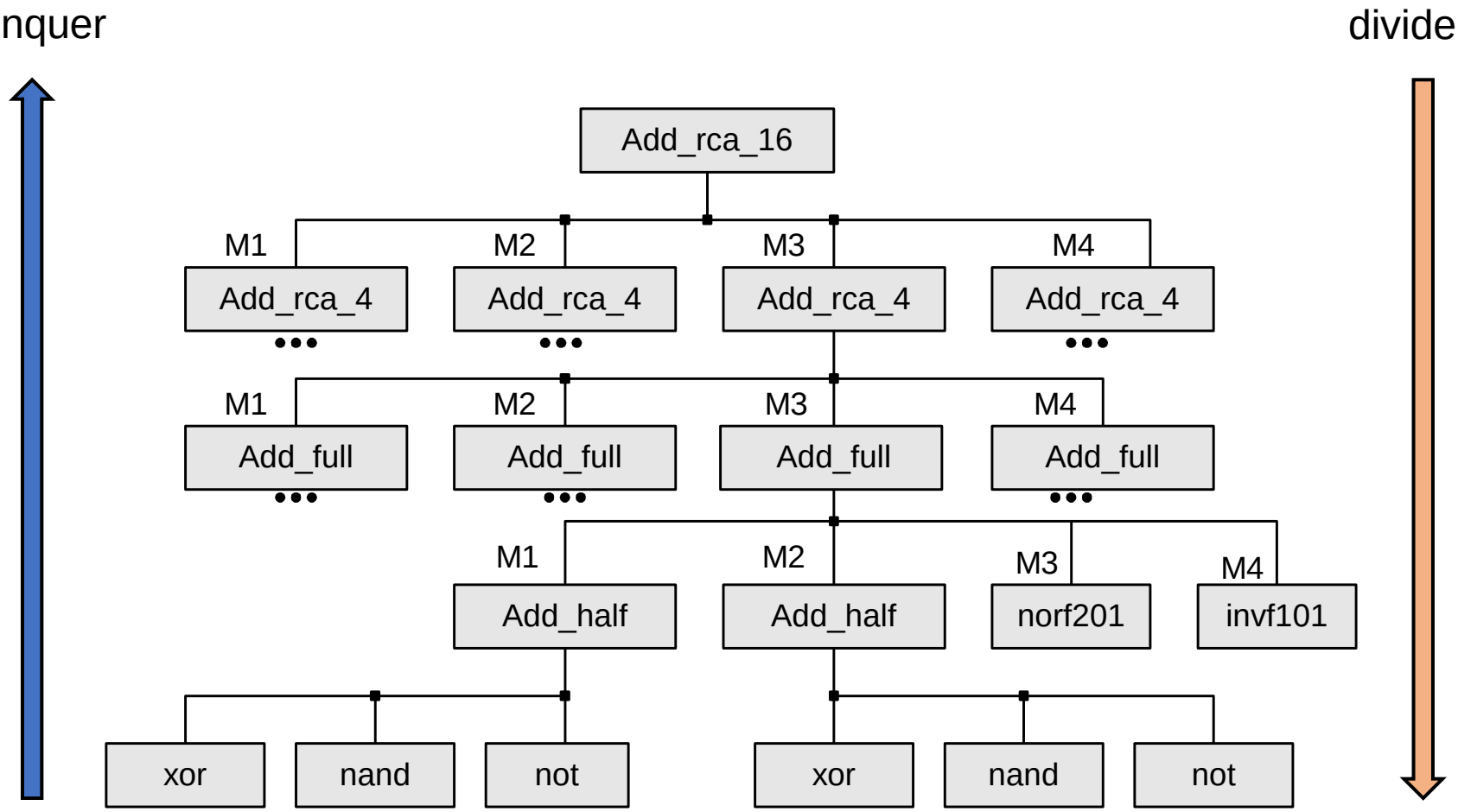


Bottom-up Design Methodology

- 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.

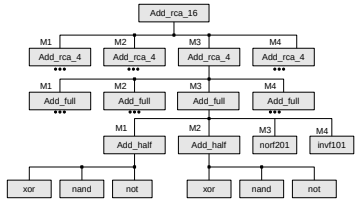


Example: 16-bit Adder



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>     // For testbench
    <System Task> // For testbench
endmodule
```

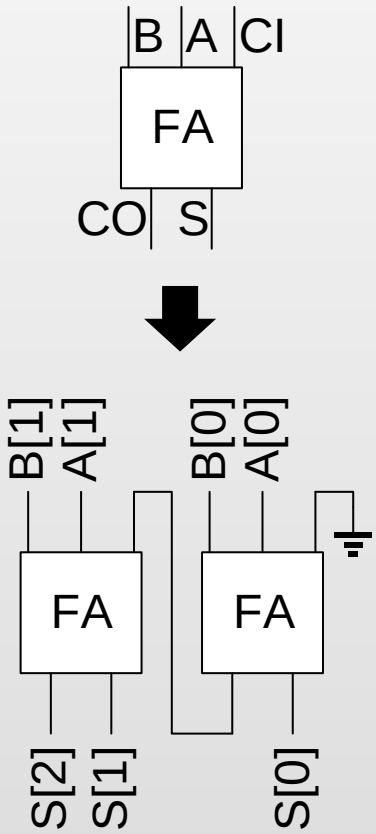
- Encapsulation makes the model available for instantiation in other modules

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.

Example

```
// Declare a module
module my_FA(A, B, CI, S, CO);
    // I/O
    input  A, B, CI;
    output S, CO;
    // Function
    assign S  = A ^ B ^ CI;
    assign CO = (A && B) || (B && CI) || (A && CI);
endmodule

// Instantiate modules
module my_CRA(A, B, S);
    // I/O and wires
    input  [1:0] A, B;
    output [2:0] S;
    wire   carry;
    my_FA inst0(.A(A[0]), .B(B[0]), .CI(1'b0), .S(S[0]), .CO(carry));
    my_FA inst1(.A(A[1]), .B(B[1]), .CI(carry), .S(S[1]), .CO(S[2]));
endmodule
```



Analogy: module ↔ class (1/2)

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
Hierarchy	instance.sub_instance.member_signal	object.sub_object.member_data

Analogy: module ↔ class (2/2)

```
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

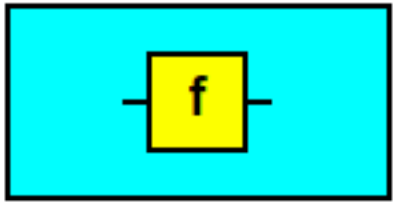
- assign and evaluate() is simulated/called at each $T_{i+1} = T_i + t_{resolution}$

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

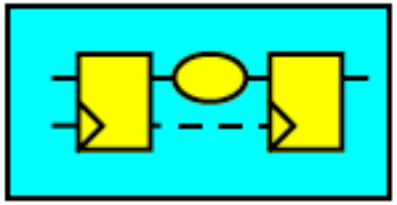
Cell-Based Design and Levels of Modeling

Behavioral Level



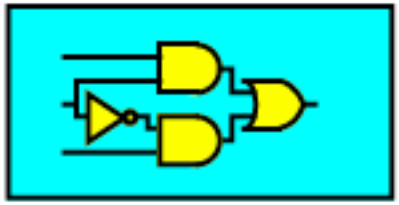
Most used modeling level, easy for designer, can be simulated and synthesized to gate-level by EDA tools

Register Transfer Level (RTL)



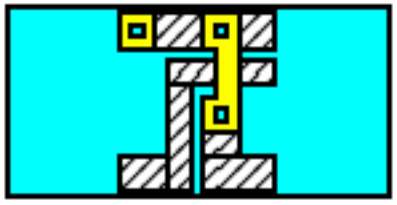
Common used modeling level for small sub-modules, can be simulated and synthesized to gate-level

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

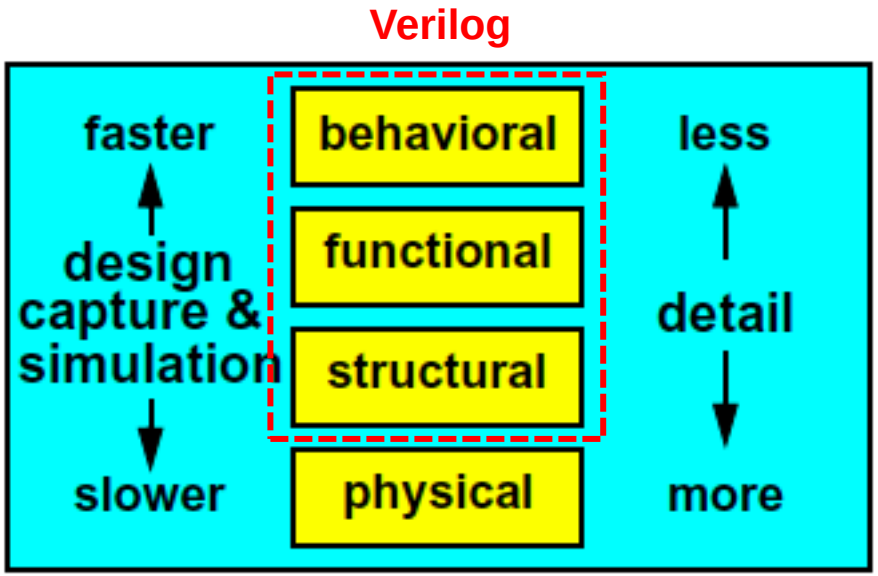
Transistor/Physical Level



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

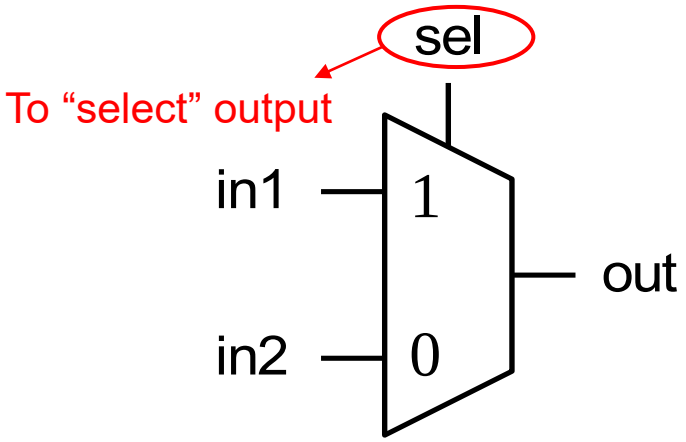
Tradeoffs Among Modeling Levels

- Each level of modeling permits modeling at a higher or lower level of details
- More details mean more efforts for designers and the simulator



An Example - 1-bit Multiplexer (1/3)

- Behavior 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

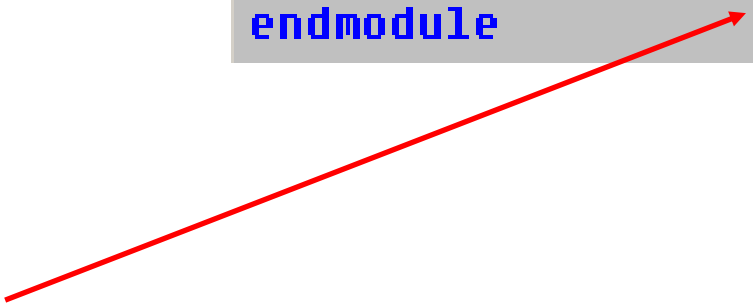
An Example - 1-bit Multiplexer (2/3)

- 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

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

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

Continuous assignment

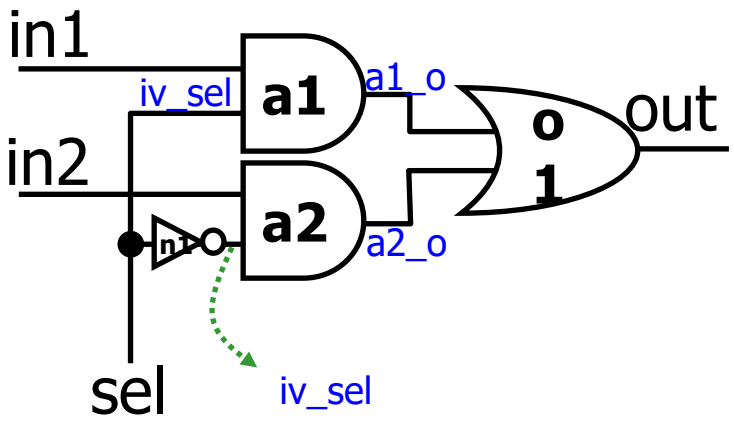


out = sel? in1 : in2

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

An Example - 1-bit Multiplexer (3/3)

- 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

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

Verilog Language Rules

- Verilog is a **case sensitive** language (with a few exceptions)
  - **Avoid to use**
- **Identifiers** (space-free sequence of symbols)
  - upper and lower case letters from the alphabet
  - digits (*0, 1, ..., 9*)
  - underscore (_)
  - *$* symbol (for system tasks)
  - Max length of 1024 symbols
- Terminate lines with semicolon **;**
- Single line comments:
  - // A single-line comment goes here
- Multi-line comments:
  - /* Multi-line comments like this
    Multi-line comments like this */

Primitives

- Primitives are modules ready to be instanced
- **Smallest modeling block for simulator**
  - Behavior as software execution in simulator, not hardware description
- Verilog build-in primitive gate
  - *and, or, not, buf, xor, nand, nor, xnor*
  - prim_name inst_name(output, in0, in1,....);
  - Ex:

```
and u0(z, a, b);
xor u1(z, a, b, c);
```

- User defined primitive (UDP)
  - Building block defined by designer

Verilog Built-in Primitives

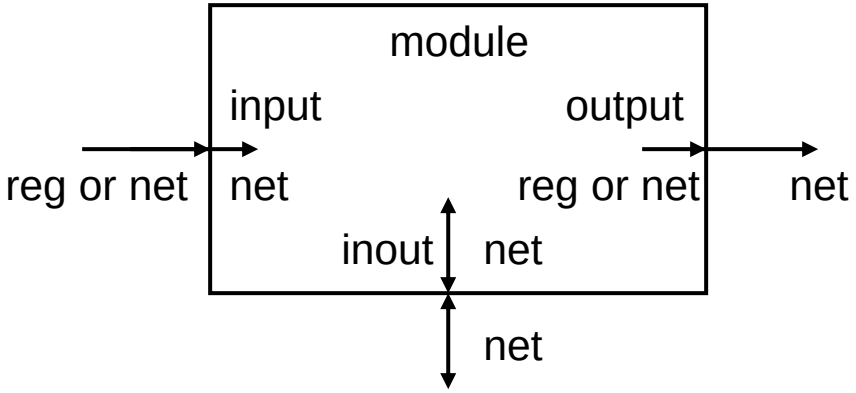
		Ideal MOS switch	Resistive gates	
and	buf	nmos	rnmos	pullup
nand	not	pmos	rpmos	pulldown
or	bufif0	cmos	rcmos	
nor	bufif1	tran	rtran	
xor	notif0	tranif0	rtranif0	
xnor	notif1	tranif1	rtranif1	

Verilog Basic Cell

- Verilog Basic Components
  - Port declarations
  - Nets or reg declarations
  - Parameter declarations
  - Gate instantiations
  - Module instantiations
  - Continuous assignments
  - Always blocks
  - Function definitions
  - Task statements

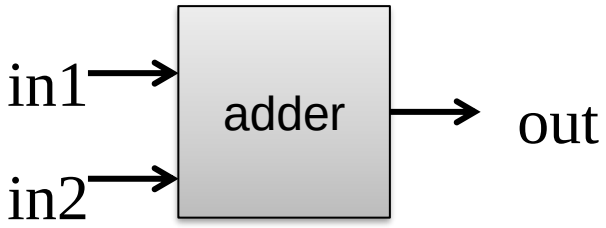
Port Declaration

- Three port types
  - Input port
    - `input a;`
  - Output port
    - output b;
  - Bi-direction port (do not use)
    - inout c;



Port Connection

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

  assign out = in1 + in2;
endmodule
```



- Connect module ports by *order list*
  - `adder adder_0 ( C , A , B );  // C = A + B`
- Connect module ports by *name* **(Recommended)**
  - Usage: .PortName (NetName)
  - `adder adder_1 ( .out(C) , .in1(A) , .in2(B) );`
- Not fully connected **(Avoid)**
  - `adder adder_2 ( .out(C) , .in1(A) , .in2() );`

Data Types

- **Common types**
  - nets are further divided into several net types
    - **wire, wand**, wor, tri, triand, trior, supply0, supply1
  - registers – variable to store a logic value for event-driven simulation - **reg**
  - integer - supports computation 32-bits signed
- **Other types**
  - time - stores time 64-bit unsigned
  - real - stores values as real numbers
  - realtime - stores time values as real numbers
  - event – an event data type

Wire & Reg (1/2)

- wire
  - Physical wires in a circuit
  - Cannot assign a value to a wire within a `function` or a `begin...end` block
  - A wire does not store its value, it must be driven by
    - Connecting the wire to the output of a gate or module
    - Assigning a value to the wire in a continuous assignment
  - An un-driven wire defaults to a value of Z (high impedance).
  - Input, output, inout port declaration -- wire data type (default)

```
output out;
input in1,in2,sel;
reg out;
```

Wire & Reg (2/2)

- reg
  - A event driven variable in Verilog
  - **Not** exactly stands for a really register
- Usage of wire & reg
  - When use "`wire`" ➔ usually use "`assign`" and "`assign`" **does not** appear in "`always`" block
  - When use "`reg`" ➔ only use "`a=b`" in "`always`" or other blocks

```
module my_DFF(clk, D, Q);
    input clk,D;
    output Q;
    always @(posedge clk) begin
        Q <= D;
    end
endmodule
```

Vector

- `wire` and `reg` can be defined vector, default is 1 bit
- Vector is multi-bit element
- Format: **[High#:Low#]** or [Low#:High#]
- Using range specify part signals

```
wire        a;      // scalar net variable, default
wire [7:0]  bus;    // 8-bit bus
reg         clock;  // scalar register, default
reg [0:23]  addr;   // Vector register, virtual address 24 bits wide
```

```
bus[7]      // bit #7 of vector bus
bus[2:0]    // Three least significant bits of vector bus
            // using bus[0:2] is illegal because the significant bit should
            // always be on the left of a range specification
addr[0:1]   // Two most significant bits of vector addr
```

Array

- Arrays are allowed in Verilog for `reg`, `integer`, time, and vector register data types

```
integer     count[0:7];          // An array of 8 count variables
reg         bool[31:0];          // Array of 32 one-bit Boolean register variables
time        chk_ptr[1:100];      // Array of 100 time checkpoint variables
reg  [4:0]  port_id[0:7];        // Array of 8 port_id, each port_id is 5 bits wide
integer     matrix[4:0][4:0]     // Two dimension array
```

```
count[5]        // 5th element of array of count variables
chk_ptr[100]    // 100th time check point value
port_id[3]      // 3rd element of port_id array. This is a 5-bit value
```

```
port_id[3][2:0]    // part select for certain element in array
```

Memory

- In digital simulation, one often needs to model register files, RAMs, and ROMs.
- Memories are modeled in Verilog simply as an array of registers.
- Each element of the array is known as a word, each word can be one or more bits.
- It is important to differentiate between
  - n 1-bit registers
  - One n-bit register

```
reg  mem1bit[0:1023];          // Memory mem1bit with 1K 1-bit words
reg  [7:0]  mem1byte[0:1023];  // Memory mem1byte with 1K 8-bit words
```

```
mem1bit[255]        // Fetches 1 bit word whose address is 255
Mem1byte[511]       // Fetches 1 byte word whose address is 511
```

String

- String: A sequence of 8-bit ASCII values

```
module string;
  reg  [8*14:1]  strvar;
  initial
    begin
      strvar = “Hello World”;    // stored as 000000486561...726c64
    end
endmodule
```

- Special characters

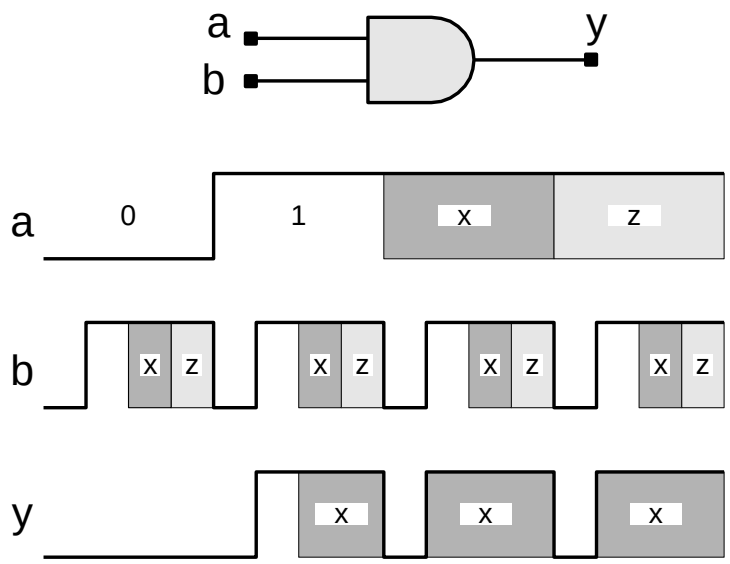
\n ➔ newline	\t ➔ tab character
\\ ➔ \ character	\” ➔ “ character
%% ➔ % character	\abc ➔ ASCII code

Four-Valued Logic System

- Verilog's nets and registers hold four-valued data
  - 0 represent a logic **low** or **false** condition
  - 1 represent a logic **high** or **true** condition
  - z
    - Output of an undriven tri-state driver – high-impedance value
    - Models case where nothing is setting a wire's value
  - x
    - Models when the simulator can't decide the value – uninitialized or unknown logic value
      - Initial state of registers
      - When a wire is being driven to 0 and 1 simultaneously
- Unknown value would propagate
  - Unknown input would cause unknown output, and make following stage all become unknown

Logic System in Verilog

- **Four values: 0, 1, x or X, z or Z // Not case sensitive here**
  - The logic value **x** denotes an unknown (ambiguous) value
  - The logic value **z** denotes a high impedance
- **Primitives have built-in logic**
- **Simulators describe 4-value logic**



	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Number Representation (1/2)

- Format: <size>'<base_format><number>
  - <size> - decimal specification of number of bits
    - default is unsized and machine-dependent but at least 32 bits
  - <base format> - ' followed by arithmetic base of number
    - <d> <D> - decimal - default base if no <base_format> given
    - <h> <H> - hexadecimal
    - <o> <O> - octal
    - <b> <B> - binary
  - <number> - value given in base of <base_format>
    - _ can be used for reading clarity
    - If first character of sized, binary number 0, 1, x or z, will extend 0, 1, x or z (defined later!)

Number Representation (2/2)

- Examples:
  - 6′b010_111 gives 010111
  - 8′b0110 gives 00000110
  - 4′bx01 gives xx01
  - 16′H3AB gives 0000_0011_1010_1011
  - 24 gives 0...0011000 **(default as 32-bit unsigned decimal)**
  - 5′O36 gives 11_100
  - 16′Hx gives xxxxxxxxxxxxxxxx
  - 8′hz gives zzzzzzzz
  - ′hff gives 0…0_1111_1111

Vector Concatenations

- An easy way to group vectors into a larger vector

Representation	Meanings
{cout, sum}	{cout, sum}
{b[7:4],c[3:0]}	{b[7], b[6], b[5], b[4], c[3], c[2], c[1], c[0]}
{a,b[3:1],c,2'b10}	{a, b[3], b[2], b[1], c, 1'b1, 1'b0}
{4{2'b01}}	8'b01010101
{{8{byte[7]}},byte}	Sign extension

Parameter Declaration

- Parameters are not variables, they are constants (hardware design view)
- Can be defined as a bit or a vector
- Typically parameters are used to specify conditions, states, width of vector, entry number of array, and delay

```
module  var_mux(out, i0, i1, sel);
  parameter width = 2, flag = 1'b1;
  output  [width-1:0]  out;
  input  [width-1:0]  v0, v1;
  input  sel;

  assign out = (sel==flag) ? v1 : v0;
endmodule
```

Good for flexibility and reusability

- If sel = 1, then v1 will be assigned to out;
- If sel = 0, then v0 will be assigned to out;

Logic Design

Materials modified from

- Computer-Aided VLSI System Design
- Introduction to VLSI Design

Outline

- Register-Transfer Level (RTL)
  - Definition and Verilog Syntax
  - Continuous Assignments for Combinational Circuits
- Behavior Level
  - Modeling and Event-Based Simulation
  - Procedural Construct and Assignment
  - Control Construct
- Finite State Machine (FSM)
  - Data Path Modeling
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

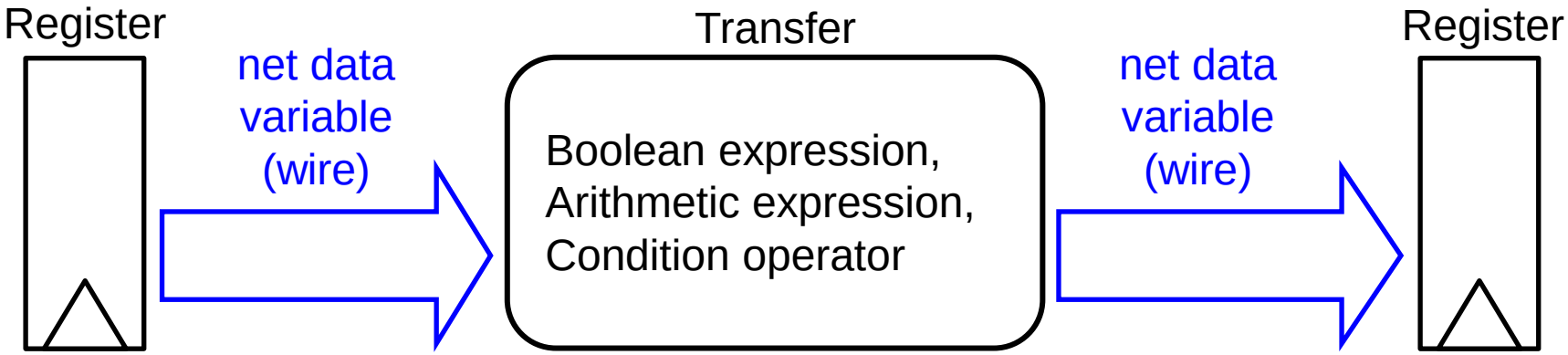
What is Register Transfer Level?

- 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

Schematic Diagram

- Register (**sequential** elements, usually D flip-flops)
- Transfer (operation, **combinational**)



- Example:
  8-bit adder

```
module my_adder(A, B, S);
    input  [7:0] A, B;
    output [8:0] S;
    // Transfer description
    assign S = {A[7],A} + {B[7],B};
endmodule
```

Procedures to Design at RT-Level

- **Design partition**
  - **Sequential (register part)**
  - **Combinational (transfer part)**
- **Transfer Circuit design**
  - **Logical operator**
  - **Arithmetical operator**
  - **Conditional operator**

```
// Input signals of transfer part
    wire [7:0] in1;
    wire [7:0] in2;
    wire [7:0] in3;
// Output signals of transfer part
    wire [7:0] out1;
    wire [7:0] out2;
    wire [7:0] out3;
// Transfer description (Continuous assignments)
    assign out1 = in1 & in2;                  // Logical operator
    assign out2 = in1 + in3;                  // Arithmetical operator
    assign out3 = in1[0]==1'b1 ? in2 : in3; // Conditional operator
```

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 complement binary

```
reg [4:0] a;
reg signed [4:0] a;
```

  - 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	? :

Operator Precedence

Type of Operators	Symbols
Concatenate & replicate	{ } {{ }}
Unary	! ~ & ^ ^~ |
Arithmetic	* / %
	+ -
Logical shift	<< >>
Relational	< <= > >=
Equality	== != === !==
Binary bit-wise	& ^ ^~ |
Binary logical	&& ||
Conditional	? :

Use parentheses (i.e. “()”) to explicitly define the operation order

Concatenation & Replication

- Concatenation and Replication Operator ({})

Concatenation operator in LHS

```
module add_32 (co, sum, a, b, ci);
  output co;
  output [31:0] sum;
  input  [31:0] a, b;
  input  ci;
    assign #100 {co, sum} = a + b + ci;
endmodule
```

Bit replication to produce *01010101*

```
assign byte = {4{2'b01}};
```

Sign Extension

```
assign word = {{8{byte[7]}}, byte};
```

Negation Operators

- The logical negation operator (**!**)
  - produces a 0, 1, or X scalar value.
- The bitwise negation operator (**~**)
  - inverts each individual bit of the operand.

```
module negation;
  initial begin
      $displayb ( !4'b0100 ); // 0
      $displayb ( !4'b0000 ); // 1
      $displayb ( !4'b00z0 ); // x
      $displayb ( !4'b000x ); // x
      $displayb ( ~4'b01zx ); // 10xx
    end
endmodule
```

Unary Reduction Operators

- The unary reduction operators (**&**, **|**, **^**, **^~**) produce a 0, 1, or X scalar value.

```
module reduction;
  initial begin
      $displayb (  &4'b1110 ); // 0
      $displayb (  &4'b1111 ); // 1
      $displayb (  &4'b111z ); // x
      $displayb (  &4'b111x ); // x
      $displayb (  |4'b0000 ); // 0
      $displayb (  |4'b0001 ); // 1
      $displayb (  |4'b000z ); // x
      $displayb (  |4'b000x ); // x
      $displayb (  ^4'b1111 ); // 0
      $displayb (  ^4'b1110 ); // 1
      $displayb (  ^4'b111z ); // x
      $displayb (  ^4'b111x ); // x
      $displayb ( ^~4'b1111 ); // 1
      $displayb ( ^~4'b1110 ); // 0
      $displayb ( ^~4'b111z ); // x
      $displayb ( ^~4'b111x ); // x
    end
endmodule
```

Arithmetic Operators

- The arithmetic operators (*, /, %, +, -)
  - Produce numerical or unknown results
  - Integer division discards any remainder
  - An unknown operand produces an unknown result
  - Assignment of a signed value to an unsigned register is 2's-complement

```
-3 >>> 1;  // -2
( assume as 4 bit number
  -3 = ~(0011) + 1 = 1101
  1101 >>> 1 → 1110 (-2) )
```

```
module arithmetic;
  initial begin
      $display  ( -3 * 5 ); // -15
      $display  ( -3 / 2 ); // -1
      $display  ( -3 % 2 ); // -1
      $display  ( -3 + 2 ); // -1
      $display  (  2 - 3 ); // -1
      $displayh ( 32'hfffffffd / 2 ); // 7ffffffe
      $displayb ( 2 * 1'bx); // xx...
    end
endmodule
```

Example

```
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 negA  =  -A;
endmodule
```

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

```
module test (); // Testbench
    reg  [3:0] A, B;
    wire [4:0] sum, diff1, diff2, negA;
    DUT mydesign(.sum(sum),.diff1(diff1),diff2(diff2),.negA(negA),
        .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,negA);
        #10 $monitoroff;
        $monitor($time,"%d%d%d%d%d%d",A,B,sum,diff1,diff2,negA);
    end
endmodule
```

Shift Operators

- Shift operator
  - **"<<"** logical shift left
  - **">>"** logical shift right
  - **">>>"** arithmetic shift right (if variable is **signed**)
- Treat right operand as unsigned

```
module shift;
  initial begin
      $displayb ( 8'b00011000 << 2    ); // 01100000
      $displayb ( 8'b00011000 >> 2    ); // 00000110
      $displayb ( 8'b00011000 >> -2   ); // 00000000
      $displayb ( 8'b00011000 >> 1'bx ); // xxxxxxxx
    end
endmodule
```

-2 = 1...110 (32-b integer)
→ Take it as unsigned
→ 1...110 = a large number

Relational Operators

- Relational operators (< , <= , >= , >)

```
module relational;
  initial begin
      $displayb ( 4'b1010 <  4'b0110 ); // 0
      $displayb ( 4'b0010 <= 4'b0010 ); // 1
      $displayb ( 4'b1010 <  4'b0x10 ); // x
      $displayb ( 4'b0010 <= 4'b0x10 ); // x
      $displayb ( 4'b1010 >= 4'b1x10 ); // x
      $displayb ( 4'b1x10 >  4'b1x10 ); // x
      $displayb ( 4'b1z10 >  4'b1z10 ); // x
    end
endmodule
```

Equality Operators

- Equality operators
  - "==" , "!="does not perform a definitive match for Z or X.
  - "===" , "!=="does perform a definitive match for Z or X.

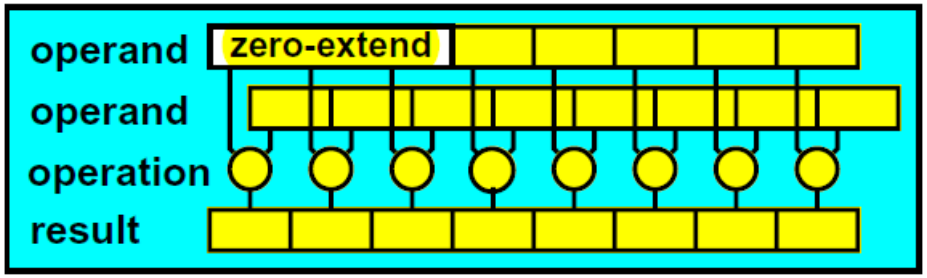
```
module equality;
  initial begin
      $displayb ( 4'b0011 == 4'b1010 ); // 0
      $displayb ( 4'b0011 != 4'b1x10 ); // 1
      $displayb ( 4'b1010 == 4'b1x10 ); // x
      $displayb ( 4'b1x10 == 4'b1x10 ); // x
      $displayb ( 4'b1z10 == 4'b1z10 ); // x
    end
endmodule
```

```
module identity();
  initial begin
      $displayb ( 4'b01zx === 4'b01zx ); // 1
      $displayb ( 4'b01zx !== 4'b01zx ); // 0
      $displayb ( 4'b01zx === 4'b00zx ); // 0
      $displayb ( 4'b01zx !== 4'b11zx ); // 1
    end
endmodule
```

avoid using in DUT

Bit-Wise Operators

- Bit-wise operators (&,|,^,^~)
  - operate on each individual bit of a vector




```
module bit_wise;
  initial begin
      $displayb ( 4'b01zx  & 4'b0000 ); // 0000
      $displayb ( 4'b01zx  & 4'b1100 ); // 0100
      $displayb ( 4'b01zx  & 4'b1111 ); // 01xx
      $displayb ( 4'b01zx  | 4'b1111 ); // 1111
      $displayb ( 4'b01zx  | 4'b0011 ); // 0111
      $displayb ( 4'b01zx  | 4'b0000 ); // 01xx
      $displayb ( 4'b01zx  ^ 4'b1111 ); // 10xx
      $displayb ( 4'b01zx ^~ 4'b0000 ); // 10xx
    end
endmodule
```

Logical Operators

- Logical operators (&&,||)
  - An operand is logically false if **all** of its bits are 0
  - An operand is logically true if **any** of its bits are 1

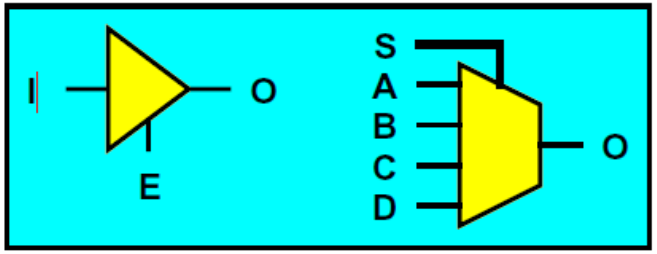
```
module logical;
  initial begin
      $displayb ( 2'b00 && 2'b10 ); // 0
      $displayb ( 2'b01 && 2'b10 ); // 1
      $displayb ( 2'b0z && 2'b10 ); // x
      $displayb ( 2'b0x && 2'b10 ); // x
      $displayb ( 2'b1x && 2'b1z ); // 1
      $displayb ( 2'b00 || 2'b00 ); // 0
      $displayb ( 2'b01 || 2'b00 ); // 1
      $displayb ( 2'b0z || 2'b00 ); // x
      $displayb ( 2'b0x || 2'b00 ); // x
      $displayb ( 2'b0x || 2'b0z ); // x
    end
endmodule
```

Usually connect two Boolean value (e.g. (a > b) && (a > 0))

Conditional Operators

- Conditional Operator
  - Usage:
    - ***conditional_expression ? true_expression: false_expression;***
    - ***If…else…***
    - ***Case….endcase…***

```
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
```



Width and Sign

- Verilog fills in operands with smaller width by **zero extension**
- Final or intermediate result widths may make expression width increase
- Constant number with undefined bit width: same as integer (usually 32 bits)

NOTICE: Verilog fills in **SIGNED** operands with sign extension

```
reg [3:0] a, b;
reg [7:0] sum;

always @(*) begin
    sum = a + b;
    // sum = {4'd0, a} + {4'd0, b}
end
```

```
reg signed [3:0] a, b;
reg [7:0] sum;

always @(*) begin
    sum = a + b;
    // sum = {{4{a[3]}}, a} + {{4{b[3]}}, b}
end
```

Continuous Assignments (1/4)

- The LHS of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register.
- Continuous assignments are always active. Any changes in RHS of the continuous assignment are evaluated and the LHS is updated.
- The operands on the RHS can be registers or nets or function calls. Register or nets can be scalars or vectors.

Continuous Assignments (2/4)

- 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 (3/4)

- Continuous assignments provide a simpler 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
```

Continuous Assignments (4/4)

- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared (not recommend)

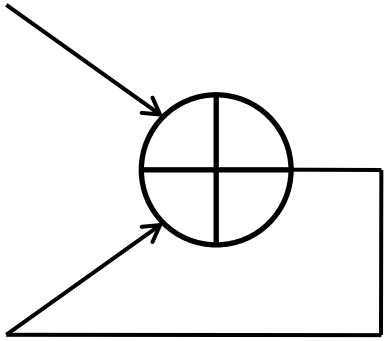
- Example

```
assign out = i1 & i2;
// i1 and i2 are nets
assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
// Continuous assign for vector nets addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers
assign {cout, sum[3:0]} = a[3:0] + b[3:0] + cin;
// LHS is a concatenation of a scalar net and vector net
```

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;
```



Outline

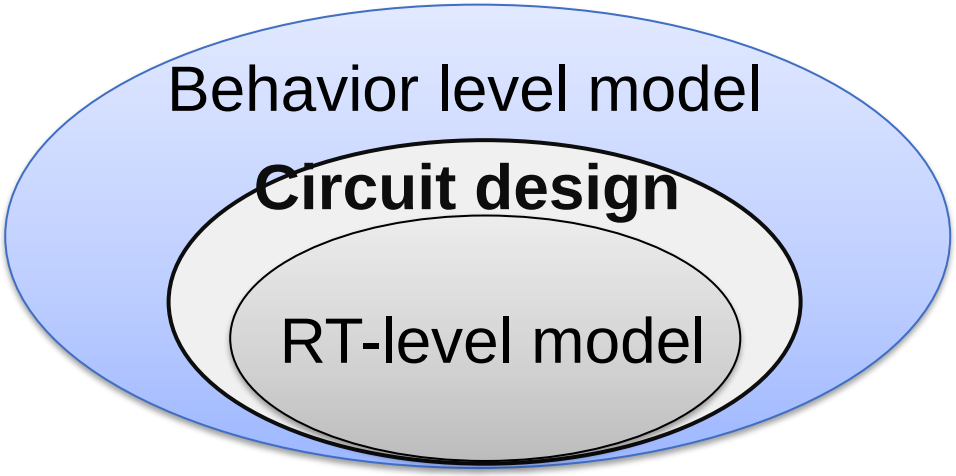
- Register-Transfer Level (RTL)
  - Definition and Verilog Syntax
  - Continuous Assignments for Combinational Circuits
- Behavior Level
  - Modeling and Event-Based Simulation
  - Procedural Construct and Assignment
  - Control Construct
- Finite State Machine (FSM)
  - Data Path Modeling
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

What is Behavioral Level

- Modeling circuits using “behavioral” descriptions and events
- Description of 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: constructs 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)**

RTL and Behavioral Level

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



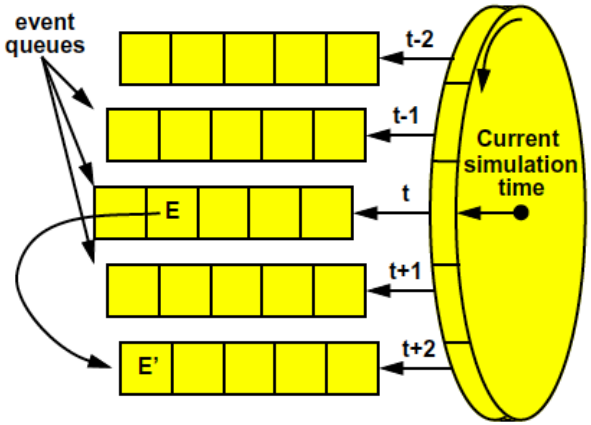
- RTL level model
  - Hardware modeling
  - Explicit structure description for modeling
  - Accurate

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 changes 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

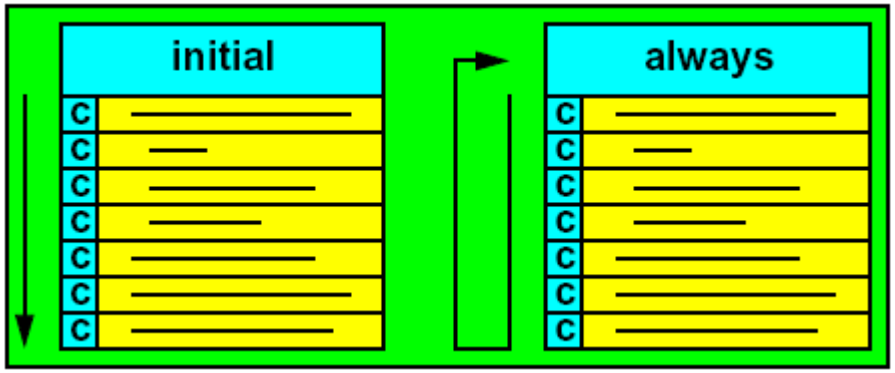
Procedural Construction

- 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)

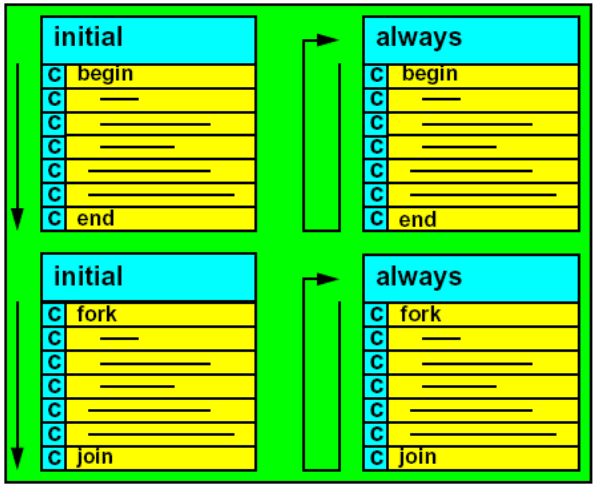
- 2 Types of blocks: **`initial`**, **`always`**
- Trigger conditions
  - **`initial`**: at the beginning of simulation (**NON**-SYNTHESIZABLE)
  - **`always`**: depends on the sensitivity list
- Components
  - Timing controls (#) (**NON**-SYNTHESIZABLE)
  - Conditional construct (if, else, case, …)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

There's no circuit can work only once then disappear



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 **`fork-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

```
// 8-b Adder
wire   [7:0] A, B;
reg    [8:0] S;

always @(*) begin
    S = {A[7],A} + {B[7],B};
end
```

```
// DFF
reg data, data_nxt;
wire clk, rst_n;

always @(posedge clk or rst_n) begin
    if (!rst_n) data <= 1'b0;
    else data <= data_nxt;
end
```

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
```

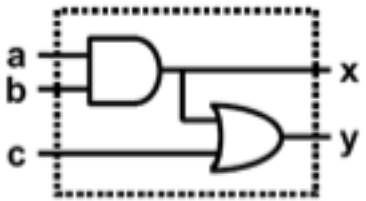
- Nonblocking assignment
  - All assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

```
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	

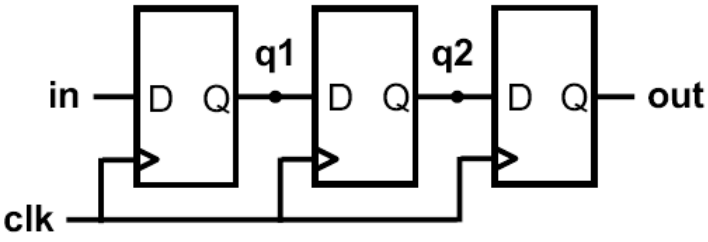
```
module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;

  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

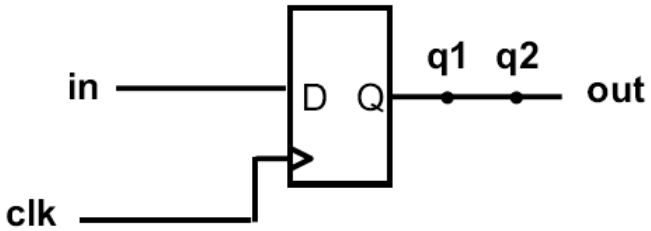
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"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*."



Conditional Statements (1/2)

- If and If-else statements

```
if (expression)
   statement
else
   statement
```

```
if (expression)
   statement
else if (expression)
   statement
else
   statement
```

```
if (sel == 0) begin
    a_w = data;
    b_w = b_r;
end
else begin
    b_w = data;
    a_w = a_r;
end
```

```
a_w = a_r;
b_w = b_r;

if (sel == 0)
    a_w = data;
else
    b_w = data;
```

```
if (sel == 0)
    a <= data;
else
    b <= data;
```

- Restrictions compared with C
  - LHS in all cases should be the same!
    - Avoid latch
  - Conditions should be full-case, if must be followed by else!
    - Avoid latch
  - In short, think about MUX!

```
if (sel == 0)
    a_w = data;
else
    b_w = data;
```

```
if (sel == 0)
    a_w = data;
```

Conditional Statements (2/2)

Example 1

```
if (sel==3)
  y = d;
else
  if (sel==2)
    y = c;
  else
    if (sel==1)
      y = b;
    else
      if (sel==0)
        y = a;
```

V.S

Example 2

```
if (sel[1])
  if (sel[0])
    y = d;
  else
    y = c;
else
  if (sel[0])
    y = b;
  else
    y = a;
```

Multiway Branching

- The nested ***if-else-if*** can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the ***case*** statement

```
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
      ...
  default: default_statement;
endcase
```

Alternative way:

```
always @(*) begin
    y = a;

    case(sel)
        3: y = d;
        2: y = c;
        1: y = b;
    endcase
end
```

NOTICE:
Not full case would cause latch

if-else statement

```
if (sel===3)
  y = d;
else
  if (sel===2)
    y = c;
  else
    if (sel===1)
      y = b;
    else
      y = a;
```

case statement

```
case (sel)
      3: y = d;
      2: y = c;
      1: y = b;
default: y = a;
endcase
```

Looping Statements

- **The for loop (conditionally synthesizable)**
- The while loop (not synthesizable)
- The repeat loop (not synthesizable)
- The forever loop (not synthesizable)

For Loop

- The keyword **`for`** is used to specify this loop. The **`for`** loop contain 3 parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable
- **Synthesizable only if the expanded form is synthesizable**
  - 沒有時間相依性
  - 不能和參數有關係

array[i+1] <= array[i] – 1; 
*array[i] = 2*i;* 

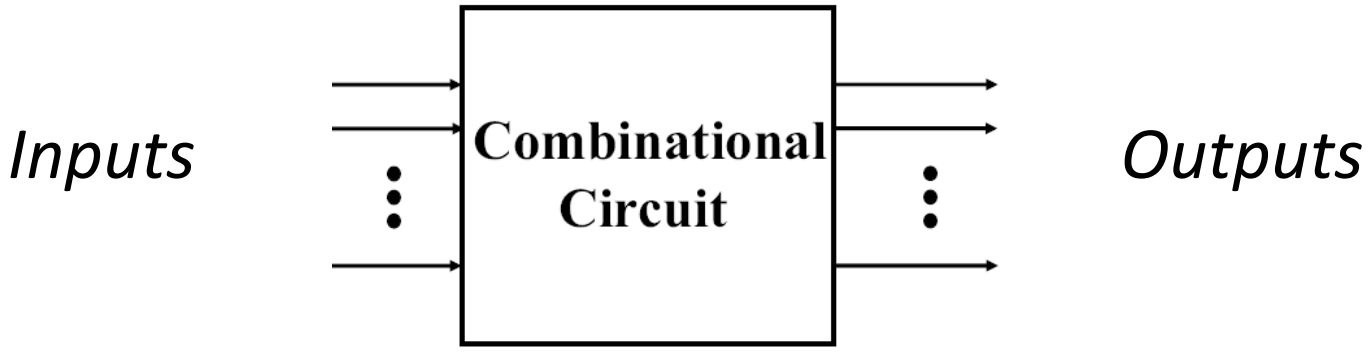
```
module bitwise_and(a, b, out);
  parameter size=2;
  input  [size-1:0] a, b;
  output [size-1:0] out;
  reg    [size-1:0] out;
  integer i;
  always @(a or b)
  begin
    for (i = 0;i < size; i = i + 1)
      out[i] = a[i] & b[i];
  end
endmodule
```

Outline

- Register-Transfer Level (RTL)
  - Definition and Verilog Syntax
  - Continuous Assignments for Combinational Circuits
- Behavior Level
  - Modeling and Event-Based Simulation
  - Procedural Construct and Assignment
  - Control Construct
- Finite State Machine (FSM)
  - Data Path Modeling
  - Moore Machine & Mealy Machine
  - Behavior Modeling of FSM

Combinational Data Path

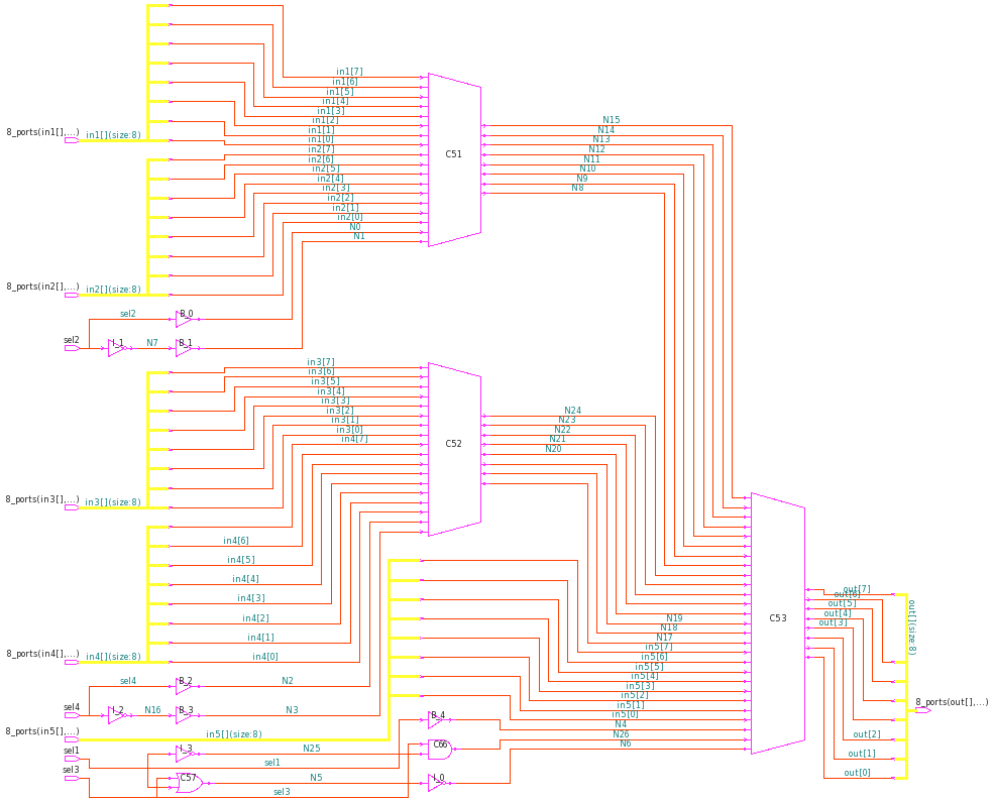
- Combinational logic circuits are **memoryless**
  - Any transition in inputs affect the whole circuit right away
  - Feedback path is not allowed
- Output can have logical transitions before settling to a stable value



Example: Mapping of *if* Statement

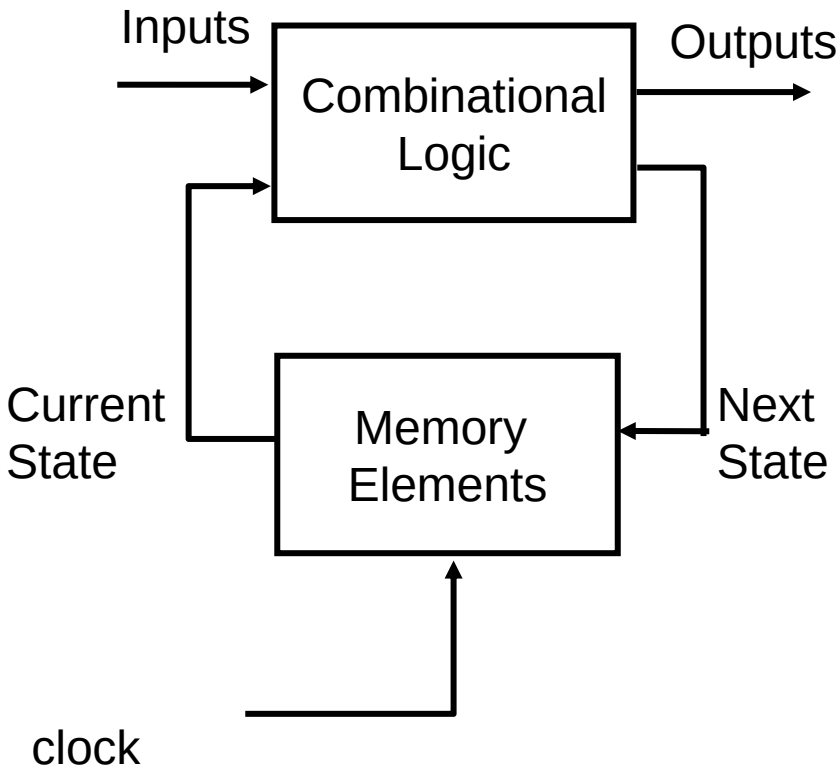
- Mapped to a Multiplexer
- *if* statement can be nested

```
always @(*) begin
    if (sel1) begin
        if (sel2) out=in1;
        else out=in2;
    end
    else if (sel3) begin
        if (sel4) out=in3;
        else out=in4;
    end
    else out=in5;
end
```



Sequential Data Path

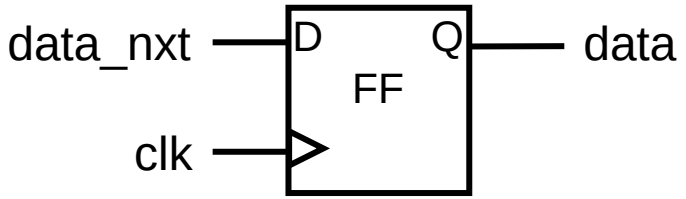
- Sequential circuits have memory (i.e. remember the past)
- The output is
  - depend on inputs
  - depend on current state
- In synchronous system
  - clock orchestrates the sequence of events
- Fundamental components
  - Combinational circuits
  - Memory elements



Example: Mapping of Sequential Circuits

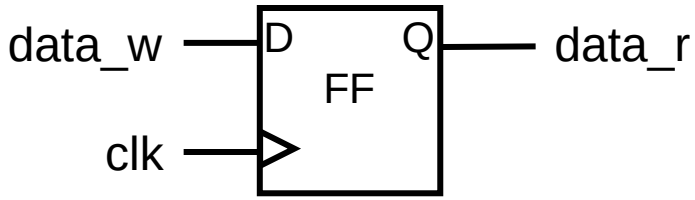
- Pure sequential circuits can be mapped as flop-flops
- The name of a flip-flop is its output port
  - Style 1

```
reg data, data_nxt;
wire clk;
always @(posedge clk)
    data <= data_nxt;
end
```



  - Style 2

```
reg data_r, data_w;
wire clk;
always @(posedge clk)
    data_r <= data_w;
end
```



Modeling of Flip-Flops (1/2)

- The use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered)
- Unlike combinational always block, the sensitivity list does determine the behavior of synthesis

D Flip-flop with synchronous clear

```
module dff_sync_clear(d, clearb,
clock, q);
input d, clearb, clock;
output q;
reg q;
always @ (posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

always block entered only at each positive clock edge

D Flip-flop with asynchronous clear

```
module dff_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;

always @ (negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

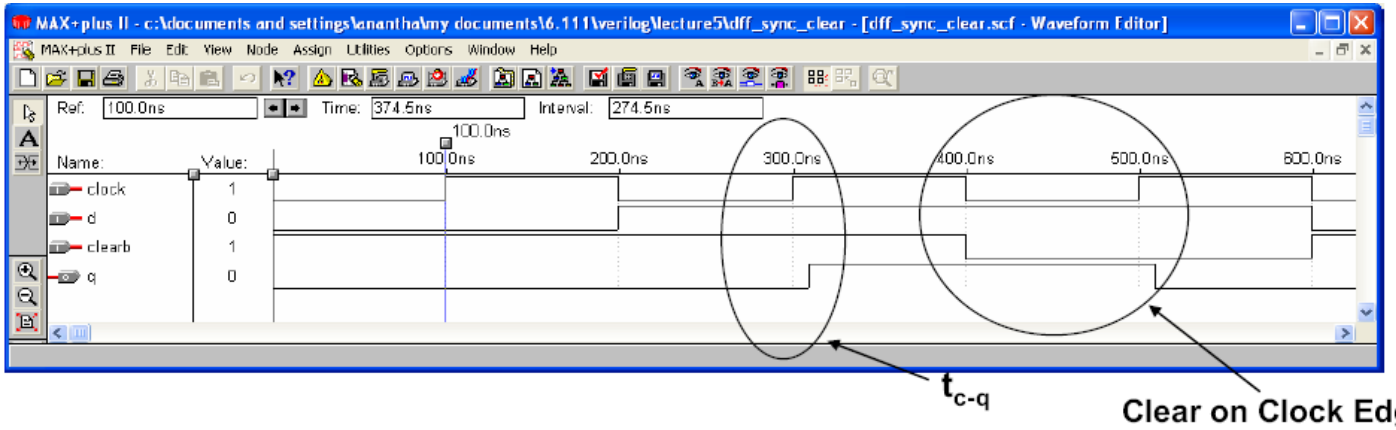
always block entered immediately when (active-low) clearb is asserted

Note: The following is **incorrect** syntax: `always @ (clear or negedge clock)`

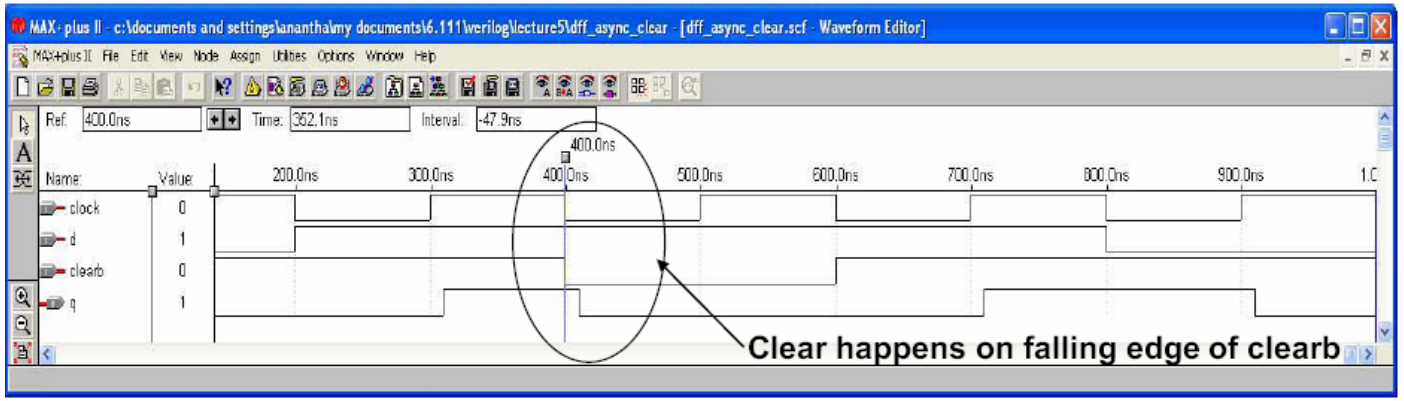
If one signal in the sensitivity list uses posedge/negedge, then all signals must.

Modeling of Flip-Flops (2/2)

- Synchronous Reset

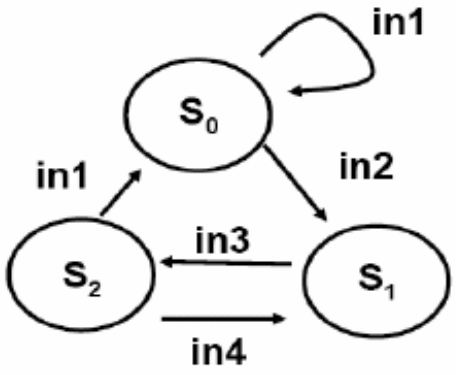


- Asynchronous Reset



Finite State Machine (FSM)

- Model of computation consisting of
  - A set (of finite number) of states
  - An initial state
  - Input symbols
  - Transition function that maps input symbols and current states to a next state



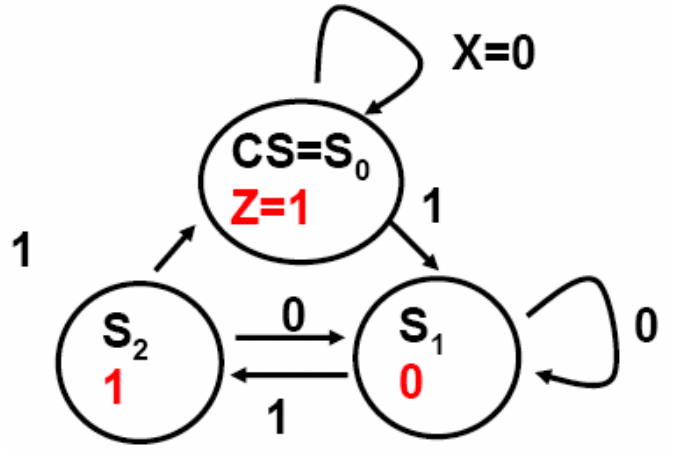
State transition diagram

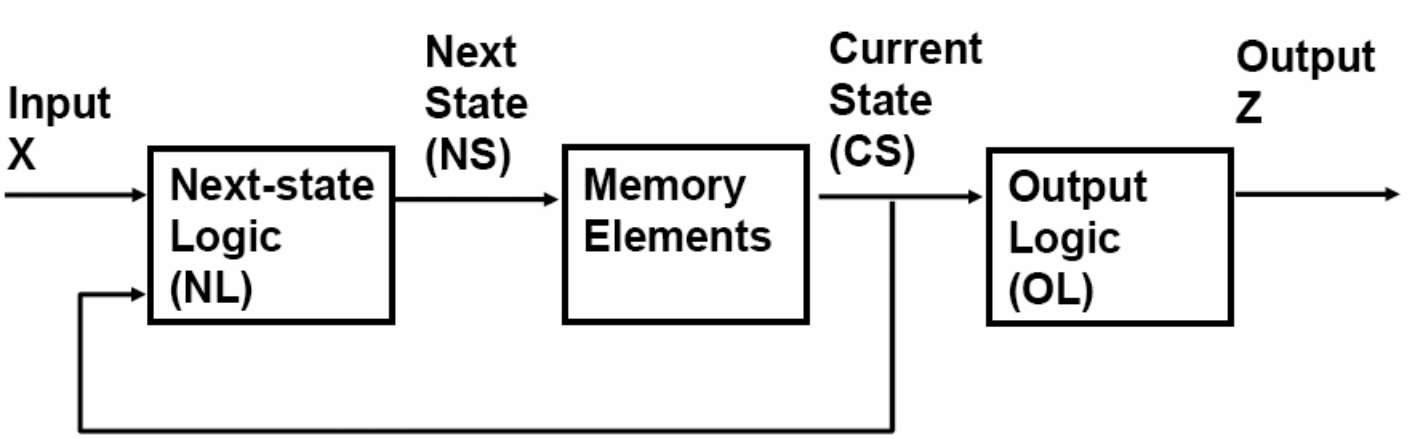
Elements of FSM

- Memory Elements
  - Memorize Current States (CS)
  - Usually consist of FF or latch
  - N-bit FF have 2^n possible states
- Next-state Logic (NL)
  - Combinational Logic
  - Produce next state
  - Based on current state (CS) and input (X)
- Output Logic (OL)
  - Combinational Logic
  - Produce outputs (Z)
    - Based on current state (Moore)
    - Based on current state and input (Mealy)

Moore Machine

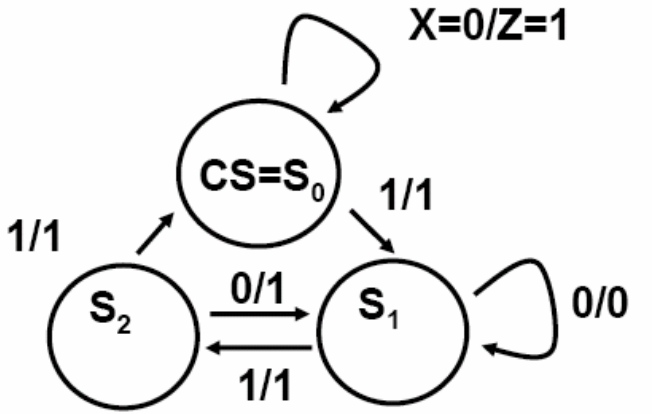
- Output is a function of
  - **Only current state**

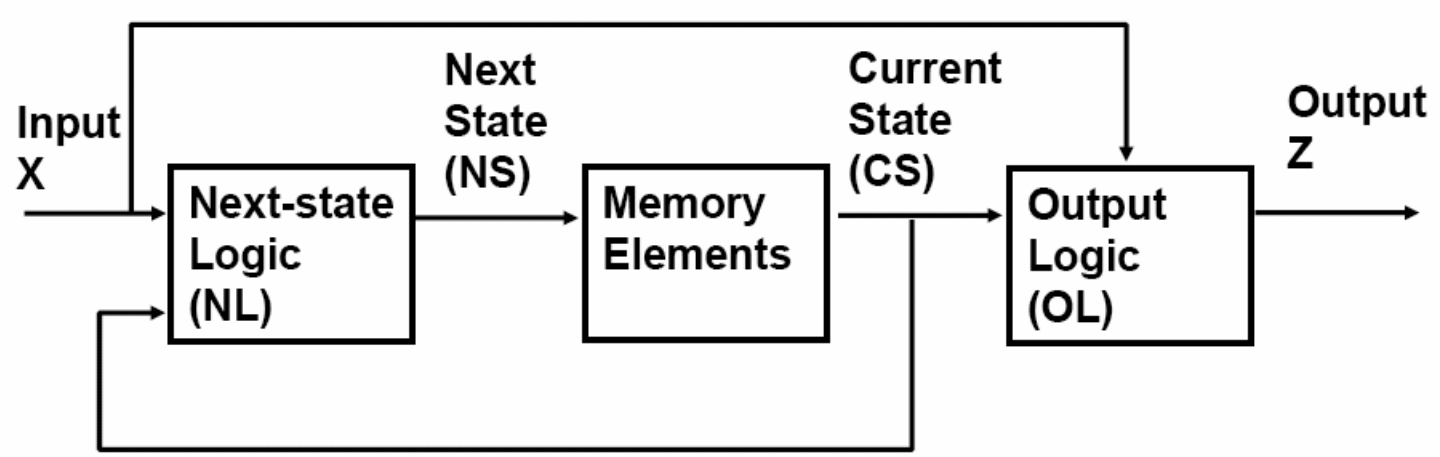




Mealy Machine

- Output is a function of
  - **Both current state & input**

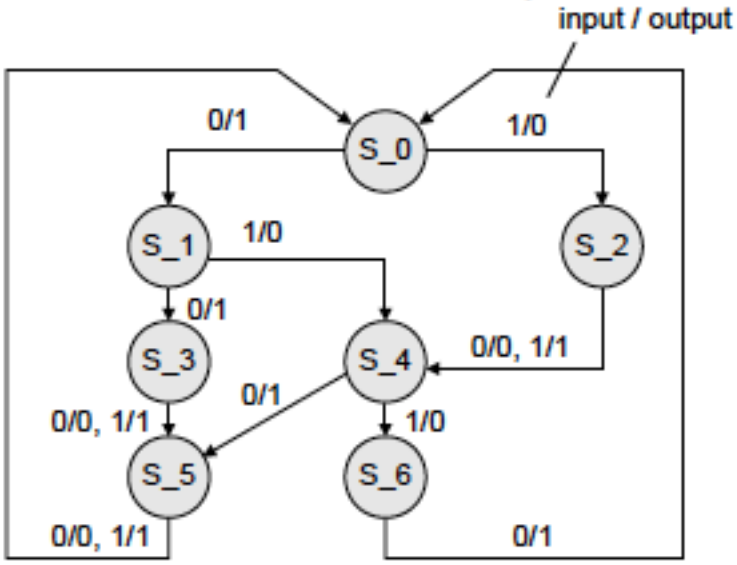




Manual Design of FSM (1/2)

- State transition graph describes the mechanism of FSM
- Then encode each state and derive the encoded table

State Transition Graph

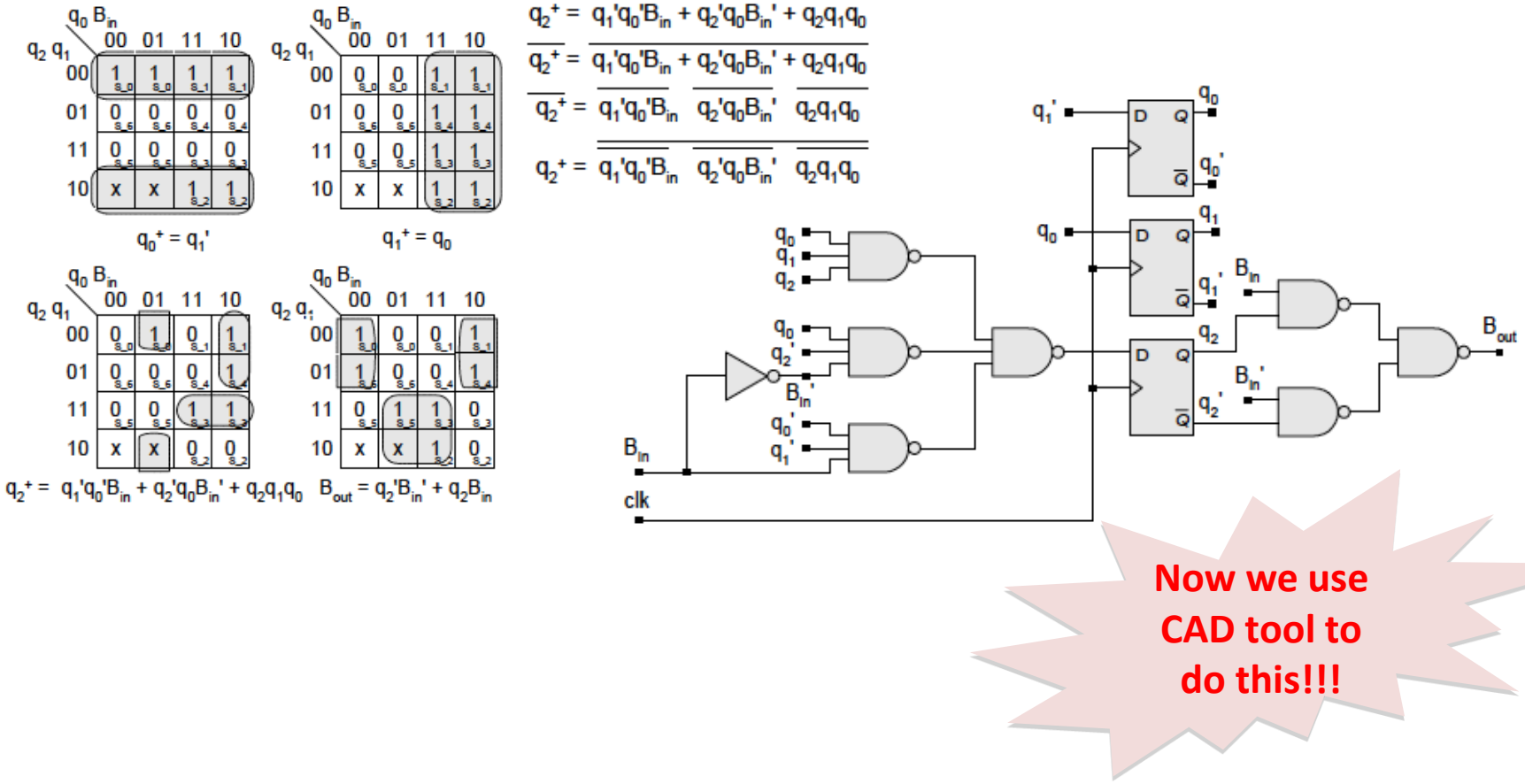


Encoded Next state/ Output Table					
	state	next state		output	
	$q_2\ q_1\ q_0$	$q_2^+\ q_1^+\ q_0^+$			
		input		input	
		0	1	0	1
S_0	000	001	101	1	0
S_1	001	111	011	1	0
S_2	101	011	011	0	1
S_3	111	110	110	0	1
S_4	011	110	010	1	0
S_5	110	000	000	0	1
S_6	010	000	-	1	-
	100	-	-		-

Now we use CAD tool to do this!!!

Manual Design of FSM (2/2)

- Use the K-Map to derive the gate-level FSM design



Behavior Modeling of FSM

- Combinational Part
  - Next-state logic (NL)
  - Output logic (OL)
- Sequential Part
  - Current state (CS) stored in flip-flops
- 3 Coding Style
  1. Separate CS, OL and NL **(Recommended)**
  2. Combine NL+ OL, separate CS
  3. Combine CS + NL, separate OL

Separate CS, OL and NL

- CS

```
always @ (posedge clk)
        current_state <= next_state;
```

- NL

```
always @ (current_state or In)
  case (current_state)
    S0: case (In)
              In0:  next_state = S1;
              In1:  next_state = S0;
              . . .
         endcase //In
    S1: . . .
    S2: . . .
 endcase //current_state
```

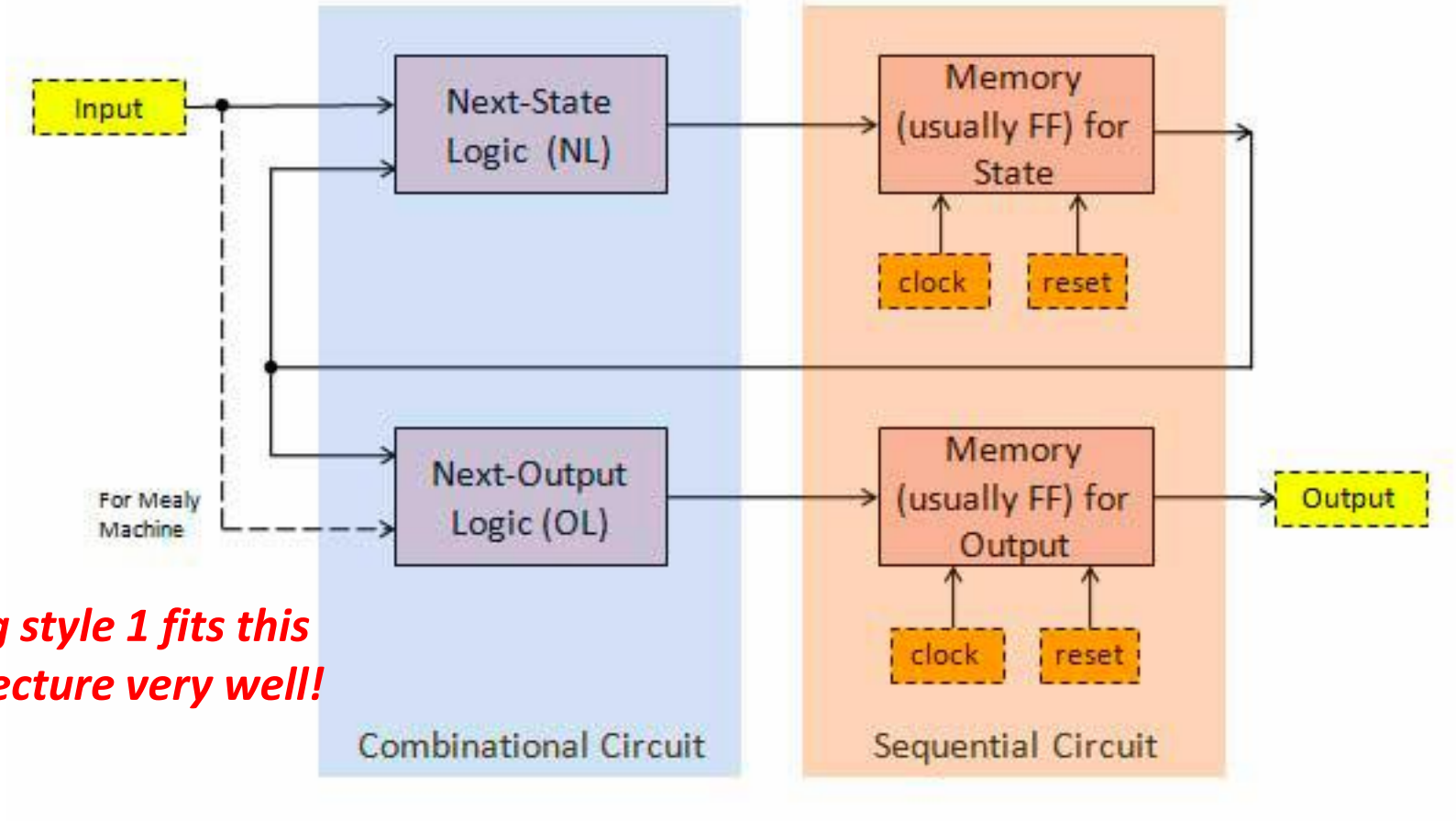
- OL

```
// if Moore
always @ (current_state)
        Z = output_value;
```

```
// if Mealy
always @ (current_state or In)
        Z = output_value;
```

Architecture of FSM

Build combinational and sequential parts separately!

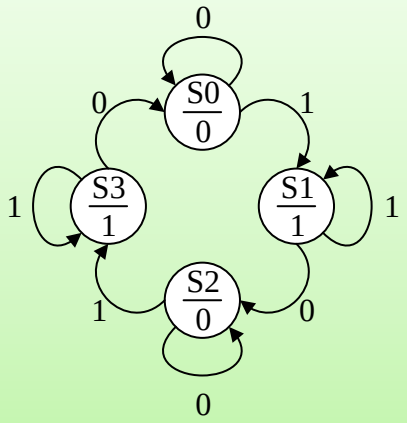


Coding style 1 fits this architecture very well!

Example

- It's a Moore machine
- Initial state is S0

```
module FSM(clk, rst_n, A, Z);
    input  clk, rst_n, A;
    output Z;
    localparam S0 = 2'd0; localparam S1 = 2'd1;
    localparam S2 = 2'd2; localparam S3 = 2'd3;
    reg [1:0] state, state_nxt;
    assign Z = (state == S1) || (state == S3);
    always @(*) begin
        case (state)
            S0: state_nxt = A ? S1 : S0;
            S1: state_nxt = A ? S1 : S2;
            S2: state_nxt = A ? S3 : S2;
            S3: state_nxt = A ? S3 : S0;
        endcase
    end
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) state <= S0;
        else        state <= state_nxt;
    end
endmodule
```



State transition graph

FSM Design Notice

- Partition FSM and non-FSM logic
  - Memory elements
  - Next-state logic
  - Output logic
- Partition combinational part and sequential part
- Use `localparam` or `parameter` to define names of the state vector
- Assign a default (reset) state

Simulation and Verification

Materials modified from

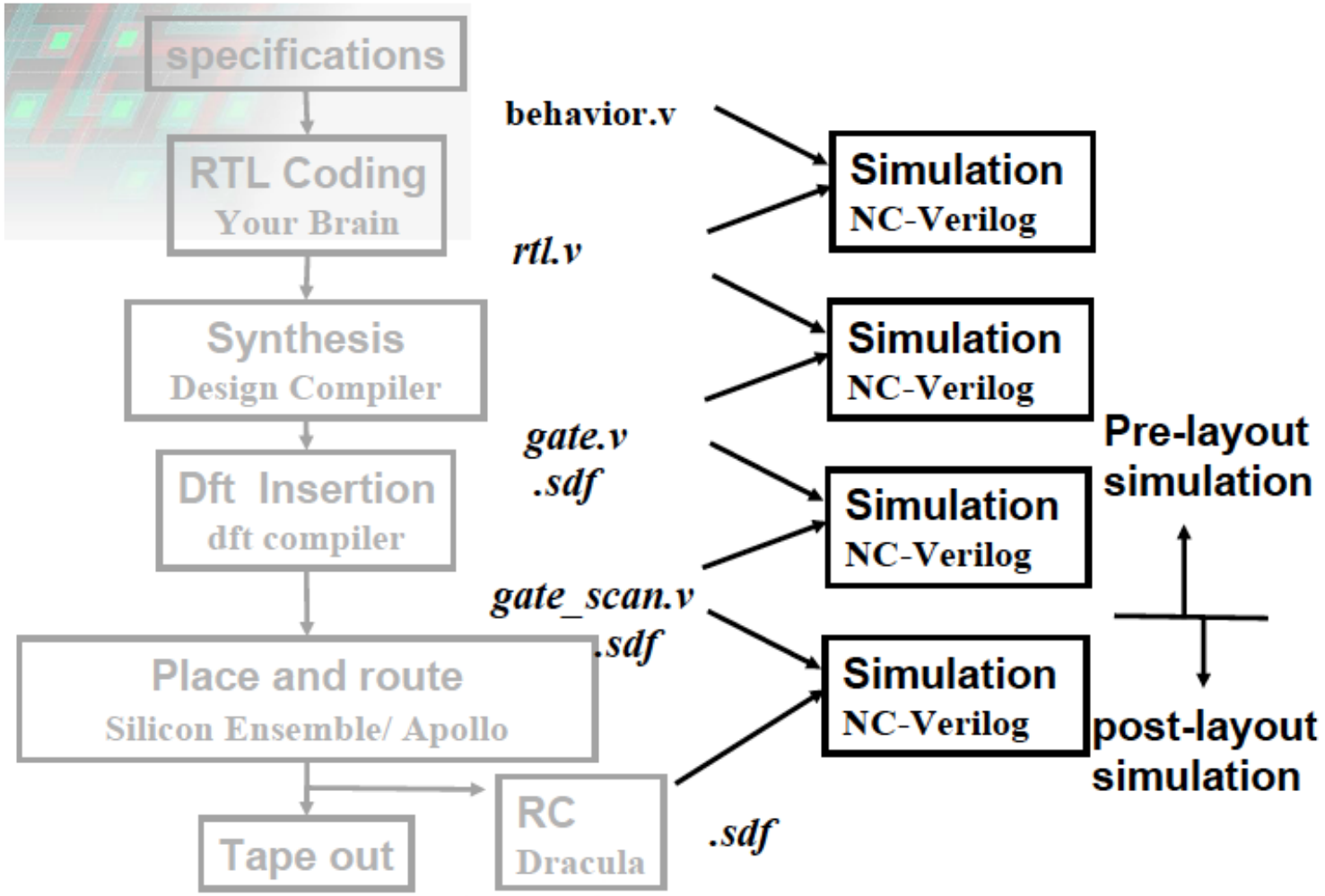
- Computer-Aided VLSI System Design
- Introduction to VLSI Design
- https://www.chipverify.com/verilog/verilog-tutorial

Outline

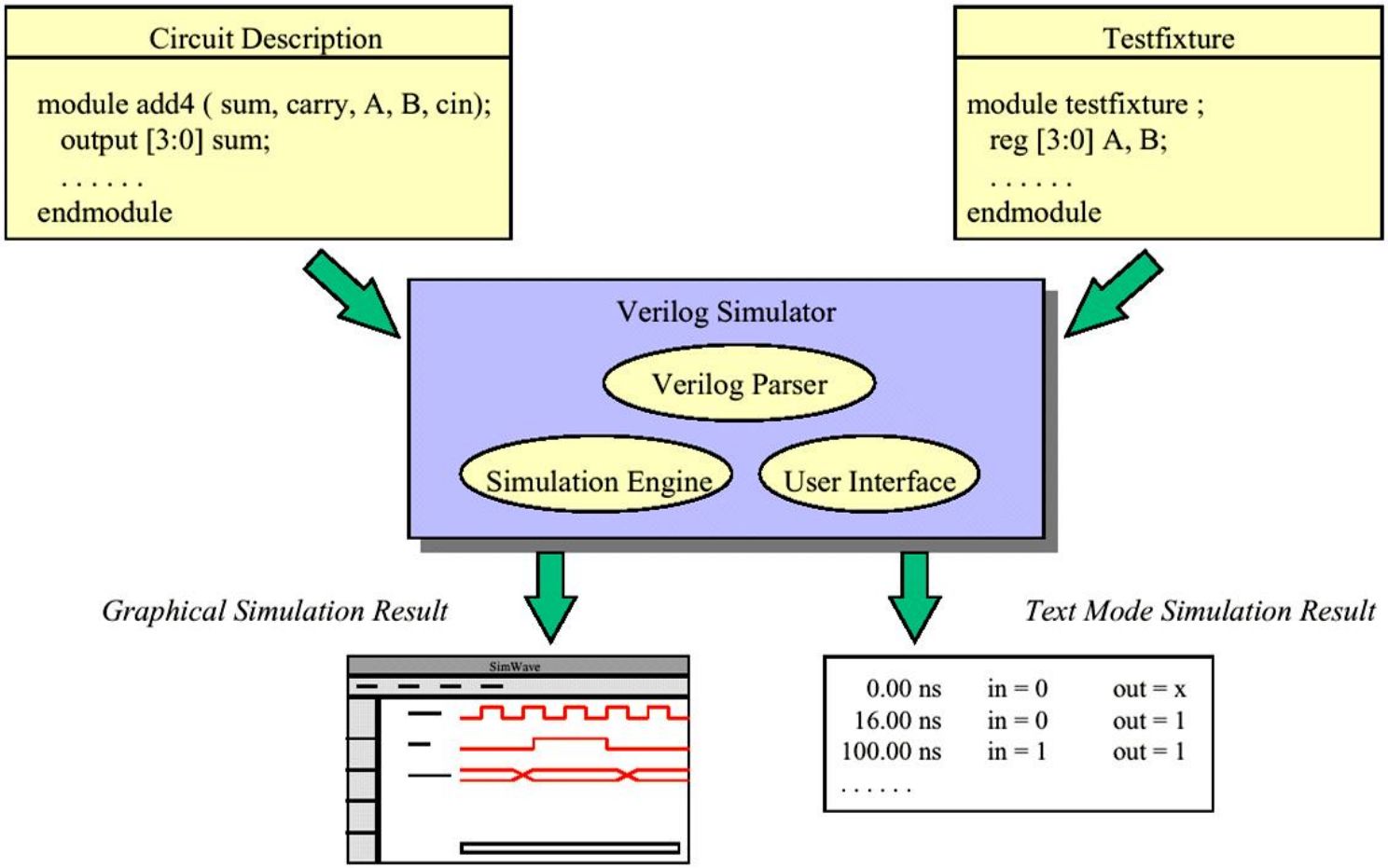
- Overview of simulation
- Testbench Examples
  - Instantiating DUT
  - Modeling Delay
  - Applying Stimulus
  - Verification
- Debugging Tool
  - Introduction to nWave
  - Tips for Debugging

Overview of Simulation

- Verification at every step



Verilog Simulator



A Common Testbench Layout

```
`timescale 1ns/10ps
`define CYCLE_TIME 10.0
// Other macros

module testbench();
    reg clk, rst_n;
    // Other wire/reg/integer

    // Other parameters

    // DUT instantiation
    MODULE_NAME DUT(
        // Port list
    );
```

```
    // Clock waveform definition
    initial begin
        clk    = 1;
        rst_n  = 1;

        #(`CYCLE_TIME*0.5);
        rst_n = 0;
        #(`CYCLE_TIME*2);
        rst_n = 1;
    end
    always #(`CYCLE_TIME*0.5)
        clk = ~clk;

    // Stimulus blocks

    // Verification blocks

endmodule
```

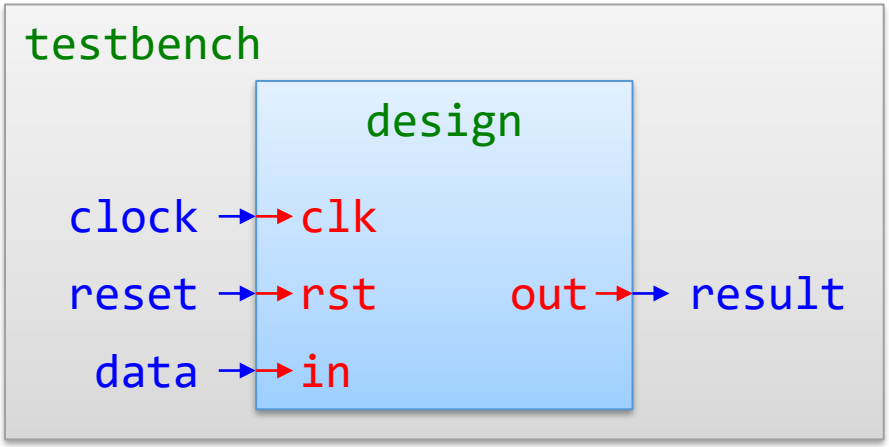
Outline

- Overview of simulation
- Testbench Examples
  - Instantiating DUT
  - Modeling Delay
  - Applying Stimulus
  - Verification
- Debugging Tool
  - Introduction to nWave
  - Tips for Debugging

Instantiating DUT (1/2)

- Device Under Test (DUT)
  - Top module of the design should be instantiated inside the testbench

```
module testbench;
    reg  clock, reset, data;
    wire result;
    design u_design(
        .clk(clock),
        .rst(reset),
        .in(data),
        .out(result)
    );
    // ......
```



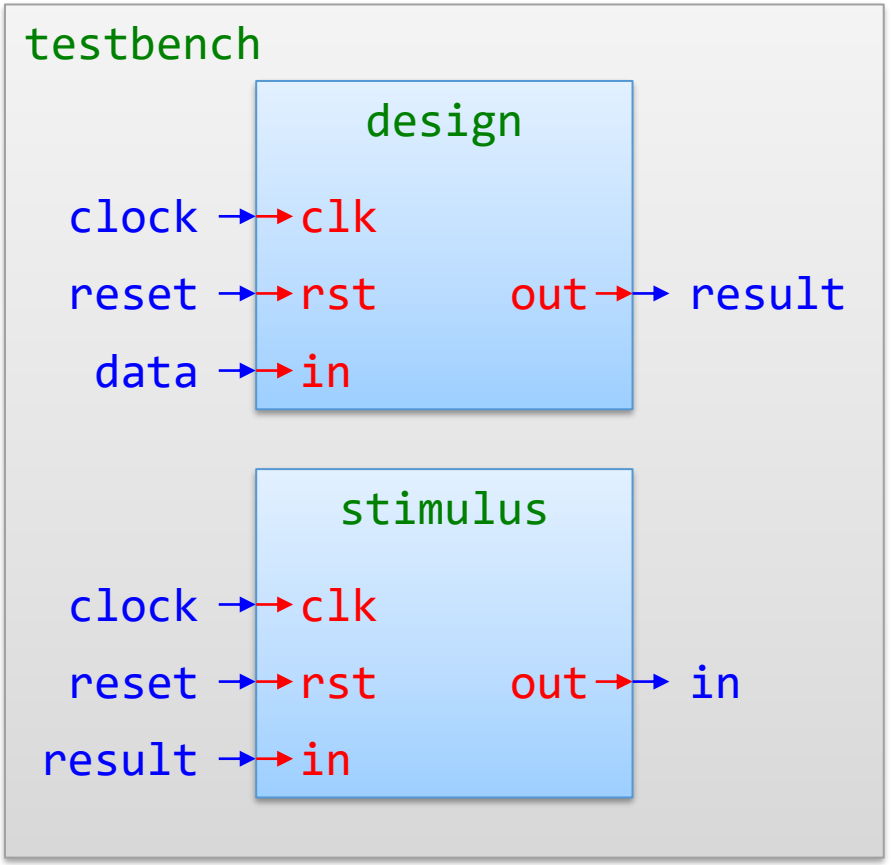
Must be wire!
Think of
`assign result = u_design.out`

Instantiating DUT (2/2)

- Device Under Test (DUT)
  - Top module of the design should be instantiated inside the testbench

```
module testbench;
    reg  clock, reset;
    wire data, result;
    design u_design(
        .clk(clock),
        .rst(reset),
        .in(data),
        .out(result)
    );
    stimulus u_stimulus(
        .clk(clock),
        .rst(reset),
        .out(data),
        .in(result)
    );
    // ......
```

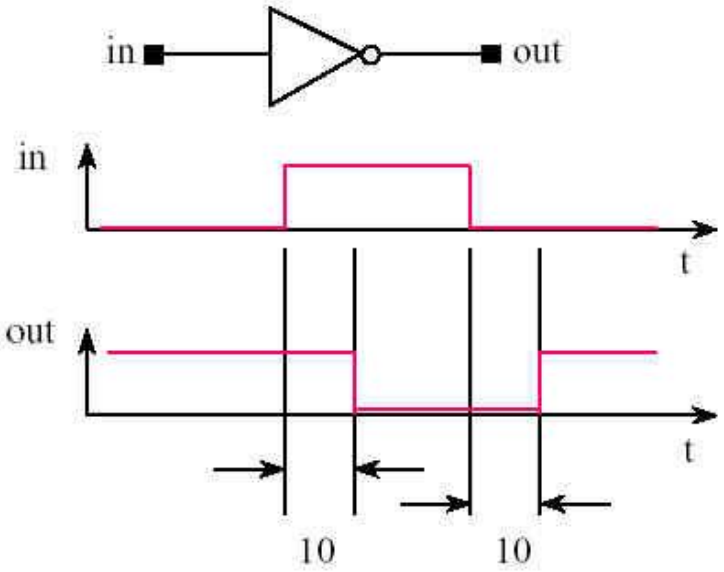
wire



Modeling Delay (1/2)

- To model these delay, we use timing / delay description in Verilog: #
- Delay specification can define propagation delay of primitive gates
  - Example

not #10 (out,in);



Modeling Delay (2/2)

- Delay specification can define propagation delay of wires and continuous assignments
  - Regular Assignment Delay

```
wire out;
assign #10 out = in1 & in2;
```

  - Implicit Continuous Assignment Delay

```
wire #10 out = in1 & in2;
```

  - Net Declaration Delay

```
wire # 10 out;
assign out = in1 & in2;
```

- Delay specification can define timing of stimuli
  - See next page

Clock and Reset Signals

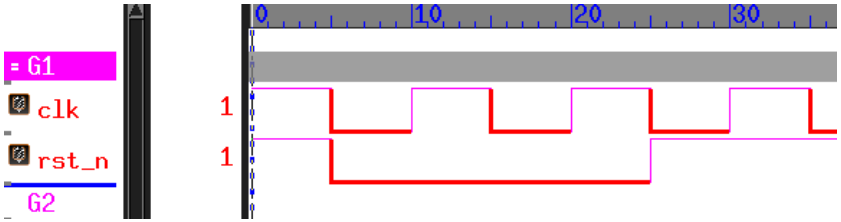
- Initialization
  - Negedge reset

```
reg clk, rst_n;
initial begin
    clk    = 1;
    rst_n  = 1;
    #(`CYCLE_TIME*0.5);
    rst_n = 0;
    #(`CYCLE_TIME*2);
    rst_n = 1;
end
```

- Define clock signal

```
always #(`CYCLE_TIME*0.5)
    clk = ~clk;
```

- Waveform
  - `define CYCLE_TIME 10



- Set a finish condition

```
initial begin
    #(`CYCLE_TIME*1000)
    $display("Simulation time longer than expected");
    $finish;
end
```

Input Test Pattern (1/3)

- Directly write in testbench
  - Fixed pattern

```
reg       clk    ;
reg [7:0] data_in;

initial clk = 1;
always #(`CYCLE_TIME*0.5)
    clk = ~clk;

initial begin
    // Change input at negedge
    #(`CYCLE_TIME*5.5)
    data_in = 8'd100;
    #(`CYCLE_TIME)
    data_in = 8'd101;
end
```

  - Random pattern

```
reg       clk;
reg [7:0] data_in;
integer   i;
parameter tot = 100;

initial clk = 1;
always #(`CYCLE_TIME*0.5)
    clk = ~clk;

initial begin
    #(`CYCLE_TIME*4.5)
    for (i=0;i<tot;i=i+1) begin
        #(`CYCLE_TIME)
        data_in = $random;
    end
end
```

Input Test Pattern (2/3)

- Test pattern files with system tasks
  - *$readmemb, $readmemh*

```
`timescale 1ns/10ps
`define CYCLE_TIME 10.0
`define INPUT_FILE "input.txt"

module test;
reg [7:0] data_in;
reg [7:0] data_arr [0:7];
integer   i;

initial begin
    $readmemb(`INPUT_FILE,data_arr);
    #(`CYCLE_TIME*4.5);
    for (i=0;i<8;i=i+1) begin
        #(`CYCLE_TIME) data_in = data_arr[i];
    end
    $finish;
end
endmodule
```

Input Test Pattern (3/3)

- Test pattern files with system tasks
  - *$fopen, $fscanf, $fclose*

```
module test;
reg        clk;
reg [7:0] data_in;
reg [7:0] data_arr [0:7];
integer    i, file_r;
initial clk = 1;
always #(`CYCLE_TIME*0.5) clk = ~clk;
initial begin
    file_r = $fopen(`INPUT_FILE,"r");
    #(`CYCLE_TIME*4.5)
    for (i=0;i<8;i=i+1) begin
        #(`CYCLE_TIME)
        $fscanf(file_r,"%d",data_in);
    end
    $fclose(file_r);
    $finish;
end
endmodule
```

Verification

- When to evaluate your design?
  - At the instant when output is valid
  - Load the result to an array in advance
  - Write a file and verify by other software such as Python

- How to evaluate your design?
  (At least, some metrics can be verified in RTL phase)
  - Usually, check the result with a prepared golden file
  - Identical results to golden
  - Relaxed results to golden
    - SNR
  - Latency (number of delayed cycles)

Verification Example (1/2)

```
module test;
parameter tot = 8;
reg  [7:0] data_in;
wire [7:0] data_out;
reg  [7:0] data_arr [0:tot-1];
reg  [7:0] ans_arr  [0:tot-1];
integer i,j,err;
my_inv dut(
    .in(data_in),
    .out(data_out)
);
```

```
initial begin
    $readmemh("input.txt",data_arr);
    for (i=0;i<tot;i=i+1) begin
        data_in = data_arr[i];
        #(`CYCLE_TIME);
    end
end
initial begin
    $readmemh("output.txt",ans_arr);
    err = 0;
    #(`CYCLE_TIME*0.5);
    for (j=0;j<tot;j=j+1) begin
        if (data_out !== ans_arr[j])
            err = err+1;
        #(`CYCLE_TIME);
    end
    if (err == 0) $display("Success");
    else          $display("Fail");
    $finish;
end
endmodule
```

Verification Example (2/2)

```
module test;
parameter tot = 8;
reg signed [7:0] data_in;
wire signed [7:0] data_out;
reg signed [7:0] data_arr [0:tot-1];
reg signed [7:0] ans_arr [0:tot-1];
integer signal,diff,energy,noise;
integer i, j;
real SNR;
my_design dut(
    .in(data_in),
    .out(data_out)
);

initial begin
    $readmemh("input.txt",data_arr);
    for (i=0;i<tot;i=i+1) begin
        data_in = data_arr[i];
        #(`CYCLE_TIME);
    end
end
```

```
initial begin
    $readmemh("output.txt",ans_arr);
    energy = 0;
    noise  = 0;
    #(`CYCLE_TIME*0.5);
    for (j=0;j<tot;j=j+1) begin
        signal = ans_arr[i];
        diff = ans_arr[i]-data_out;
        energy = energy
               + (signal*signal);
        noise  = noise
               + (diff*diff);
        #(`CYCLE_TIME);
    end
    SNR = 10*$log10(energy/noise);
    if (SNR > 40)
        $display("Success");
    else
        $display("Fail"   );
    $finish;
end
endmodule
```

Other Tips

- Use === and !== in testbench for equivalence check
  - Comparisons with x/z are always false
  - In this case, if output_data is always x, errors will still be 0:

```
if (output_data != output_golden) begin
    err = err + 1;
end
```

- Use . to access members of lower level

```
module tb;
    behav_ram i_mem(
        // ...
    );
    initial begin
        $readmemh("data.mem",
                  i_mem.mem);
    end
endmodule
```

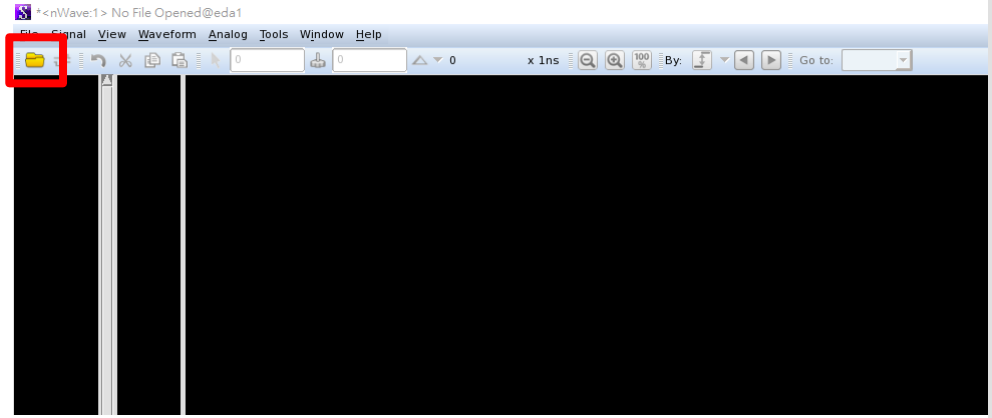
```
module behav_ram(
    // ...
);
    reg [31:0] mem [0:32767];
endmodule
```

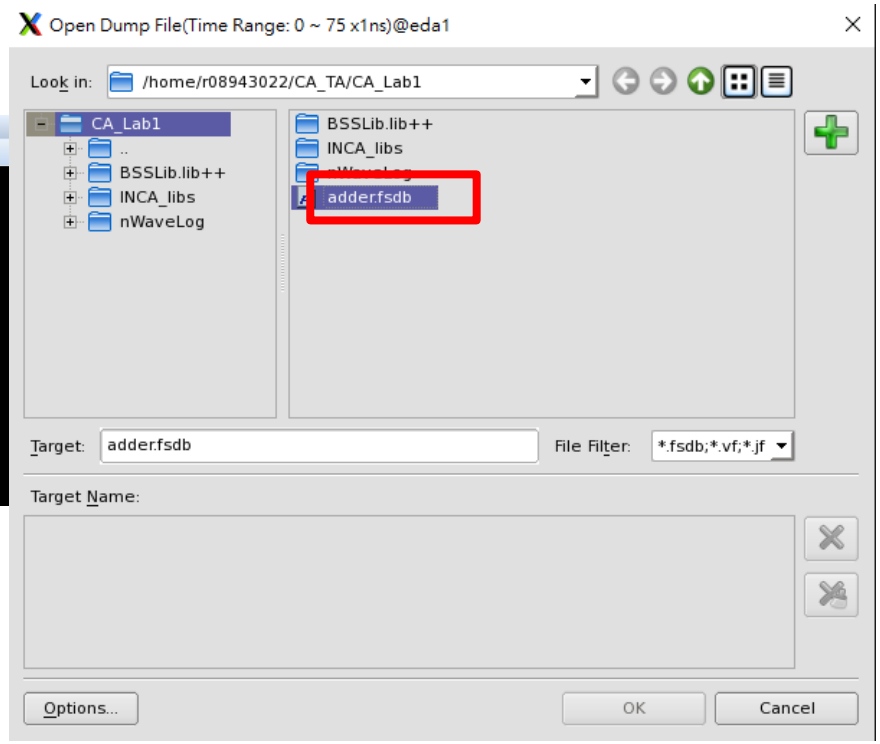
Outline

- Overview of simulation
- Testbench Examples
  - Instantiating DUT
  - Modeling Delay
  - Applying Stimulus
  - Verification
- Debugging Tool
  - Introduction to nWave
  - Tips for Debugging

Introduction to nWave (1/3)

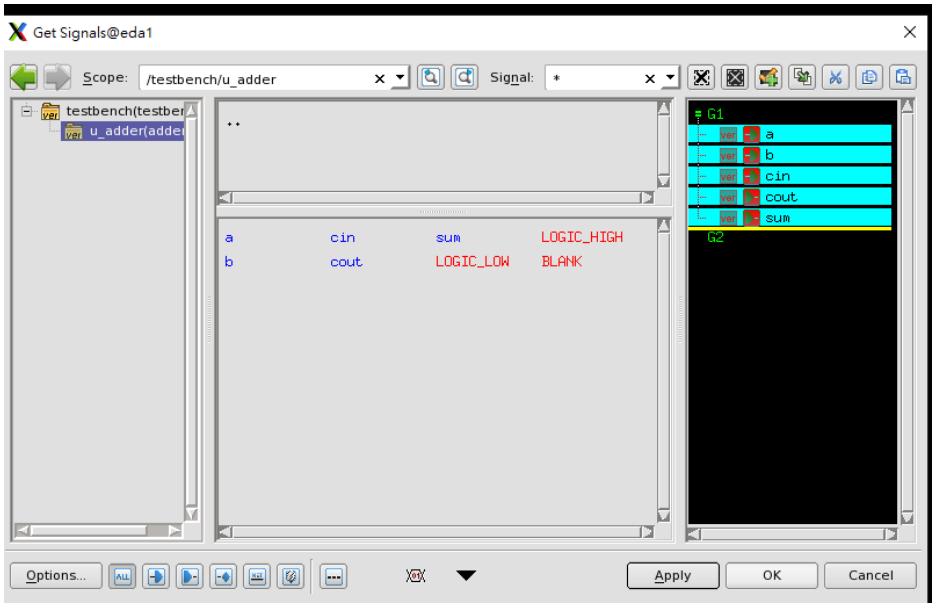
- $ nWave &

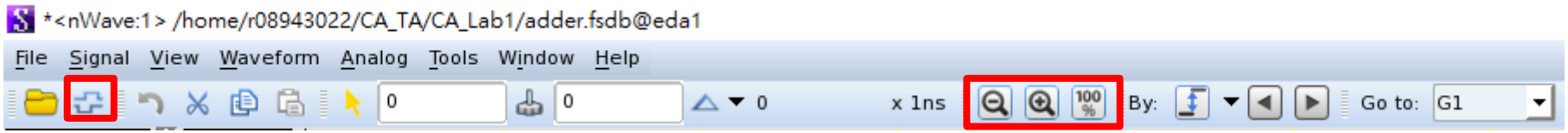




Introduction to nWave (2/3)

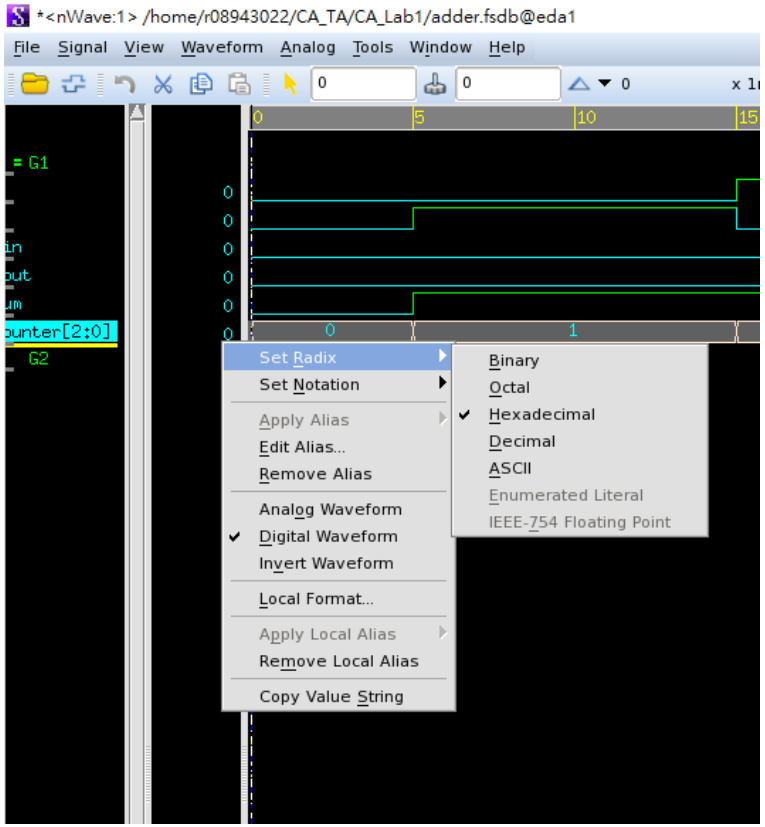
- Get signals
- Use the scroll wheel button to sort the signals
- Click 100% button

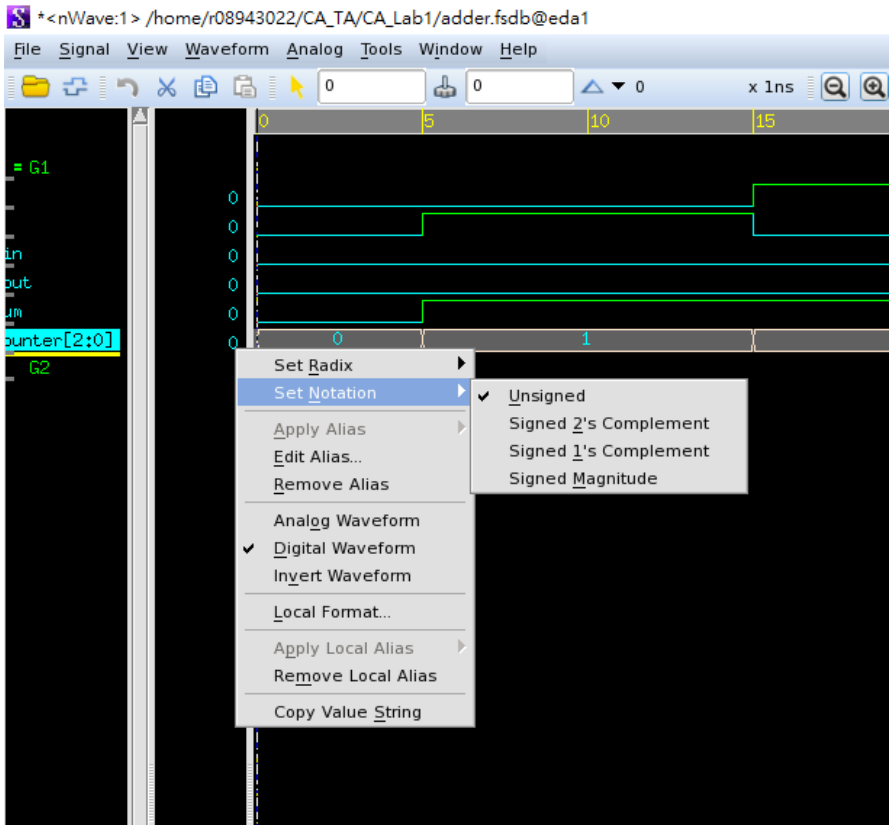




Introduction to nWave (3/3)

- Right click to set radix and notation





Tips for Debugging

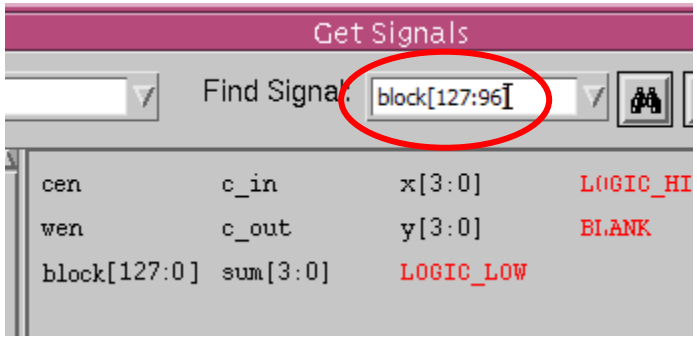
- Open clock and reset waveforms
- Open other interested waveforms
- Change to interested waveform format
- Search interested edge by 

- Update waveforms after a new simulation
  - Hotkey: ctrl+L

- If needed, modify part of the testbench code
  - *$display* some interested information

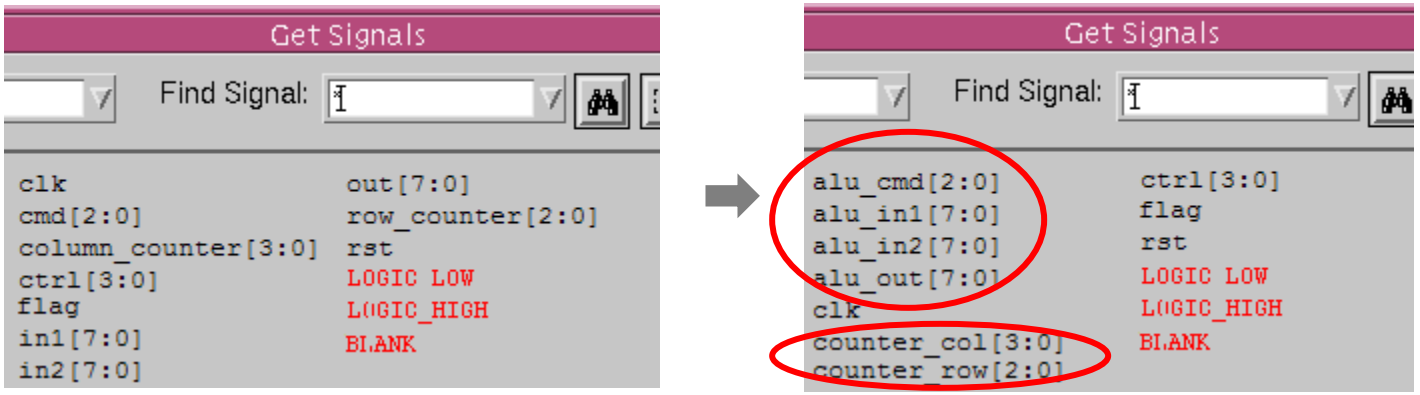
RTL Debugging

- Backward trace
  - When *S* is incorrect, see what drives *S* first
  - Backward trace until the source of error is found
- Check control signals then data flow
  - Again, wrong control signals won't let data flow correct!
- Partial vector signal in nWave



Easy-Debugging Coding Style (1/2)

- Signal naming with prefix & suffix
  - Utilize the prefix & suffix to show the attributes of signals
  - signal_n: low-active
  - signal_w: wire
  - signal_r: register/flip-flop
  - next_signal: next-state signal of FSM
  - cur_signal: current-state signal of FSM
- Alphabetically naming for waveform debugging



Easy-Debugging Coding Style (2/2)

- Pure sequential block

```
always@(posedge clk) begin
    if(state==2'd0) begin
        state <= 2'd1;
    end
    else if(state==2'd1) begin
        state <= 2'd2;
    end
    else if(state==2'd2) begin
        if(flag) state <= 2'd3;
        else state <= state;
    end
    else begin
        state <= 2'd0;
    end
end
```

Not Recommended

```
always@(*) begin
    if(state==2'd0) begin
        next_state = 2'd1;
    end
    else if(state==2'd1) begin
        next_state = 2'd2;
    end
    else if(state==2'd2) begin
        if(flag) next_state = 2'd3;
        else next_state = state;
    end
    else begin
        next_state = 2'd0;
    end
end
always@(posedge clk) begin
    state <= next_state;
end
```

Recommended