Rapid Prototyping & Layout 聯盟
教育部顧問室「超大型積體電路與系統設計」教育改進計劃

Verilog 教學改進計劃

南台科技大學 電子系
王立洋 博士

編撰者：

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫P&L聯盟

Verilog-HDL Overview

編撰者：王立洋

Outline

Introduction

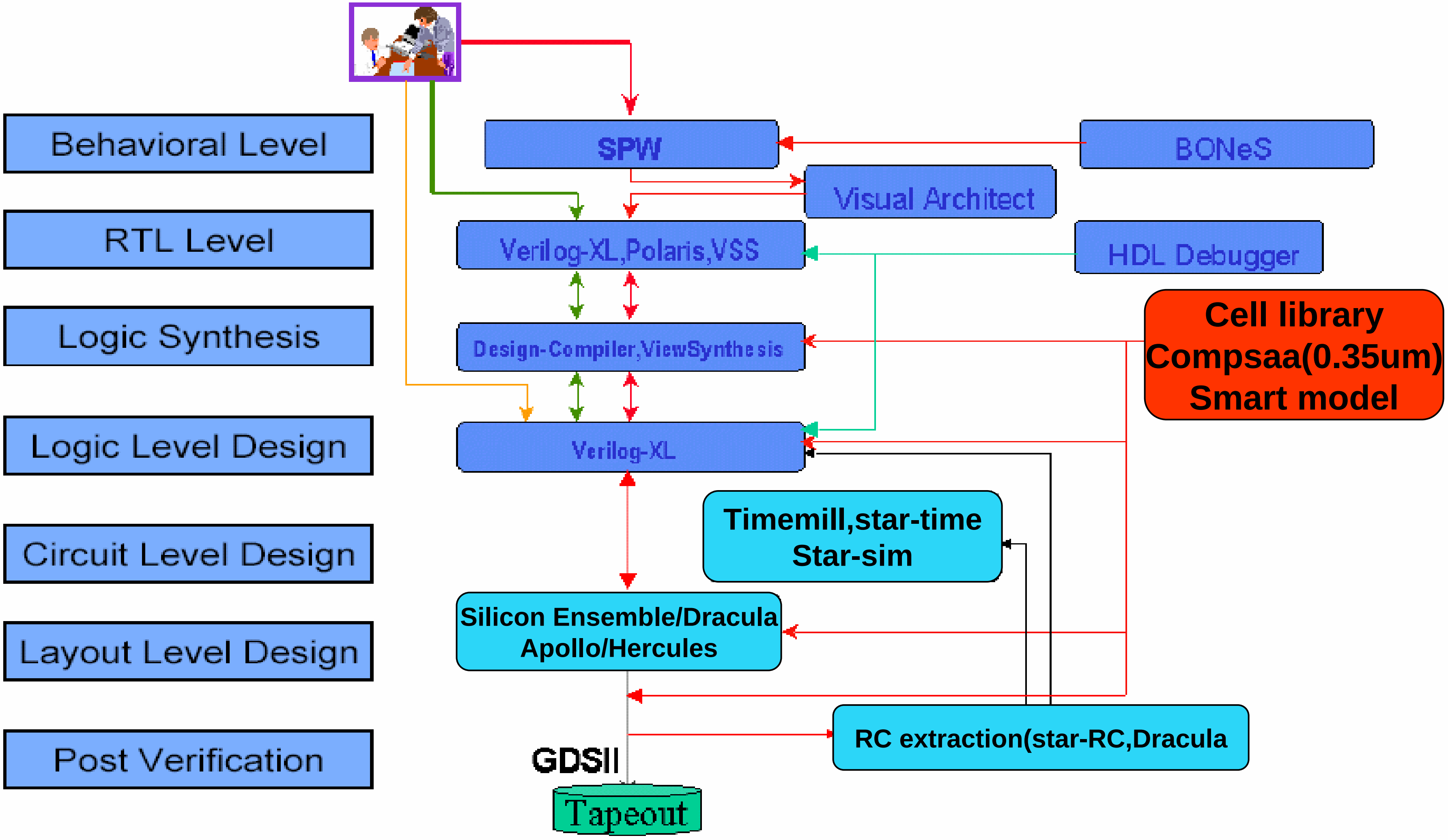
What is Verilog HDL ?

- High-level programming language constructs
- Describe the functionality of the devices model
- Hardware description language that allows you to describe circuits at different levels of abstractions and allow you to mix any level of abstraction in the design.

Verilog History

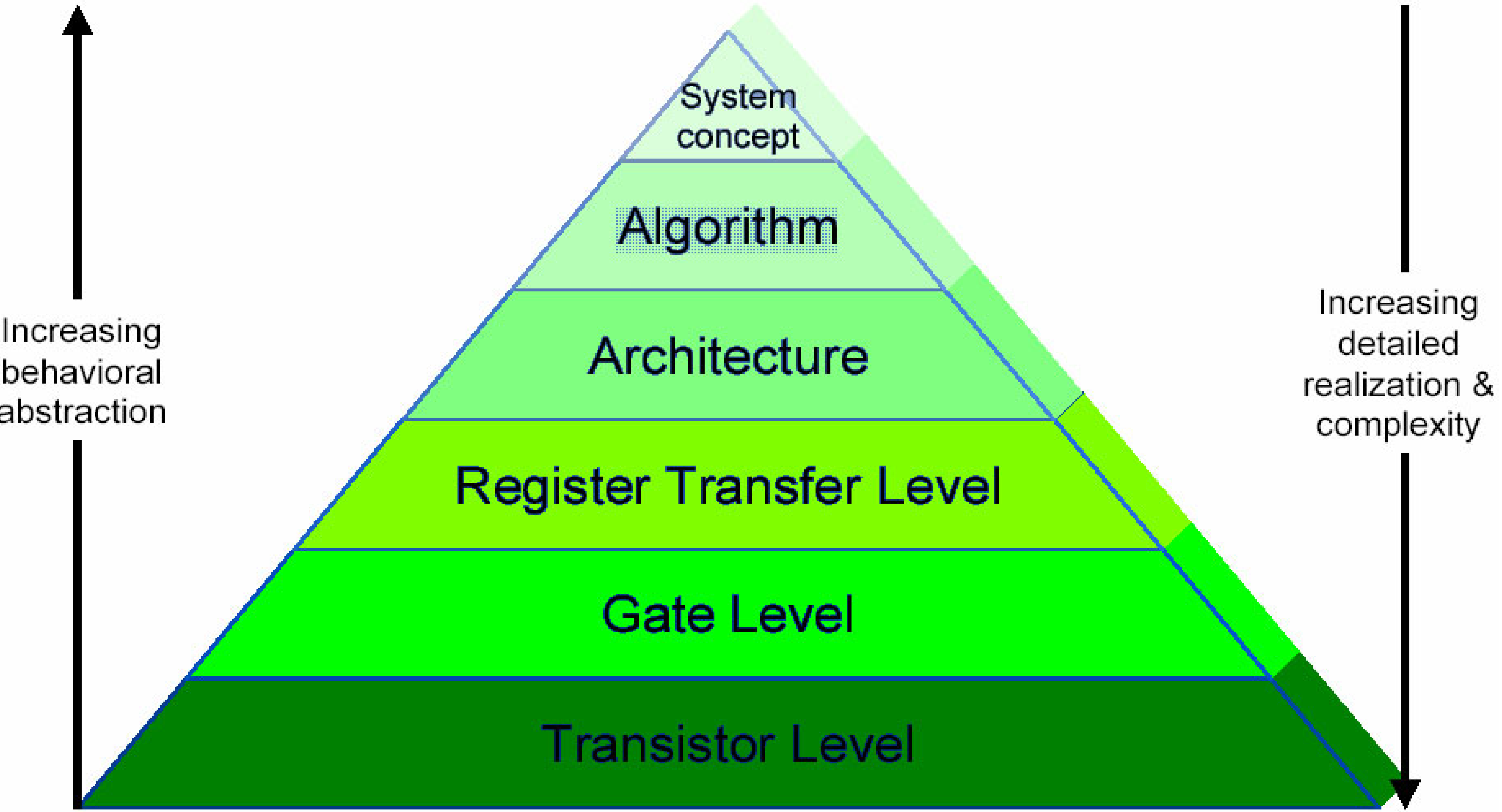
- Verilog was written by Gateway Design Automation in the early 1980
- Cadence acquired Gateway in 1990
- Cadence released Verilog to the pubic domain in 1991
- In 1995 the language was ratified as IEEE standard 1364

Cell Based Design Flow

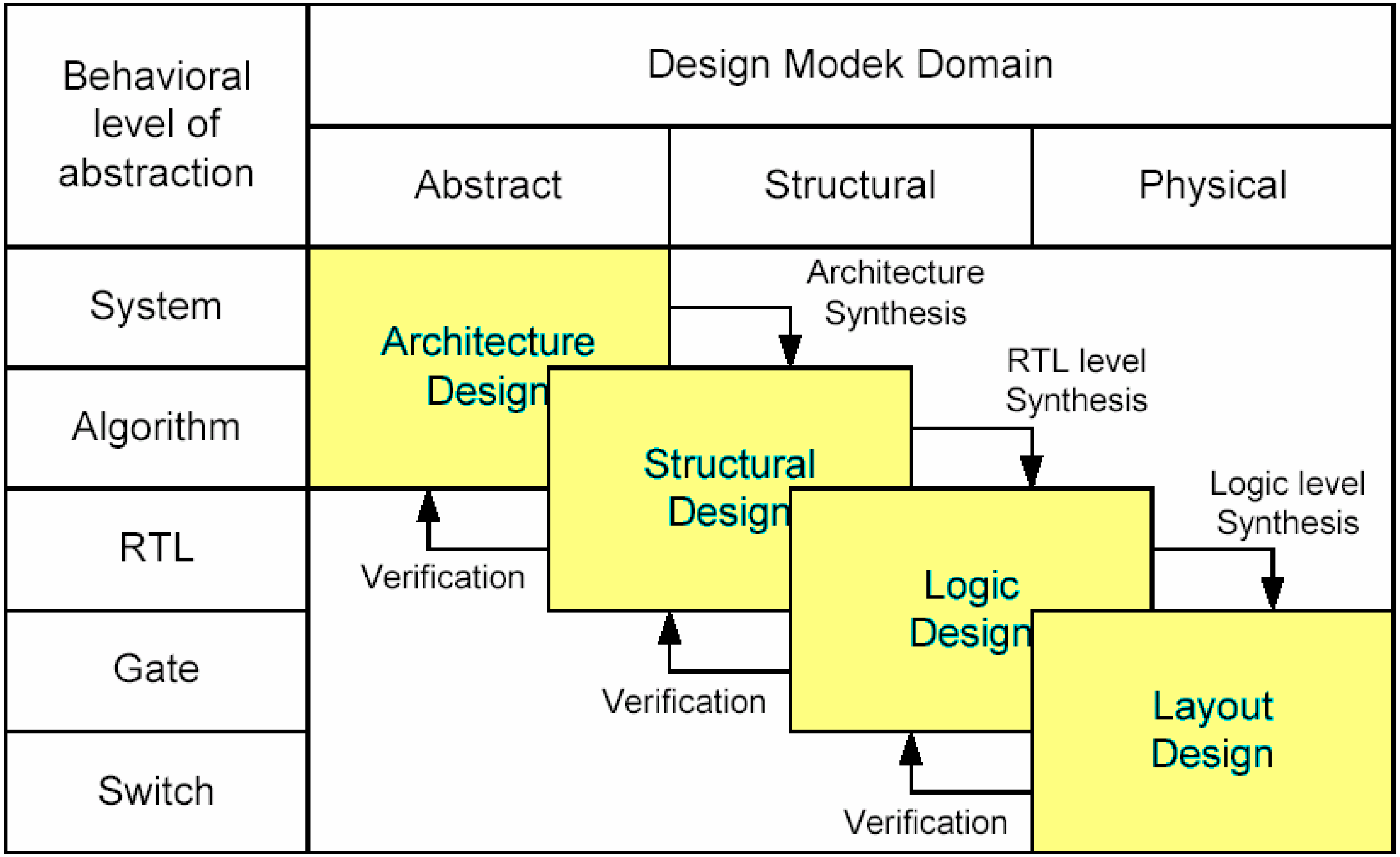


編撰者：王立洋

Top-Down Design Methodology



Design Domain



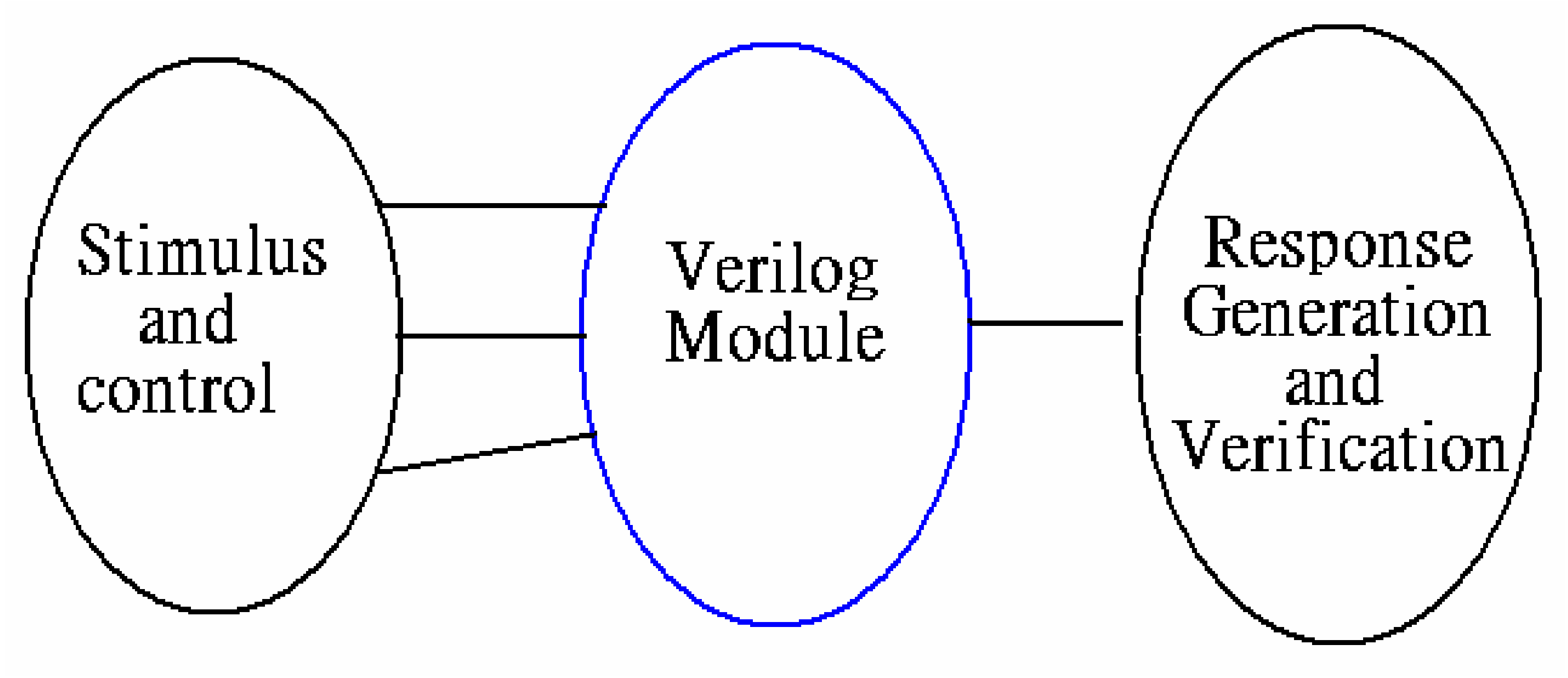
Applications

- ASIC and FPGA designers writing RTL code for synthesis
- System architects doing level system simulations
- Verification engineers writing advanced tests for all level of simulation
- Model developers describing ASIC or FPGA cells,or higher level components

Basic of the Verilog Language

1. Verilog Module
2. Identifier
3. Keyword
4. Four Value Logic
5. Data Types
6. Port Mapping
7. Numbers
8. Operator
9. Comments

Verilog Module



Verilog Module(cont.)

```
  module module_name (port_name);
        port declaration
     data type declaration
    module functionality or structure
endmodule
```

Verilog Module(cont.)

Verilog Module:basic building block

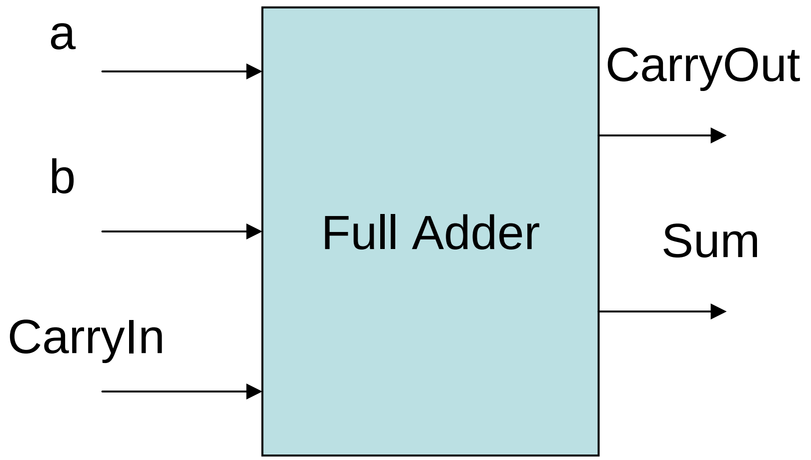
```
module DFF
---------------------
---------------------
--------------------
-
-
-
-------------------
-------------------
endmodule
```

```
module ALU
---------------------
---------------------
--------------------
-
-
-
-------------------
-------------------
endmodule
```

```
module MUX
---------------------
---------------------
--------------------
-
-
-
-------------------
-------------------
endmodule
```

Verilog Module:Example

Full Adder



```
Module FullAdd (a,b,CarryIn,Sum,CarryOut);
Input a,b,CarryIn;
output Sum,CarryOut;
wire Sum,CarryOut;


   assign {CarryOut,Sum}=a+b+CarryIn

endmoudue
```

Verilog Module:Example(cont.)

Testfixture.v

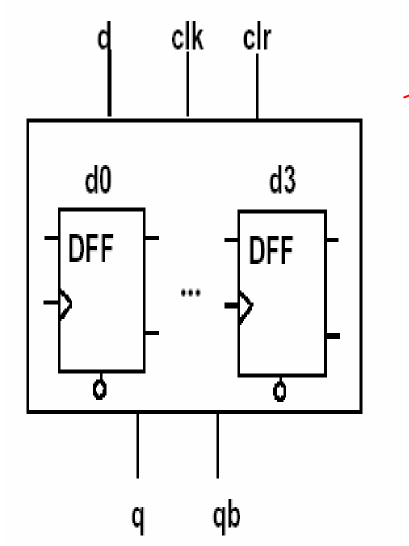
```
module testfixture;
reg a,b,CarryIn;
FullAdd (a,b,CarryIn,Sum,CarruOut);;
initial
begin
     a=0,b=0,CarryIn=0;
#5 a=1,b=0,CarryIn=0;
#5 a=1,b=1,CarryIn=0;
#5 a=1,b=1,CarryIn=1;
end
initial
  $monitor ($time,a,b,CarryIn,Sum,CarryOut);
endmodul
```

Input vectors

value	input		
time	a	b	Ca
0	0	0	0
5	1	0	0
10	1	1	0
15	1	1	1

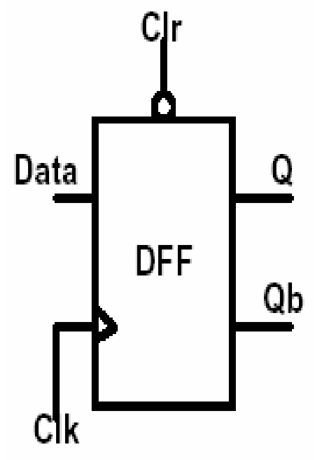
Register Verilog code

REG



```
module REG(d,clk,clr,q,qb);
output [3:0] q, qb;
input [3:0] d;
input clk, clr;
DFF d0 (d[0],clk,clr,q[0],qb[0]);
DFF d1 (d[1],clk,clr,q[1],qb[1]);
DFF d2 (d[2],clk,clr,q[2],qb[2]);
DFF d3 (d[3],clk,clr,q[3],qb[3]);
endmodule
```

DFF



```
module DFF (d, clk,clr, q, qb) ;
....
endmodule
```

Identifier

- Identifiers are names given to Verilog objects
- Names of modules, ports and instances are all identifiers
- First character must use a letter,other character can to use letter,number or“_”

Keywords

- Predefined identifiers to define the language Constructs
- All keywords are defined in lower case
- Cannot be used as identifiers

  Example:module,initial,assign,always…

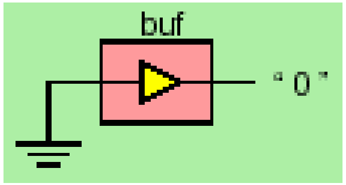
Timescale in Verilog

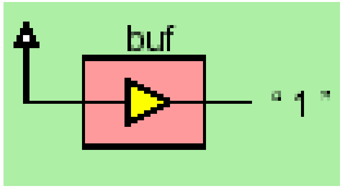
- The ‘timescale compiler directive declares the time unit and its precision.

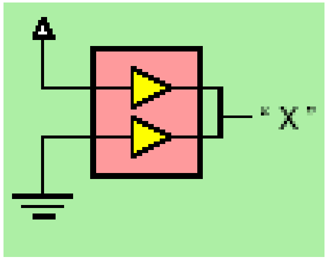
  ‘timescale <time_unit> / <time_precision>

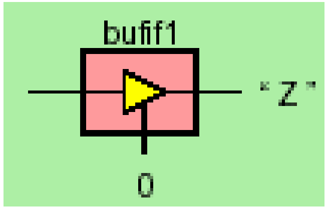
  ex:‘timescale 1 ns / 100 ps

Four Value Logic

- 

0:logic 0 / false
- 

1:logic 1 / true
- 

X:unknown logic value
- 

Z:high-impedance

Data Type

- Nets

  --- physical connection between devices

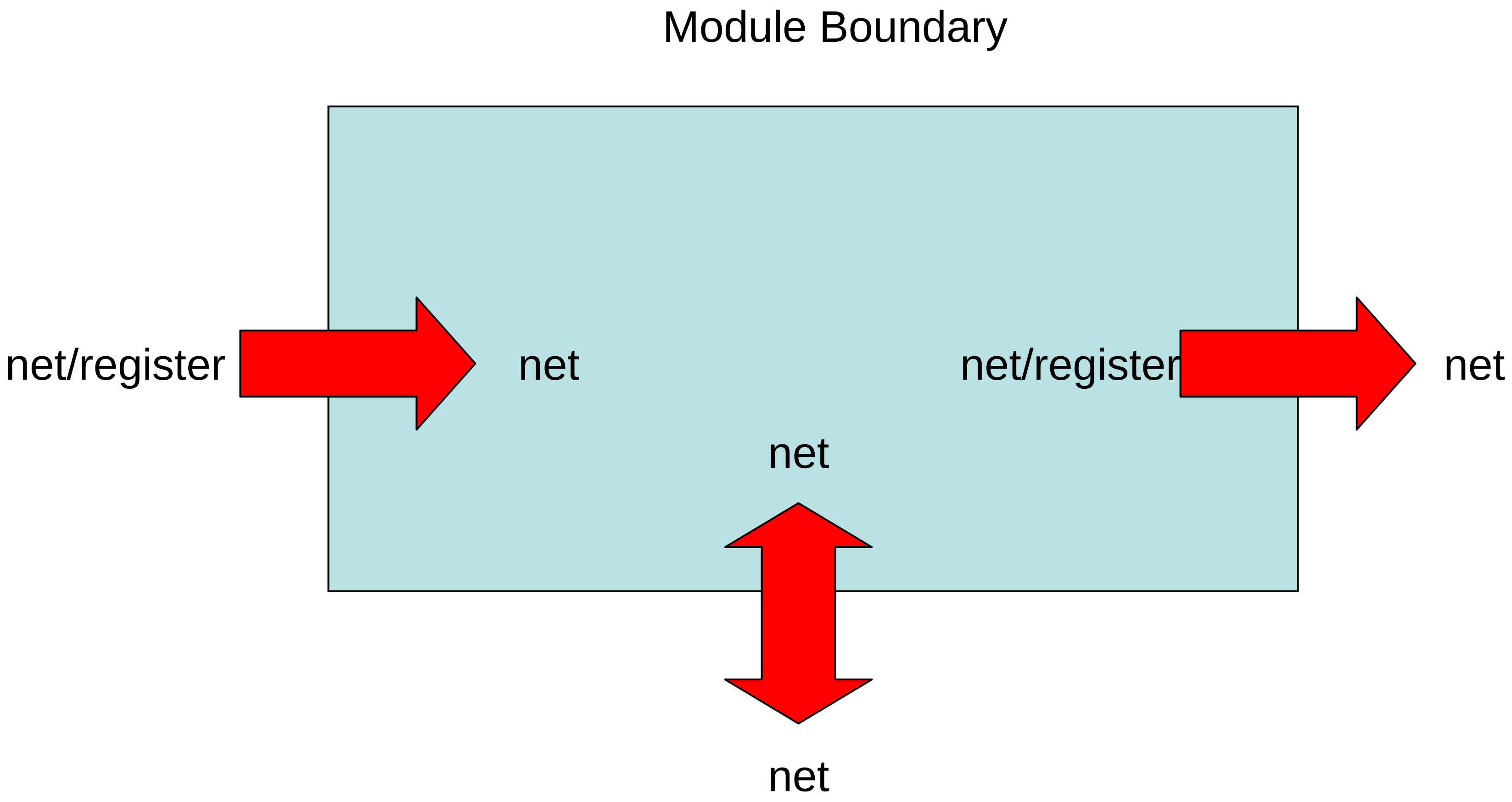
- Registers

  --- abstract storage devices

- Parameters

  --- run-time constants

Choosing the correct Data Type

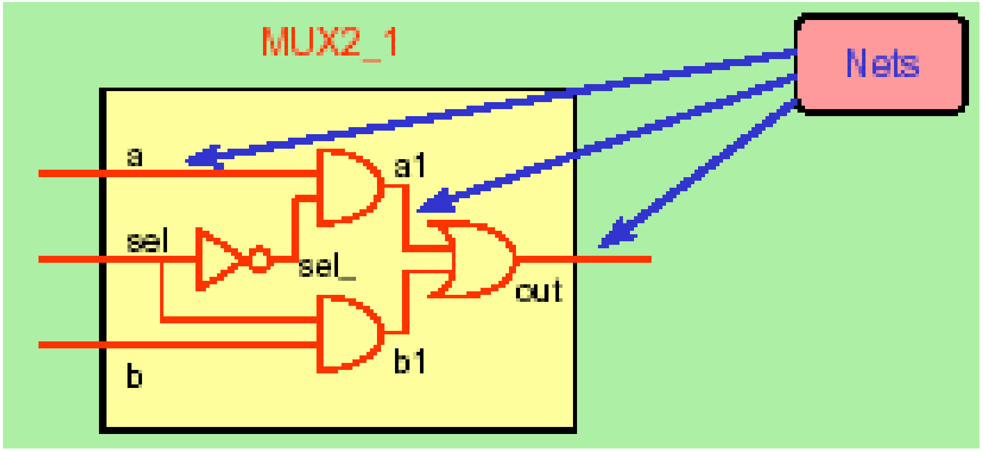


NET

- Connects between structural elements
- Must be continuously driven by

  -Continuous assignment

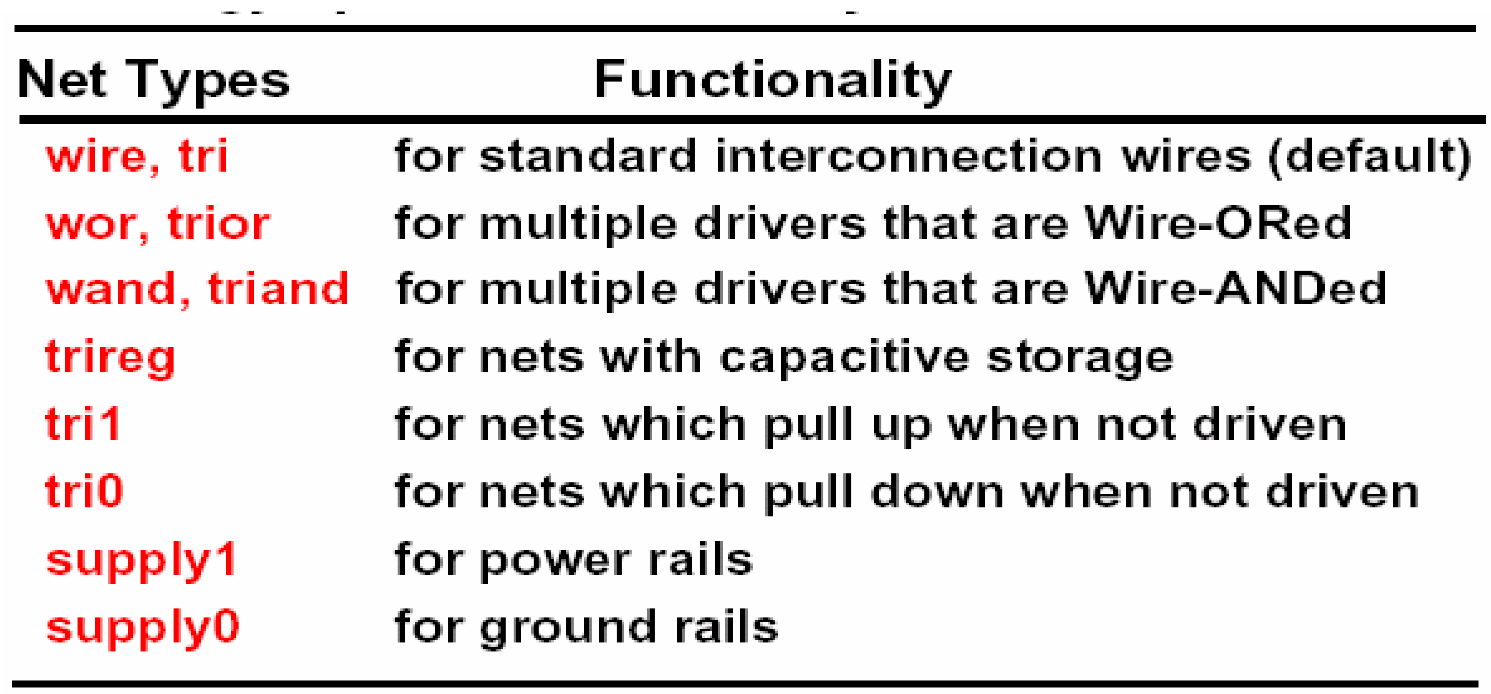
  -Module or gate instantiation
- Default initial value for a wire is “Z”



Types of Nets

- Net declaration

  <nettype> <range>? <delay_spec>? <<net_name> <,net_name>*>



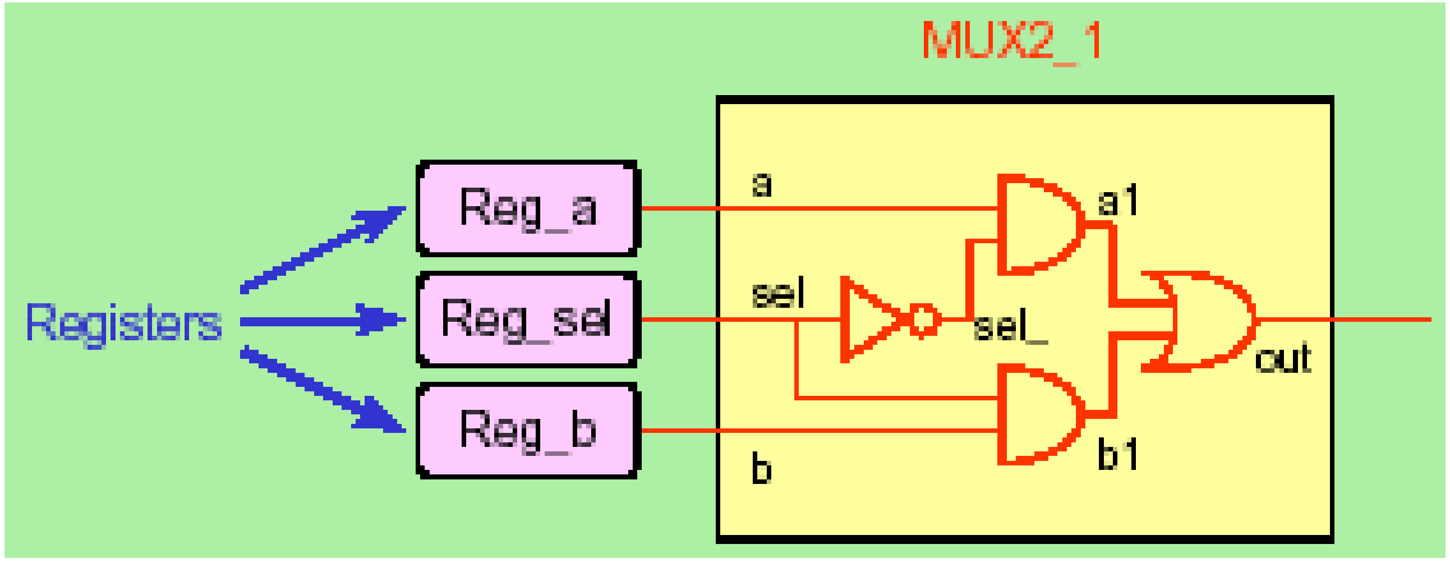
Net Types	Functionality
wire, tri	for standard interconnection wires (default)
wor, trior	for multiple drivers that are Wire-ORed
wand, triand	for multiple drivers that are Wire-ANDed
trireg	for nets with capacitive storage
tri1	for nets which pull up when not driven
tri0	for nets which pull down when not driven
supply1	for power rails
supply0	for ground rails

Examples

- wand w; // a scalar net of type “wand”
- tri [15:0] bus; // a 16-bit tri-state bus
- wire [0:31] w1, w2; // Two 32-bit wires with msb being the 0 bit

Registers

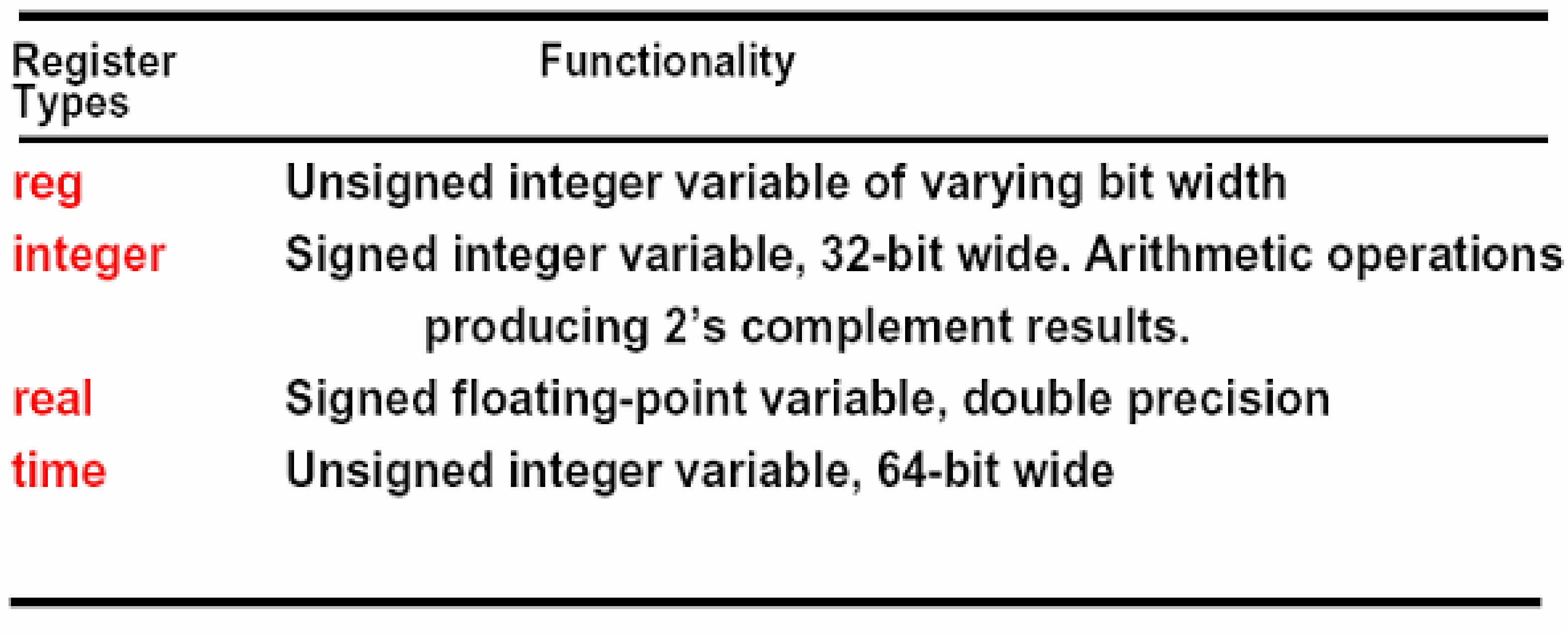
- Represent abstract data storage elements
- Registers are used extensively in behavioral modeling
- Default initial value for a wire is “X”



Types Of Register

- ➢ Register declaration

<register_type> <range>? <<register_name> <,register_name>*>



Register Types	Functionality
reg	Unsigned integer variable of varying bit width
integer	Signed integer variable, 32-bit wide. Arithmetic operations producing 2's complement results.
real	Signed floating-point variable, double precision
time	Unsigned integer variable, 64-bit wide

Examples

- reg a;   // a scalar register
- reg [3:0] b;   // a 4-bit vector register from msb to lsb
- reg [7:0] x, y;   // two 8-bit registers

Parameters

- Use parameters to declare runtime constants.
- You can use a parameter anywhere that you can use a literal.
- You can use a parameter anywhere that you can use a literal.

Types Of Parameter

- Parameter declaration

  parameter<range>?<list_of_assignments>

- You can use a parameter anywhere that you can use a literal.

ex:

```
module mod(ina, inb,out);
    ……..
    Parameter m1=8,
              real_constant =1.032,
              x_word = 16'bx,
    ……..
    wire [m1:0]  w1;
    ……..
endmodule
```

Port Mapping

- In Order

```
Mux         Mux_1(Sel,x,y,Mux_Out);
Register8   Register8_1(Clock,Reset,Mux_Out,Reg_Out);
```

- By Name

```
Mux         Mux_1(.Sel(Sel),.x(x),.y(y),.out(Mux_Out));
Register8   Register8_1(.Clock(Clock), .Reset(Reset)
            ,.data(Mux_Out),.q(Reg_Out));
```

Numbers

- numbers are integer or real constants.

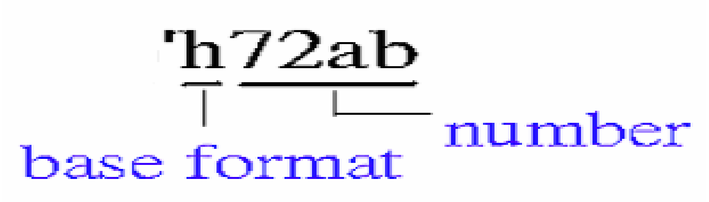
  Integer constants are written as

  <size>’<base format><number>
- Real number can be represented in decimal or scientific format.
- A number may be **sized** or **unsized**

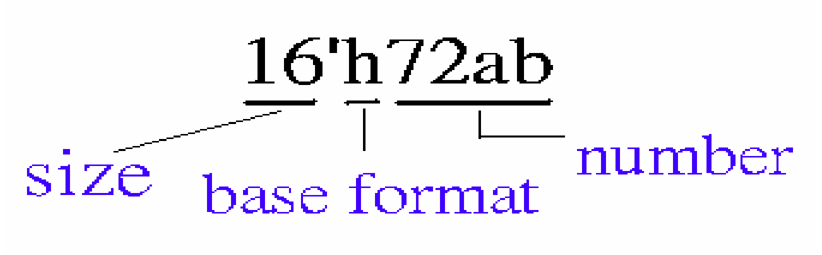
Numbers(cont.)

- The base format indicates the type of number
- Decimal (d or D)
- Hex (h or H)
- Octal (o or O)
- Binary (b or B)

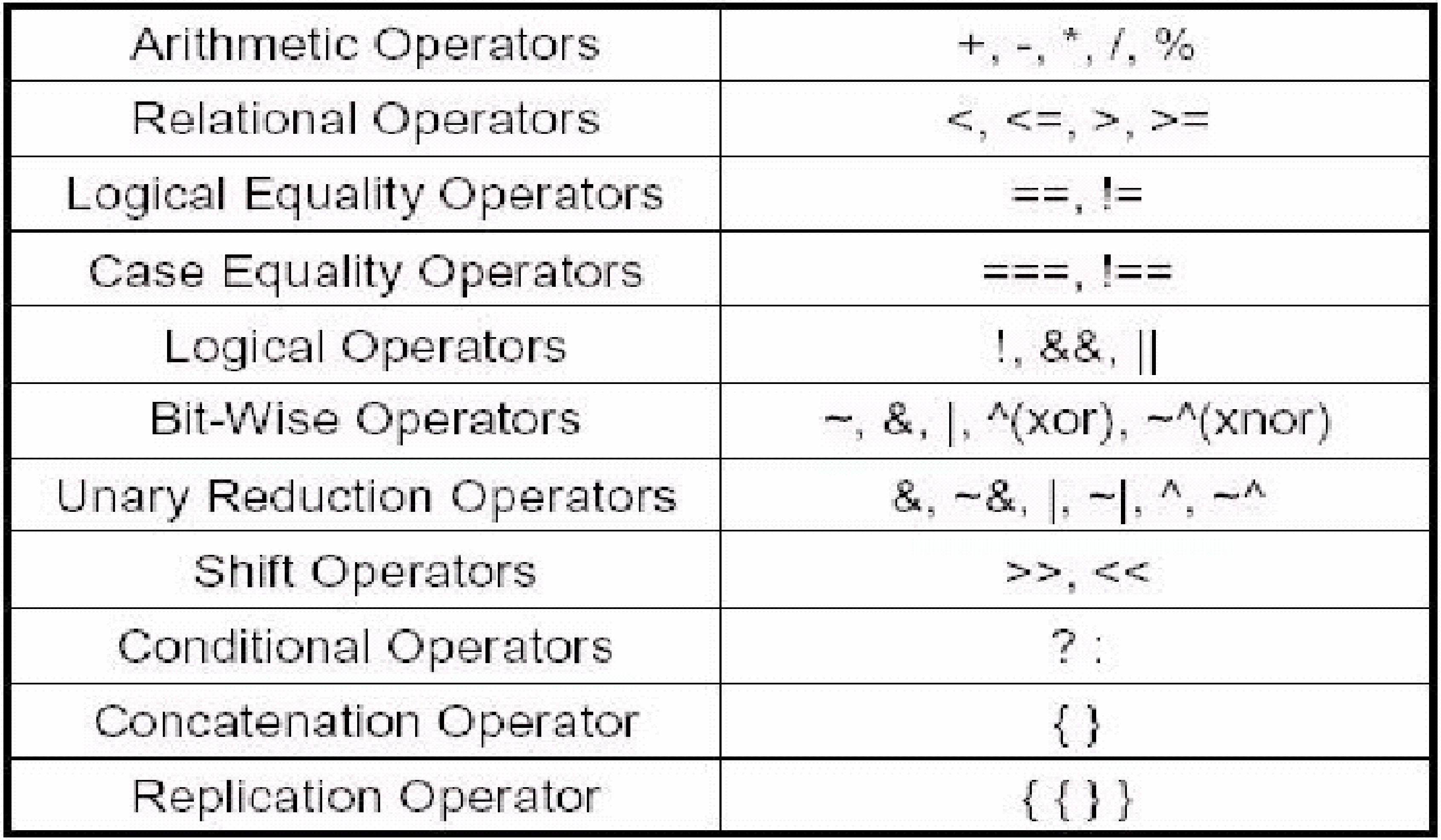
ex: unsize



size

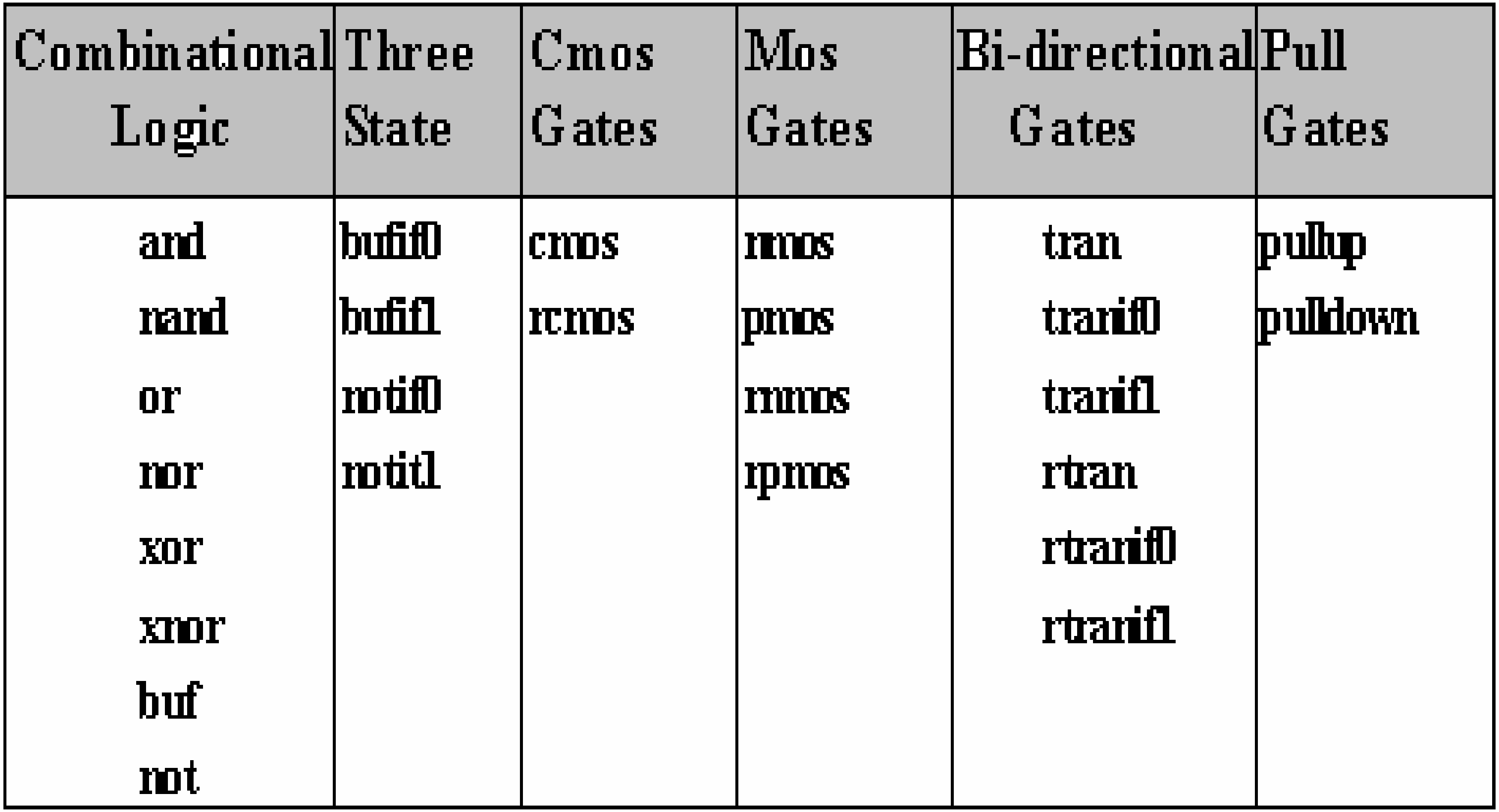


Operators



Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

Built-in Verilog primitives



Combinational Logic	Three State	Cmos Gates	Mos Gates	Bi-directional Gates	Pull Gates
and	bufif0	cmos	nmos	tran	pullup
nand	bufif1	rcmos	pmos	tranif0	pulldown
or	notif0		rnmos	tranif1	
nor	notif1		rpmos	rtran	
xor				rtranif0	
xnor				rtranif1	
buf					
not					

Comments

- Single-linecomments begin with “// ”and end with a new line character.
- Multiple-linecomments start with “/* ”and end with “*/ “.

```
module MUX (out,a,b,sel);
//Port declarations
output out;
input a,b,sel;
/*
 The netlist logic selections input “a”
  when sel=0 and it selects “b” when sel=1
*/
not (sel_,sel);
and (a1,a,sel_);
and (b1,b,sel);
or (out,a1,b1);
endmodule
```

Verilog-HDL Circuit Design

1. Behavioral Modeling
2. Structural   Modeling

Behavioral vs.Structure

- Behavioral

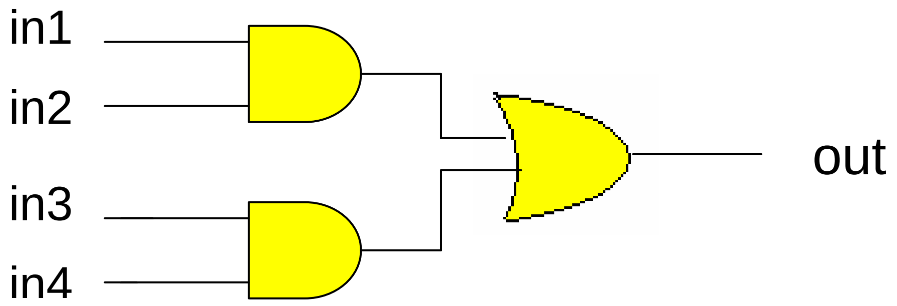
- Initial and Always blocks
- Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)
- Processes run until they delay for a period of time or wait for a triggering event

Behavioral vs.Structure(cont.)

➢ Structure

- Verilog program build from modules with I/O interfaces
- Modules may contain instances of other modules
- Modules contain local signals, etc.
- Module configuration is static and all run concurrently

Behavioral vs.Structural (cont.)



Behavioral Modeling

```
module and_or(in1,in2,in3,in4,out);
            input in1,in2,in3,in4;
            output out;
            reg out;

  always @(in1 or in2 or in3 or in4)
     begin
         if(in1 & in2)
         out=1;
     else
         out=in3&in4;
     end
endmodule
```

Structural Modeling

```
module and_or(in1,in2,in3,in4,out);
            input in1,in2,in3,in4;
            output out;
            wire tmp;
      and  m1(tmp,in1,in2),
             m2(undec in3,in4);
       or        (out,tmp,undec);
endmodule
```

Behavioral Modeling

- Behavioral modeling enables you to describe the system at a high level of abstraction
- Behavioral modeling in Verilog is described by specifying a set of concurrently active procedural blocks
- High-level programming language constructs are available in Verilog for behavioral modeling

Behavioral Modeling (cont.)

- A much easier way to write testbenches
- Also good for more abstract models of circuits
  - Easier to write
  - Simulates faster
- More flexible
- Provides sequencing
- Verilog succeeded in part because it allowed both the model and the testbench to be described together

Behavioral Modeling(cont.)

- Procedural blocks:
  - initial block: executes only once
  - always block:executes in a loop
- Block execution is triggered based on user- specified conditions
  - always @ (posedge clk) ………

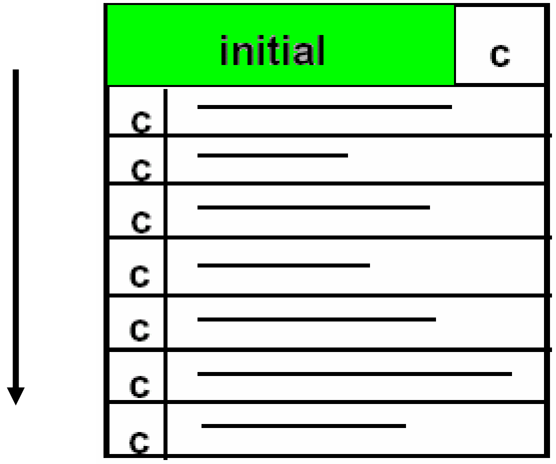
Initial Blocks

```
initial
begin
… imperative statements …
end
```

Runs when simulation starts

terminates when control reaches the end

good for providing stimulus

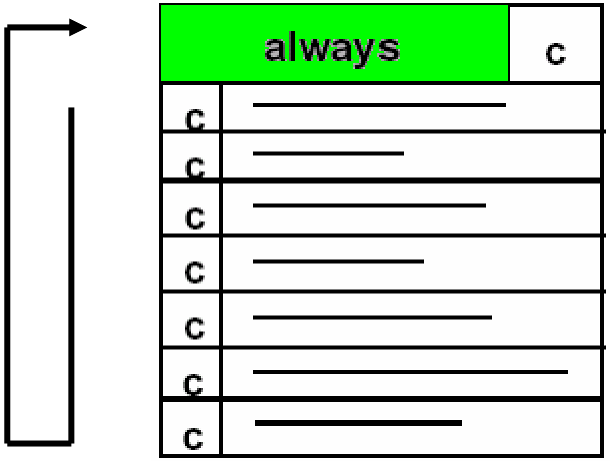


Initial procedural blocks

Always Blocks

```
always
begin
… imperative statements …
end
```

Runs when simulation starts
restarts when control reaches the end
good for modeling/specifying hardware



always procedural blocks

Descriptions in Initial and Always

- Run until they encounter a delay

```
initial begin
   #10 a = 1; b = 0;
   #10 a = 0; b = 1;
 end
```

- A wait for an event

```
always @(posedge clk)
  if(reset)
  q=1'b0;
  else
  q=d;
 endmodule
```

For Loops

- A increasing sequence of values on an output

```
 reg [4:0] i, output;
for ( i = 0 ; i <= 31 ; i = i + 1 )
begin
  output = i;
end
```

While Loops

- A increasing sequence of values on an output

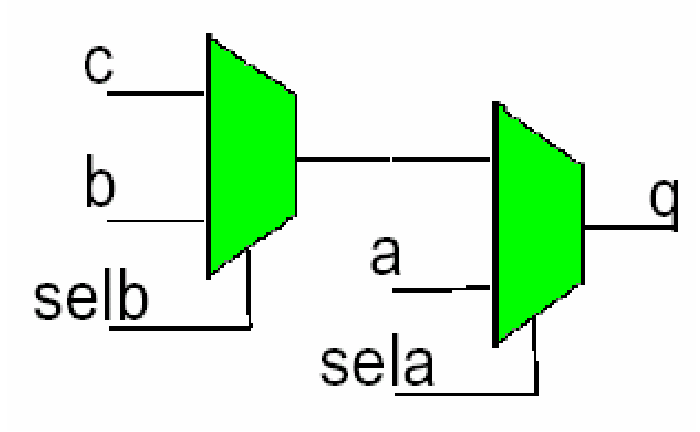
```
reg [3:0] i, output;
i = 0;
while (i <= 31)
begin
output = i;
  i = i + 1;
end
```

If-Else Statements

- Conditions are evaluated in order from top to bottom
- The first condition, that is true, causes the corresponding sequence of statements to be executed
- If all conditions are false, then the sequence of statements associated with the “else” clause isevaluated

If-Else:Example

```
always @(sela or selb or a or b or c)
     begin
         if (sela)
            q = a;
         else
         if (selb)
            q = b;
         else
            q = c;
      end
```

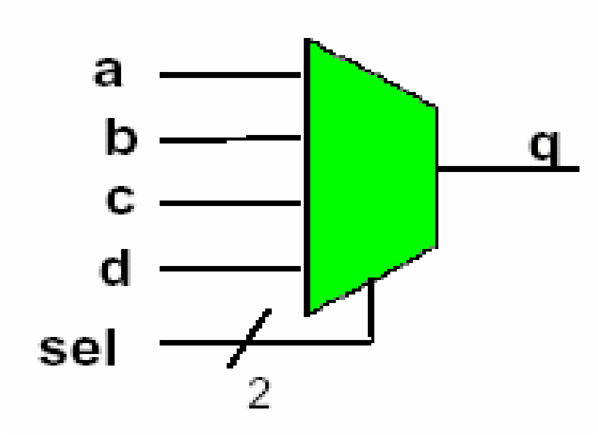


Case Statement

- Conditions are evaluated at once
- All possible conditions must be considered
- default clause evaluates all other possible conditions that are not specifically stated

Case:Example

```
always @(sel or a or b or c or d)
   begin
      case (sel)
        2'b00 :
                 q = a;
        2'b01 :
                 q = b;
        2'b10 :
                 q = c;
      default :
                 q = d;
      endcase
   end
```



Imperative Statements

```
if (select == 1) a = x;
else a = y;

case (op)
2’b00: a = x + y;
2’b01: a = x – y;
2’b10: a = x ^ y;
default: y = ‘hzzzz;
endcase
```

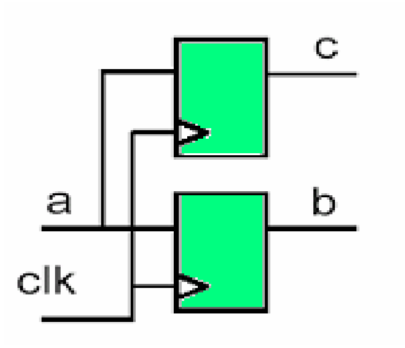
Blocking vs. Nonblocking

- ➢ Verilog has two types of procedural assignment
- ➢ Fundamental problem:
- In a synchronous system, all flip-flops sample simultaneously
- In Verilog, always @(posedge clk) blocks run in some undefined sequence

Blocking & non-Blocking

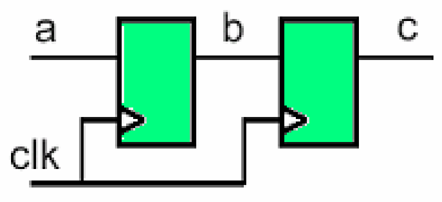
- Blocking assignments

```
always @(posedge clk)
  begin
    b = a;
    c = b;
  end
```



- Non-blocking assignments

```
always @(posedge clk)
  begin
    b <= a;
    c <= b;
  end
```

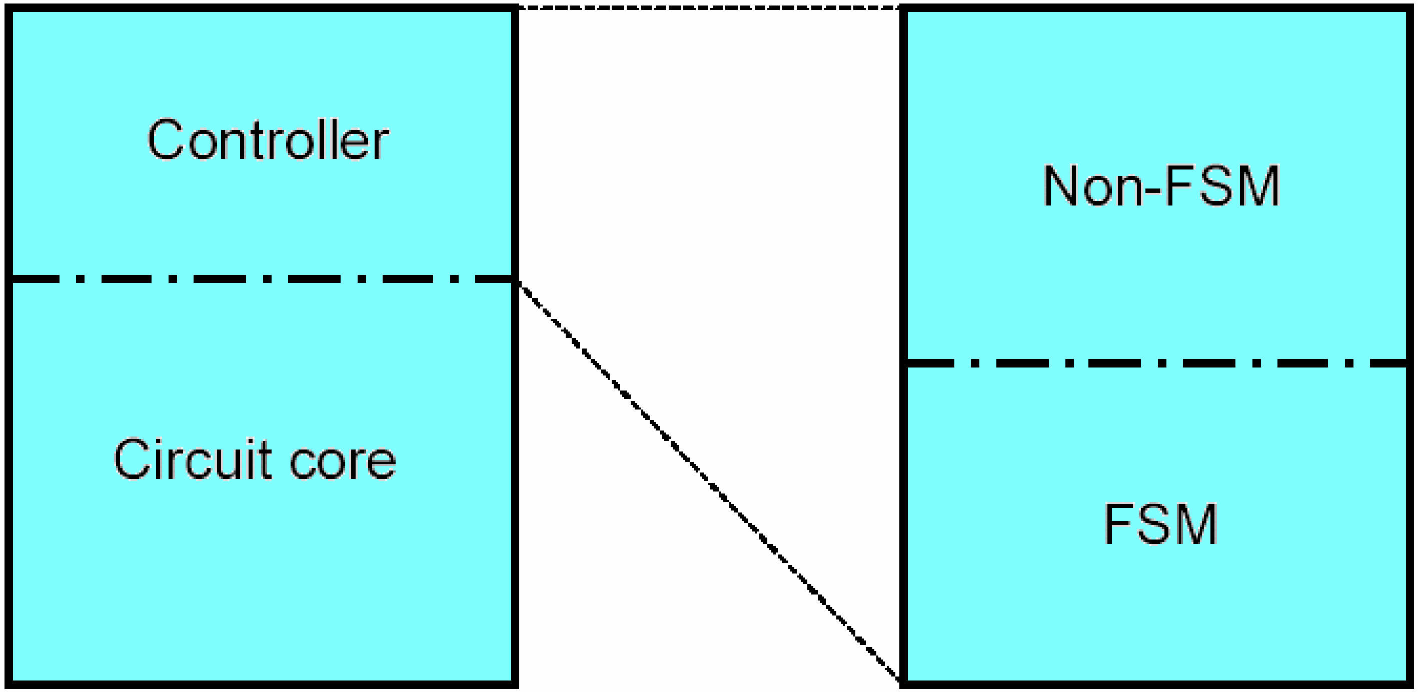


Modeling FSMs Behaviorally

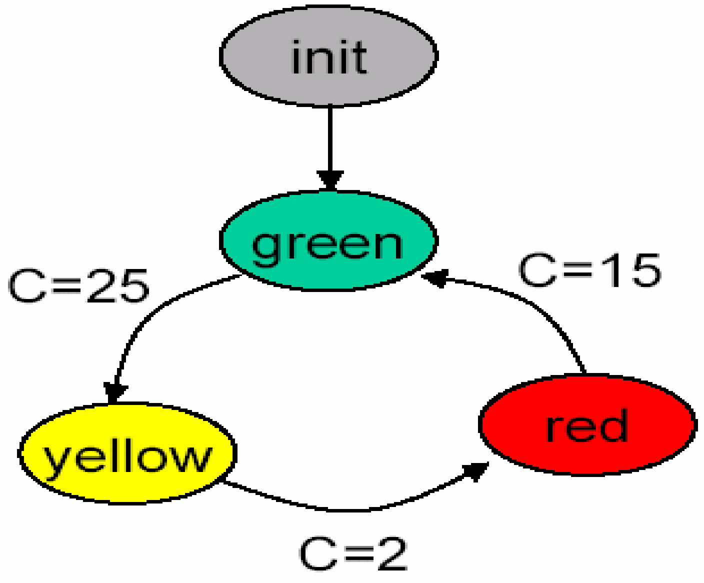
- There are many ways to do it:
- Define the next-state logic combinationally and define the state-holding latches explicitly
- Define the behavior in a single always @(posedge clk) block
- Variations on these themes

Finite State Machine

- Used control the circuit core
- Partition FSM and non-FSM part



Example of FSM



```
always @(fsm or count)
  begin
  parameter [1:0] init = 0, g = 1, y = 2, r = 3;
  red = 0; greed = 0; yellow = 0;
  fsm_nxt = fsm; start = 0;

     casex(fsm)
               init: begin
                   start = 1; fsm_next = g;
                   end
                g : begin
                   red = 0; greed = 1; yellow = 0;
                   if (count == 25) begin
                      start = 1; fsm_next = y; end
                   end
                y : begin
                   red = 0; greed = 0; yellow = 1;
                   if (count == 2) begin
                      start = 1; fsm_next = r; end
                   end
                r : begin
                   red = 1; greed = 0; yellow = 0;
                   if (count == 15) begin
                      start = 1; fsm_next = g; end
                   end
          default : begin
                   red = 0; greed = 0; yellow = 0;
                   fsm_nxt = fsm; start = 0;
                   end
      endcase
    end
```

Structural Modeling

- When Verilog was first developed (1984) most logic simulators operated on netlists
- Netlist: list of gates and how they're connected
- A natural representation of a digital logic circuit
- Not the most convenient way to express test benches

Structural modeling(cont.)

- The following gates are built-in types in the simulator
- and, nand, nor, or, xor, xnor
- First terminal is output, followed by inputs
- and a1 (out1, in1, in2);
- nand a2 (out2, in21, in22, in23, in24);
- buf, not
- One or more outputs first, followed by one input
- not N1 (OUT1, OUT2, OUT3, OUT4, INA);
- buf B1 (BO1, BIN);

Structural modeling(cont.)

- bufif0, bufif1, notif0, notif1: three-state drivers
  - Output terminal first, then input, then control
  - bufif1 BF1 (OUTA,INA,CTRLA);
- pullup, pulldown
  - Put 1 or 0 on all terminals
  - pullup PUP (PWRA, PWRB, PWRC);
- Instance names are optional
  - ex: not

User-Defined Primitives

- Way to define gates and sequential elements using a truth table
- Often simulate faster than using expressions, collections of primitive gates
- Gives more control over behavior with X inputs
- Most often used for specifying custom gate libraries

Example:Combination Logic

```
primitive drt_and(out,a,b);
input a,b;
output out;

table
0 0 :0;
0 1 :0;
1 0 :0;
1 1 :1;
endtable
endprimitive
```

Always have exactly one output

drt_and truth table

Example:Sequential Logic

```
primitive dff( q, clk, data);
output q; reg q;
input clk, data;

 table
// clk data  q new-q
  (01)   0  : ?  : 0;   // Latch a 0
  (01)   1  : ?  : 1;   // Latch a 1
  (0x)   1  : 1  : 1;   // Hold when d and q both 1
  (0x)   0  : 0  : 0;   // Hold when d and q both 0
  (?0)   ?  : ?  : -;    // Hold when clk falls
    ?   (??): ?  : -;    // Hold when clk stable
endtable
endprimitive
```

Continuous Assignment

- Another way to describe combinational function
- Convenient for logical or datapath specifications

```
ex:   wire [8:0] sum;
      wire [7:0] x, y;
      wire carry_in;

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

define bus widths

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

Recomputed when a, b, or carryin changes

Test_bench

- Understand textural and graphic outputs from Verilog.
- Understand different system function to read simulation time.
- Understand file I/O in Verilog.

Test_bench (cont.)

- Verilog has system functions to read the current simulation

  $time, $stime, $realtime

- Verilog has system tasks to support textual output

  $display, $strobe, $write, $monitor

- Verilog has system tasks to support graphic output

  $gr_waves, $gr_regs, $cWaves

Test_bench (cont.)

```
Module test_bench
    data type declaration
    module instantiation
    applying stimulus
    display  results
endmodule
```

Test_bench (cont.)

- A test bench is a top level module without inputs and outputs
- Data type declaration
- Module instantiation
- Applying stimulus
- Display results

Example

```
module testfixture;
reg [15:0]  a,b,ci;
wire [15:0] sum;
wire cout;

adder adder0(sum,cout,a,b,ci);
initial begin
a=0;b=0;ci=0;
#10 ci=1; #10 ci=0; b=1;
#10 ci=1; a=1;
#10 $finish;
end
Initial  $monitor ($time,” sum= %b  cout= %b a= %b b= %b
                              ci=%b”,sum,cout,a,b,ci);
endmodule
```

data type declaration

module instantiation

applying stimulus

display results

Timing Control

- Simple Delay

  #10 rega=regb ;

  #(cycle/2) clk=~clk ; // cycle is declared as a parameter

- Edge-Triggered Timing Control

  @(r or q) rega=regb ; // controlled by in “r” or “q”

  @(posedge clk) rega=regb ; // controlled by postive edge

  @(negedge clk) rega=regb ; // controlled by negative edge

Timing Control

- Level-Triggered Timing Control

```
wait (!enable) rega=regb ;   // wait until enable = 0
```

Reading Simulation Time

- The $time,$realtime,$stime functions return the current simulating time.
- $time returns time as a 64-bit integer.
- $stimereturns time as a 32-bit integer.
- $realtimereturns time as a real number.

Displaying Signal Values

- $display prints out the current values of the signals in the argument list.
- $display automatically prints a new line.
- $display supports different bases.

```
$display ($time, “%b \t %h \t %d \t %o”,
                    sig1, sig2, sig3, sig4) ;
```

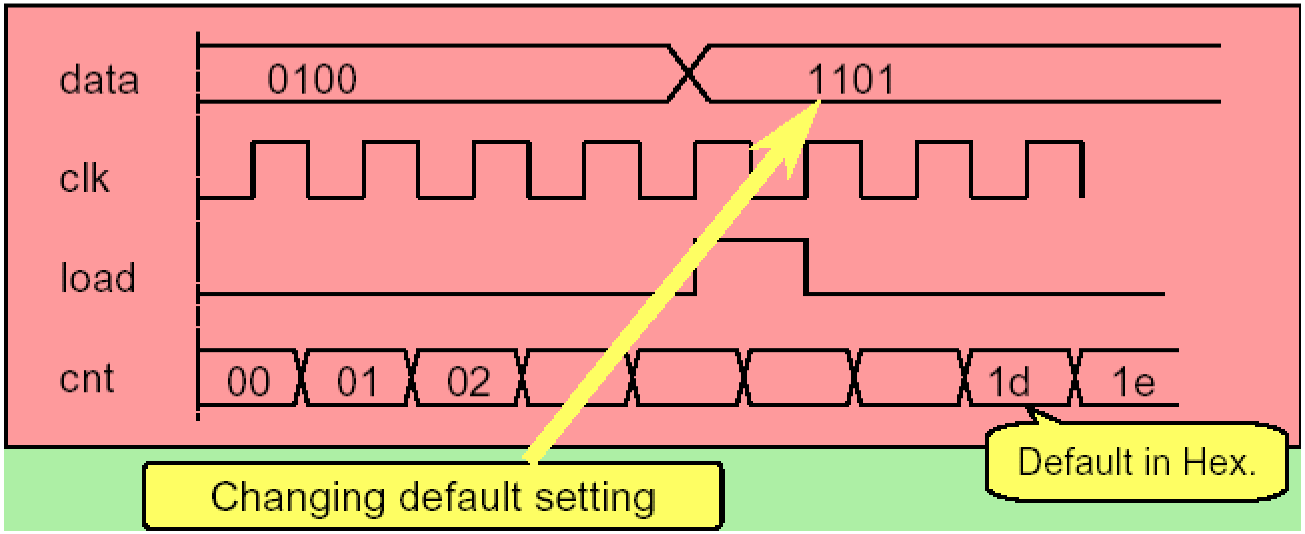
Displaying Signal Values(cont.)

- $write is identical to $display except that it does not print a new line character.
- $strobe is identical to $display except that the argument evaluation is delayed just prior to advance of the simulation time.

Verilog Graphic Display

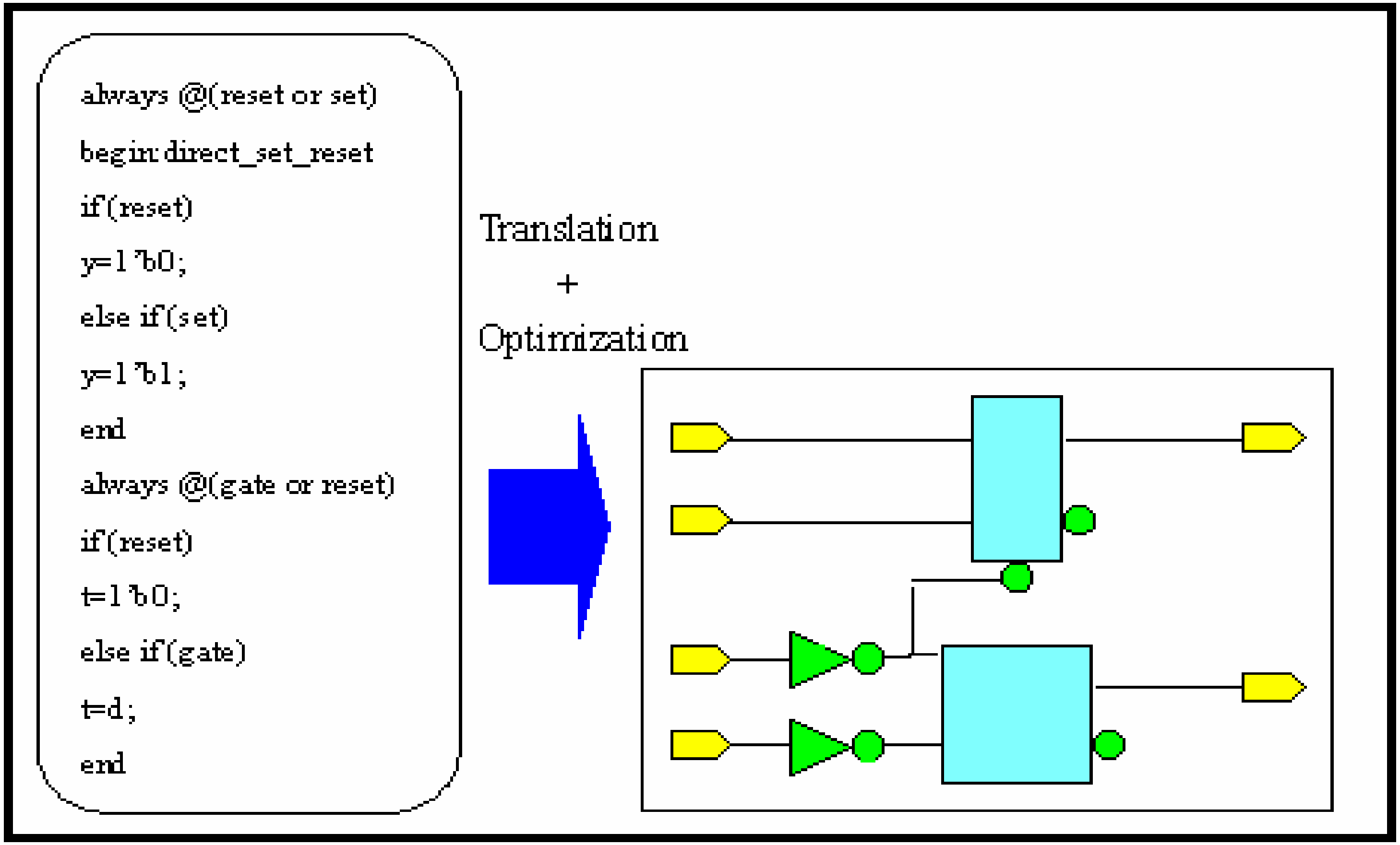
- $gr_waves displays the argument list in a graphic window.
- Example :

$gr_waves (“data %b”, data, “clk”,clk, “load”, load, “cnt”,cnt);



What is Synthesis

- Synthesis = translation + optimization



```
always @(reset or set)
begin: direct_set_reset
if(reset)
y=1'b0;
else if(set)
y=1'b1;
end
always @(gate or reset)
if(reset)
t=1'b0;
else if(gate)
t=d;
end
```

Logic Synthesis

- Takes place in two stages:
- Translation of Verilog (or VHDL) source to a netlist
- Optimization of the resulting netlist to improve speed and area

Translating Verilog into Gates

- ➢ Parts of the language easy to translate
  - Structural descriptions with primitives
  - Continuous assignment

- ➢ Behavioral statements the bigger challenge

What Can Be Translated

- ➢ Structural definitions
- ➢ Behavioral blocks
  - Only when they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
  - Blocks sensitive to both edges of the clock, changes on unrelated signals, changing sensitivity lists
- ➢ User-defined primitives
  - Primitives defined with truth tables

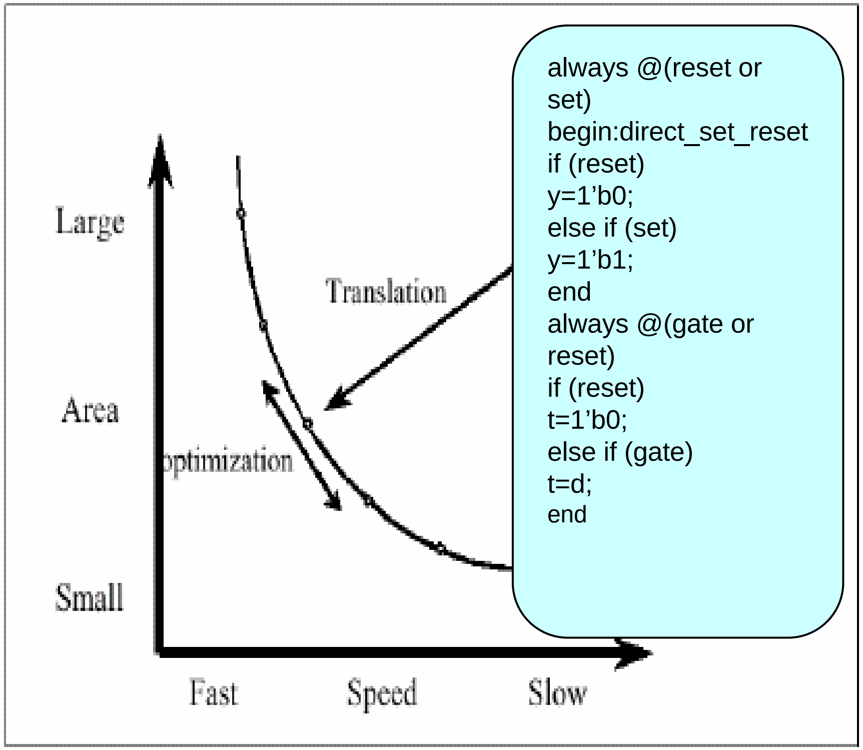
What Isn’t Translated

- Initial blocks
  - Used to set up initial state or describe finite testbench
  - Don’t have obvious hardware component

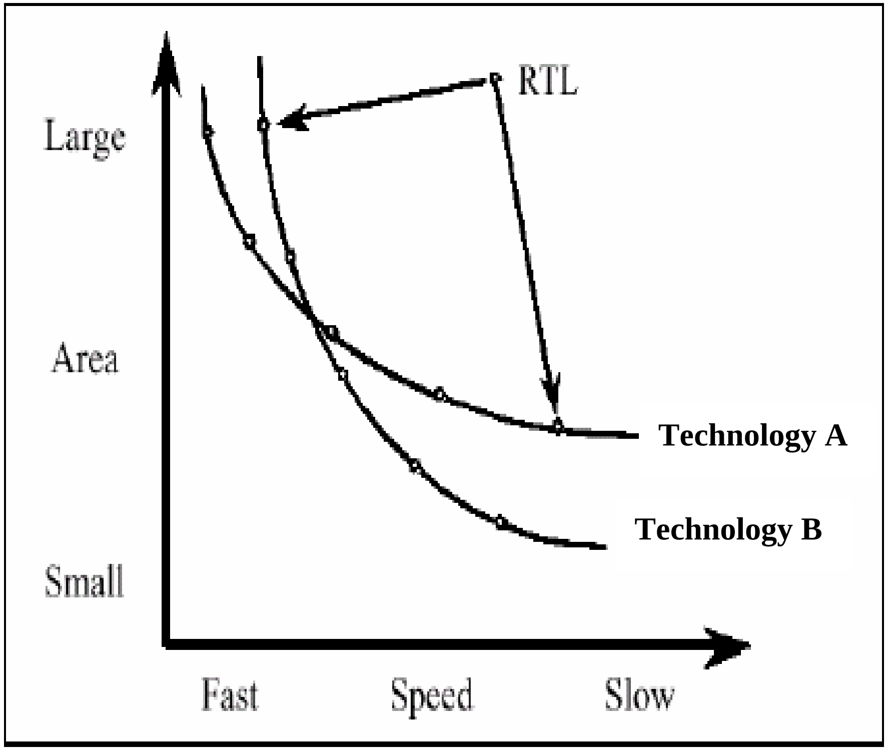
- Delays
  - May be in the Verilog source, but are simply ignored

Translation & Optimization

- Synthesis is Constraint Driven
- Technology Independent



```
always @(reset or
set)
begin:direct_set_reset
if (reset)
y=1’b0;
else if (set)
y=1’b1;
end
always @(gate or
reset)
if (reset)
t=1’b0;
else if (gate)
t=d;
end
```



References

(1). (楊智喬) “*Logic Synthesis(Synopsys)*” CIC

(2). (XILINX) “*FPGA CompilerII/FPGA Express Verilog HDL Reference manual* ”

(3). “*Verilog Training Manual* ” CIC Feb.-2002

(4). (MICHAELD.CILETTI)“*Modeling,Synthesis, and Rapid Prototyping with the Verilog HDL*” PRENTICE HALL