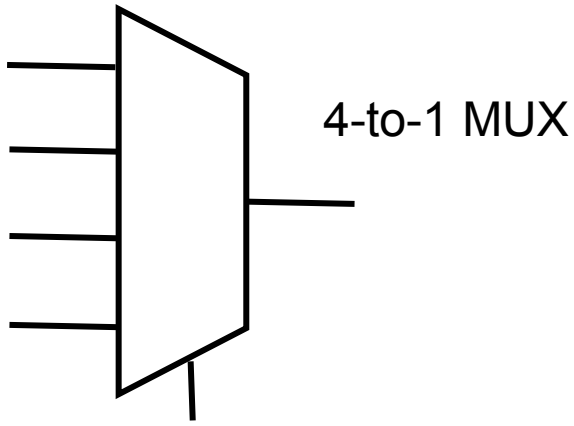
Logic Design with Verilog I: Basic Syntax

Jae W. Lee (jaewlee@snu.ac.kr)
Department of Computer Science and Engineering
Seoul National University

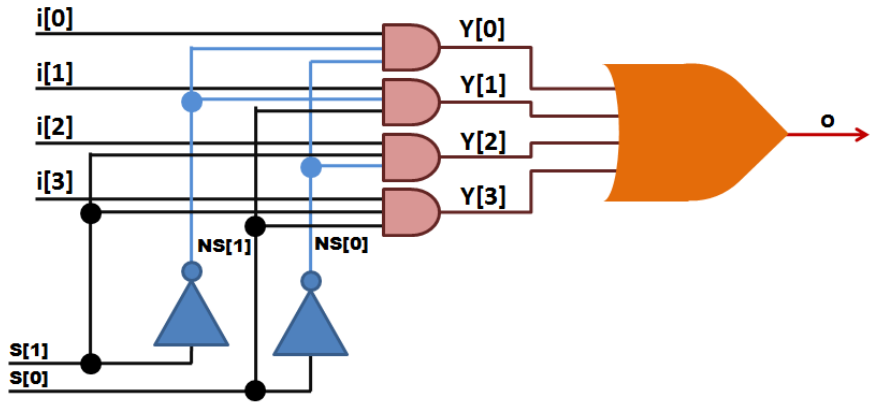
Slide credits: Prof. Sungjoo Yoo @ CMA Lab

What is HDL / Verilog

- Why use HDL (Hardware Description Language)?
  - Design can be described at a very abstract level
  - Functional verification can be done early in the design cycle
  - Reduce cost and time to design hardware
- Verilog is one of the most popular HDLs
  - Allowing different levels of abstraction in the same module
  - A general-purpose, easy to learn, and easy to use HDL language



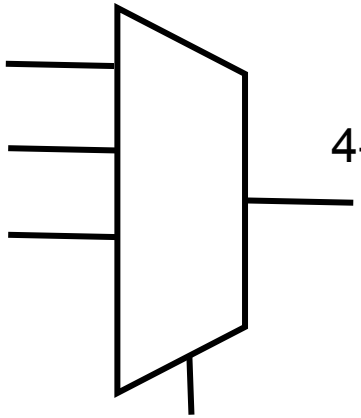
4-to-1 MUX



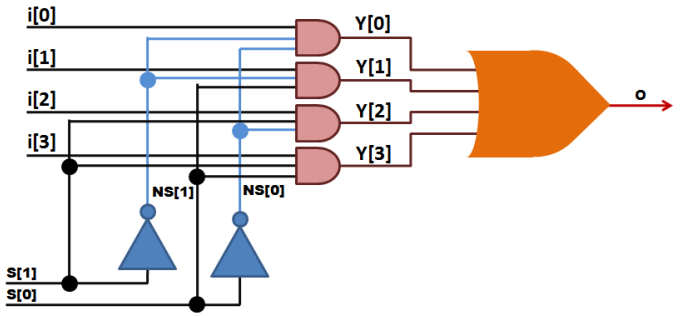
Source: http://www.vlsifacts.com/different-coding-styles-verilog-language/

What is HDL / Verilog

- Key features of Verilog
  - Supports various levels of abstraction
    - Behavior level
    - Register transfer level
    - Gate level
  - Simulate design functions
  - Combines structural and behavioral modeling styles



4-to-1 MUX

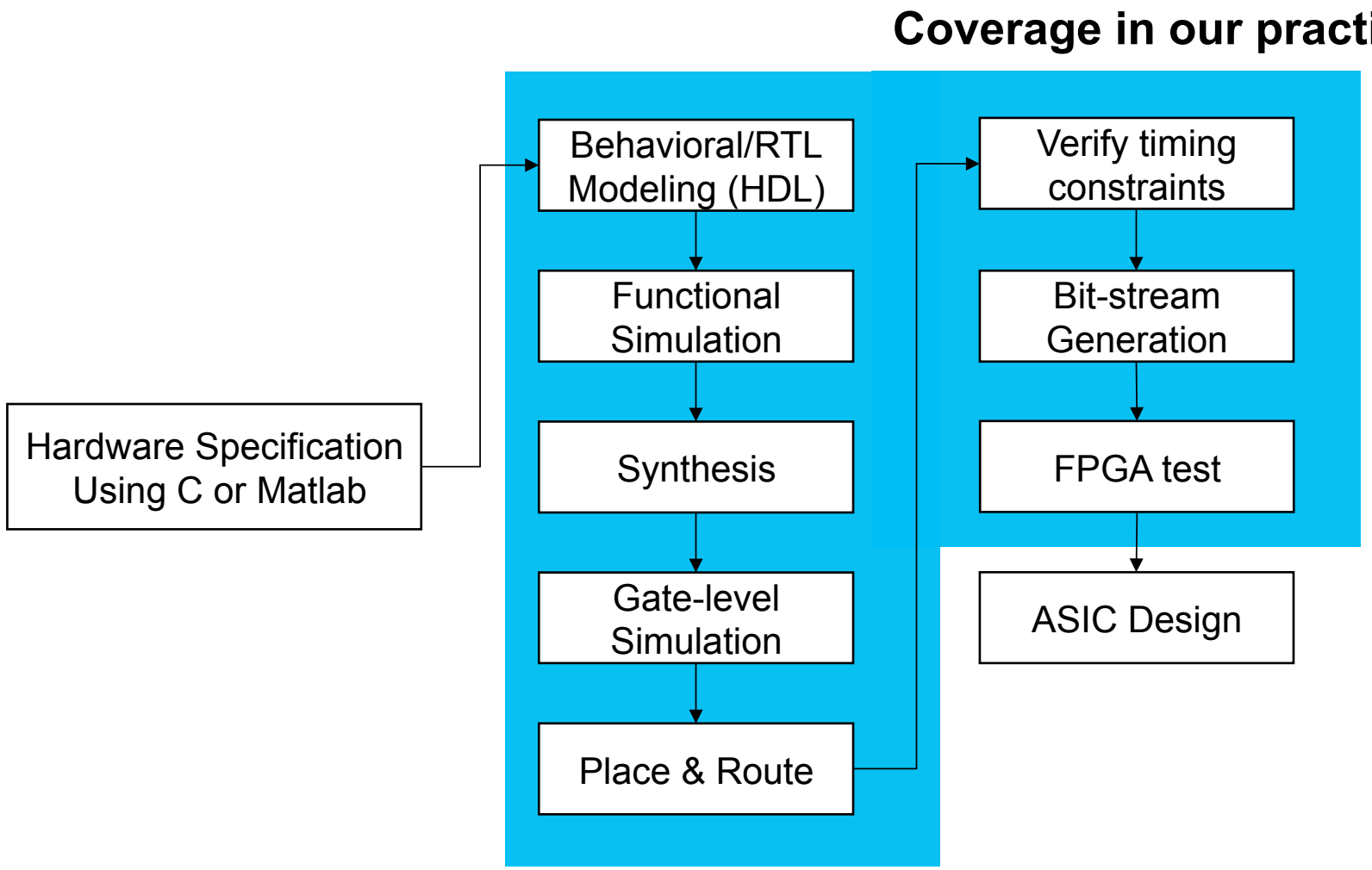



```
// structural model
module Mux_4to1_gate(
input [3:0] i,
input [1:0] s,
output o
);
wire NS0, NS1;
wire Y0, Y1, Y2, Y3;
not N1(NS0, s[0]);
not N2(NS1, s[1]);
and A1(Y0, i[0], NS1, NS0);
and A2(Y1, i[1], NS1, s[0]);
and A3(Y2, i[2], s[1], NS0);
and A4(Y3, i[3], s[1], s[0]);
or O1(o, Y0, Y1, Y2, Y3);
endmodule
```

```
// behavioral model
module Mux_4to1(
   input [3:0] i,
   input [1:0] s,
   output reg o
);
always @(s or i)
begin
   case (s)
      2'b00 : o = i[0];
      2'b01 : o = i[1];
      2'b10 : o = i[2];
      2'b11 : o = i[3];
      default : o = 1'bx;
   endcase
end
endmodule
```

Source: http://www.vlsifacts.com/different-coding-styles-verilog-language/

HDL-Based Design Flow



Outline

- Number Representation
- Data Types
- Dataflow Modeling with Continuous Assignments
- Operators
- Behavioral Modeling with `initial` and `always` Blocks

Number Representation

- <size>'<base format><number>
  - <size> : Number of bits (optional)
  - <base format> : Single character (b, d, o, and h)
  - <number> : Contains digits which are legal for the <base format>

- Examples
  - 549 // decimal number
  - ‘h8FF // hex number
  - ‘o765 // octal number
  - 4’b11 // 4-bit binary number 0011
  - 3’b10x // 3-bit binary number with least significant bit unknown
  - 5’d3 // 5-bit decimal number
  - -4’b11 // 4-bit two’s complement of 0011, or equivalently 1101

Number Representation

- Sized number
  - <size>'<base format><number>
  - 4'b1001
  - 16'habcd
- Unsized number: 32 bits or more (machine specific)

  '<base format><number>
  - 2009
  - 'habc
- x or z
  - x: an unknown value
  - z: a high impedance

Number Representation

- Verilog's nets and registers hold four-valued data
  - 0
    - Obvious, logic 0, false condition
  - 1
    - Obvious, logic 1, true condition
  - Z
    - Output of an undriven tri-state driver
    - Models case where nothing is setting a wire's value
  - X
    - Models when the simulator can't decide the value
    - Initial state of registers
    - When a wire is being driven to 0 and 1 simultaneously
    - Output of a gate with Z inputs

Outline

- Number Representation
- Data Types
- Dataflow Modeling with Continuous Assignments
- Operators
- Behavioral Modeling with `initial` and `always` Blocks

Data Types

- **Nets:** any hardware connection points
- **Variables:** any data storage elements

Nets		Variables
wire	supply0	reg
tri	supply1	integer
wand	tri0	real
wor	tri1	time
triand	trireg	realtime
trior	uwire	

Data Types: Two Main Types

- Nets (wire) represent connections between things
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - Cannot be assigned in an *initial* or *always* block

- Regs (reg) represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Never connected to something
  - Can be used to model latches, flip-flops, etc.
  - A variable of type register does not necessary represent a physical register

Data Types: `integer` and `time` Variables

- `integer` variable
  - Contains integer values
  - Has at least 32 bits
  - Is treated as a signed reg variable

```
integer i, j;
integer data[7:0];
```

- `time` variable
  - Used for storing and manipulating simulation time
  - Used in conjunction with the $time system task
  - Only unsigned value and at least 64 bits

```
time events;
time current_time;
```

Data Types: Variable Declaration

- Size of register or wire

```
reg  [0:7]  A, B; // A and B are 8-bit wide with MSB as 0th bit
wire [3:0]  Dout; // Dout is a 4-bit wire
reg  [7:0]  C;    // C is 8-bit register with MSB as 8th bit
```

- Assignments and concatenations

```
A = 8'b0101_1010;
B = {A[3:0] | A[7:4], 4'b0000};
```

  - B is set to the 1st 4-bit of A bitwise or-ed with the last 4-bit of A and then concatena ted with 0000.
  - {} brackets means the bits of the 2 or more arguments separated by commas are concatenated together.

Data Types: Vectors

- A vector = multiple bit width
  - [high:low] or [low:high]
  - The leftmost bit is the MSB
  - include nets and reg data types
- A scalar = 1-bit vector
- Array and Memory Elements
  - All net and variable data types are allowed
  - An array element can be a scalar or a vector

```
wire a[3:0];
reg  d[7:0];
wire [7:0]  x[3:0];
reg  [31:0] y[15:0];
integer states [3:0];
time    current[5:0];
```

Data Types: Memory

- Memory
  - model ROM, RAM, or register files

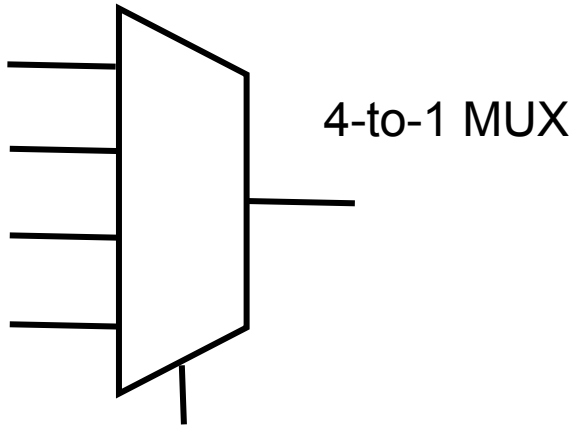
```
reg [7:0] mema [7:0];
reg [7:0] memb [3:0][3:0];
wire sum [7:0][3:0];

mema[4][3]
mema[5][7:4]
memb[3][1][1:0]
sum[5][0]
```

Outline

- Number Representation
- Data Types
- Dataflow Modeling with Continuous Assignments
- Operators
- Behavioral Modeling with `initial` and `always` Blocks

Continuous Assignment: `assign` Statement

- Lvalue, Rvalue
  - Lvalue: wire
  - Rvalue: wire, reg
- Connecting wire physically




```
module Mux_4to1_df(
input [3:0] i,
input [1:0] s,
output o
);

assign o = (~s[1] & ~s[0] & i[0]) | (~s[1] & s[0] & i[1]) |
           (s[1] & ~s[0] & i[2]) |  (s[1] & s[0] & i[3]);
endmodule
```

Continuous Assignment: Basic Grammars

- Describe a system by a set of modules
  - Equivalent to functions in ‘C’
- Keywords
  - E.g., module, are reserved in all lower case letter
- Operators (some examples)
  - Arithmetic: +, -, *, /, !, ~
  - Binary operators: &, |, ^, ~(bitwise), !(logical)
  - Shift: <<, >>
  - Relational: <, <=, >, >=, ==, !=
  - Logical: &&, ||
- Identifiers
  - Equivalent to variable names
  - Identifiers can be up to 1024 characters
- Comments
  - “//” for 1-line or /* to */ across several lines

Outline

- Number Representation
- Data Types
- Dataflow Modeling with Continuous Assignments
- Operators
- Behavioral Modeling with `initial` and `always` Blocks

Operators

Arithmetic	Bitwise	Reduction	Relational
+: add - : subtract * : multiply / : divide % : modulus **: exponent	~ : NOT &: AND | : OR ^: XOR ~^, ^~: XNOR	&: AND |: OR ~&: NAND ~|: NOR ^: XOR ~^, ^~: XNOR	>: greater than <: less than >= : greater than or equal <=: less than or equal
Shift	**case equality**	**Logical**	**Miscellaneous**
<< : left shift >> : right shift <<< : arithmetic left shift >>>: arithmetic right shift	===: equality !==: inequality	&&: AND || : OR ! : NOT	{ , }: concatenation {c{ }}: replication ? : conditional
	Equality		
	==: equality !=: inequality		

Operators: Arithmetic Ops

- The result is x if any operand bit has a value x
- Negative numbers except integer variable are represented as 2's c omplement

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent (power)
%	Modulus

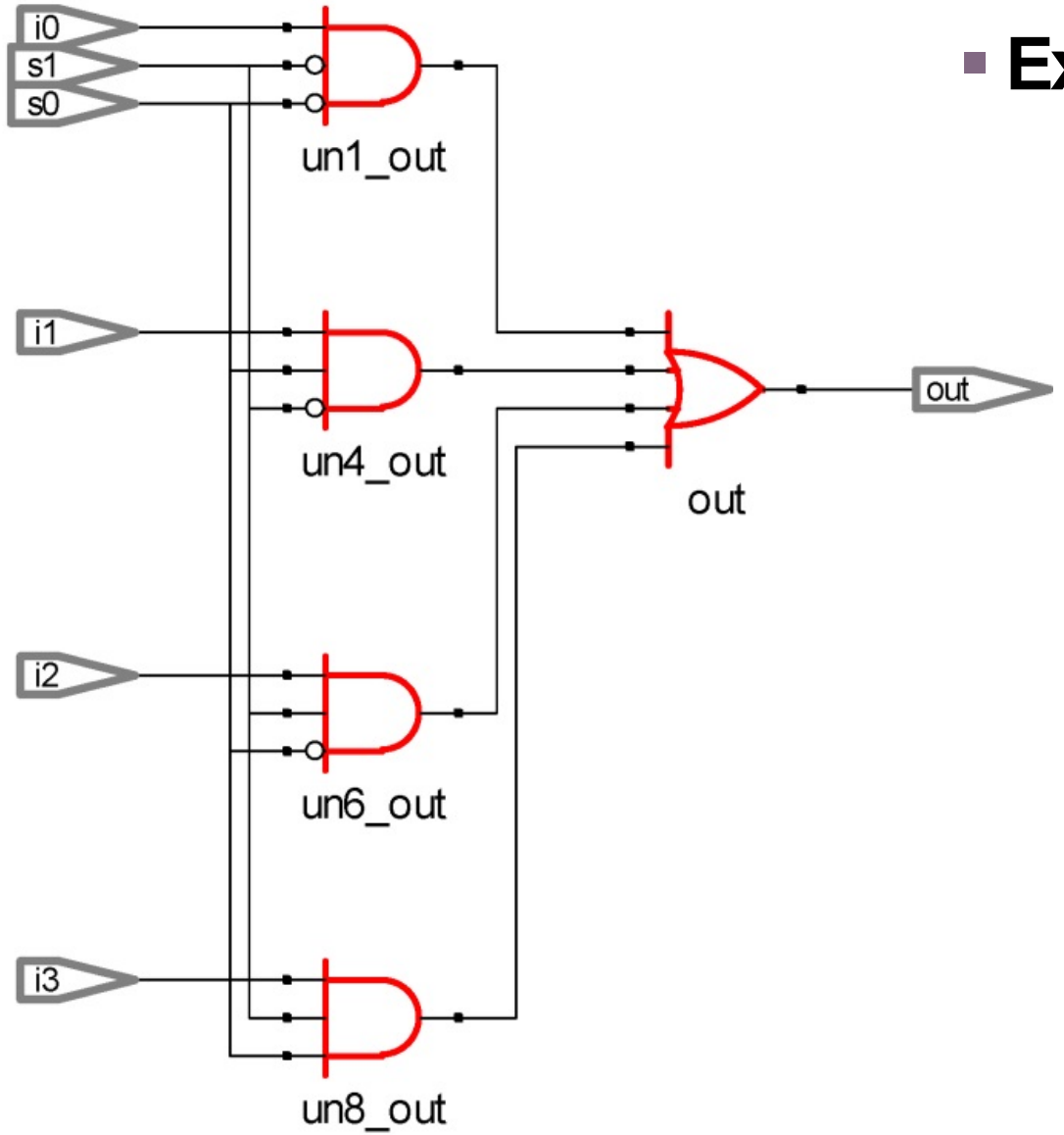
```
-10/5       // -2
-'d10 / 5   // (2's comp. of 10) / 5
            // = (2^32-10) / 5
```

Operators: Bitwise Ops

- A bit-by-bit operation
  - A z is treated as x
  - The shorter operand is zero-extended

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Operators: Bitwise Ops



- **Example: 4-to-1 MUX**

```
// using basic and, or , not
assign out = (~s1 & ~s0 & i0) |
             (~s1 & s0 & i1)  |
             (s1 & ~s0 & i2)  |
             (s1 & s0 & i3) ;
```

Operators: Logical Ops

- Always evaluate to a 1-bit value, 0, 1, or x
- The result is x (a false condition) if any operand bit is x or z

Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or

```
//   A = 4'b1010,
//   B = 4'b1000
X | Y   // 4’b1010
X || Y  // 1
X & Y   // 4’b1000
X && Y  // 1
```

Operators: Concatenation/Replication

- Concatenation operator

  y = {a, b[0], c[1]};

  only for sized operand
- Replication operator

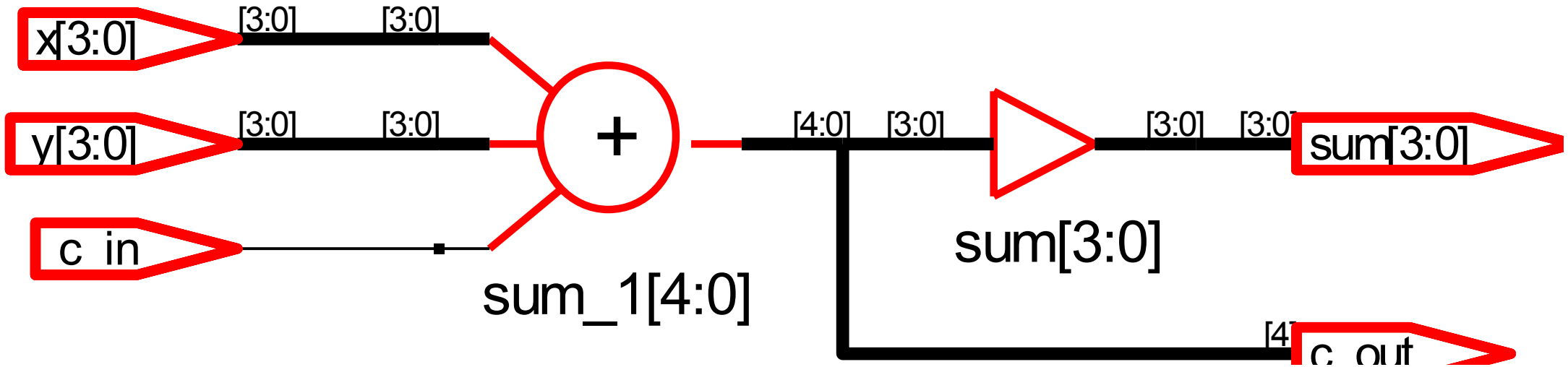
  y = {a, {4{b[0]}}, c[1]}; // y = {a, b[0], b[0], b[0], b[0], c[1]};

Symbol	Operation
{ , }	Concatenation
{const_expr{}}	Replication

```
reg A;
reg [1:0] B, C;
reg [2:0] D;

A = 1'b1; B = 2'b00: C = 2'b10; D = 3'b110;

Y = { 4{A} };           // Y = 4'b1111
Y = { 4{A}, 2{B} };     // Y = 8'b11110000
Y = { 4{A}, 2{B}, C };  // Y = 11'b11110000110
```

Operators: Concatenation/Replication

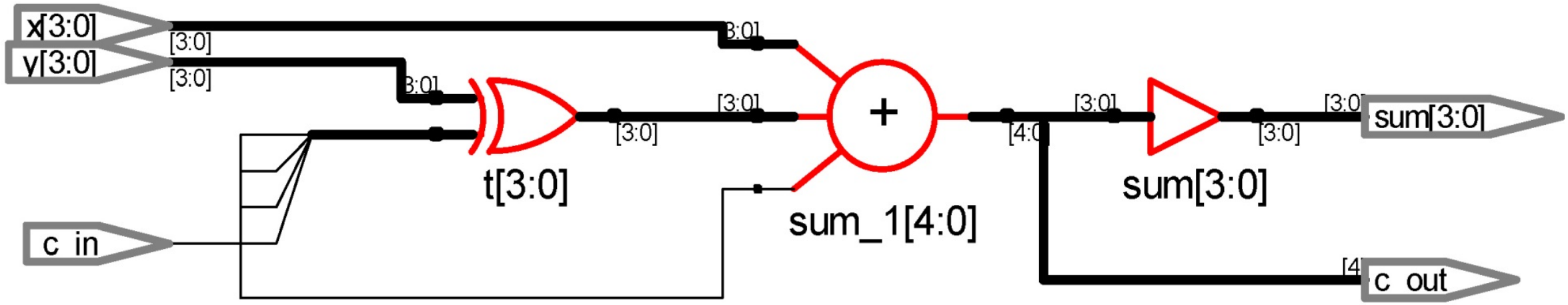
- Example: 4-Bit Full Adder




```
// specify the function of a 4-bit adder
    assign {c_out, sum} = x + y + c_in;
```

Operators: Concatenation/Replication

- Example: 4-Bit Two's Complement Adder




```
// specify the function of a two's complement adder
    assign t = y ^ {4{c_in}};
    assign {c_out, sum} = x + t + c_in;
```

Operators: Reduction

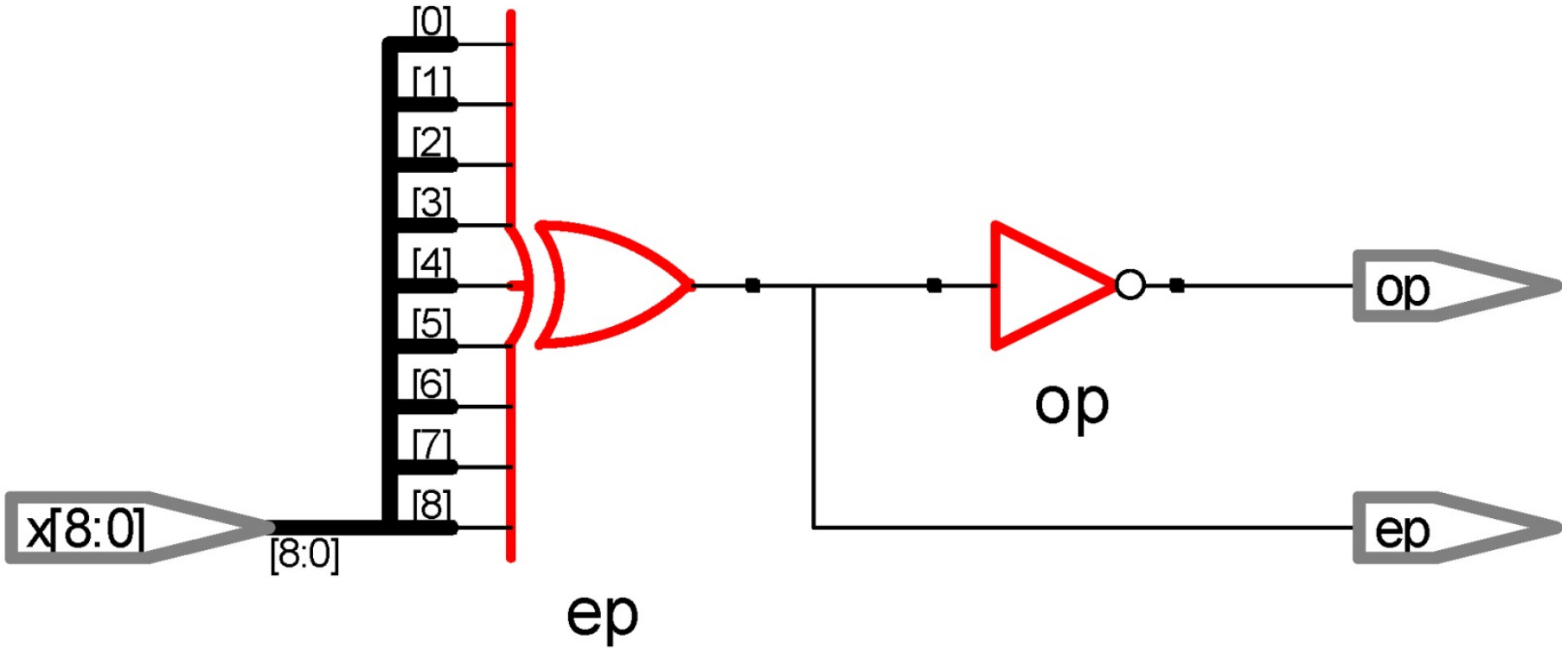
- Perform only on one vector operand
  - Carry out a bit-wise operation
  - Yield a 1-bit result
  - Work bit by bit from right to left

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^, ^~	Reduction exclusive nor

```
// A = 4'b1010

&A    // 1 & 0 & 1 & 0  -> 1'b0
|A    // 1 | 0 | 1 | 0  -> 1'b1
^A    // 1 ^ 0 ^ 1 ^ 0  -> 1'b0
```

Operators: Reduction

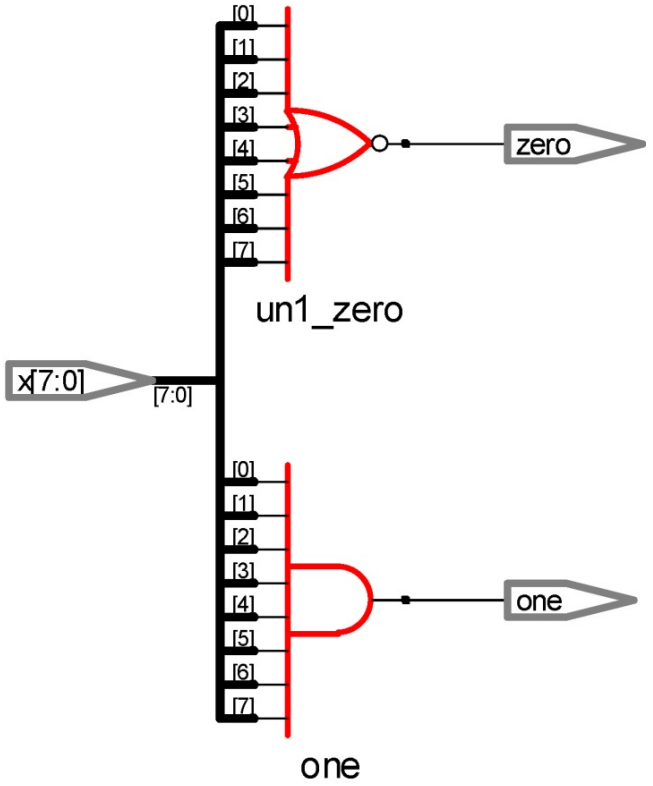
- Example: A 9-bit Parity Generator




```
// dataflow modeling using reduction operator
assign ep = ^x;     // even parity generator
assign op = ~ep;    // odd parity generator
```

Operators: Reduction

- Example: An All-Bit-Zero/One Detector




```
// dataflow modeling
   assign zero = ~(|x);  // all-bit zero
   assign one = &x;      // all-bit one
```

Operators: Relational

- The result is 1 if the expression is true and 0 if the expression is false
  - Return x if any operand bit is x or z

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
A <= B      // Evaluates to a logical 0
A > B       // Evaluates to a logical 1
Y >= X      // Evaluates to a logical 1
Y < Z       // Evaluates to an x
```

Operators: Equality

- Compare the two operands bit by bit
  - The shorter operand is zero-extended
  - Return 1 if the expression is true / 0 if the expression is false
  - The operators (==, !=)
    - yield an x if any operand bit is x or z
  - The operators (===, !==)
    - yield 1 if the two operands exactly match (‘x’ is also compared)
    - 0 if the two operands not exactly match

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

```
// A = 4, B = 3
// J = 4'b1010, K = 4'b1101
// S = 4'b1xxz, T = 4'b1xxz, U = 4'b1xxx

A == B     // false
J != K     // true
J == S     // x(unknown)
S === T    // true
T !== U    // true
```

Operators: Shift Ops

- Logical shift operators
  - 0 is added in empty space
- Arithmetic shift operators
  - 0(<<<) or MSB(>>>) is added in empty space

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

```
// A = 4'b1100

Y = A >> 1;   // 4'b0110
Y = A << 1;   // 4'b1000
Y = A << 2;   // 4'b0000
```

Operators: Conditional

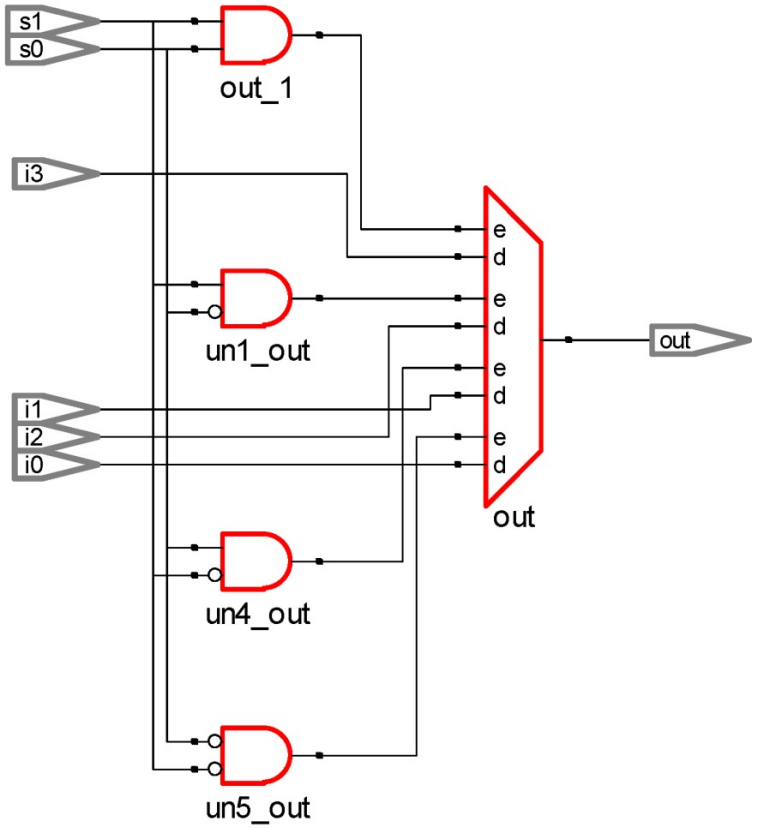
- Usage: `condition_expr ? true_expr: false_expr;`
  - If condition_expr = x or z: the result = true_expr & false_expr (0 and 0 gets 0, 1 and 1 ge ts 1, others gets x)
- Nesting is allowed
  - True_expr or Flase_expr can have condition
- For example

```
assign out = selection ? in_1: in_0;
```

```
// modeling tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

// modeling 2-to-1 mux
assign out = control ? in1 : in0;

// nesting
assign out = (A == 3) ? (control ? x : y) : (control ? m : n);
```

Operators: Conditional

- Example: A 4-to-1 MUX




```
// using conditional operator (?:)
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
```

Operators: Precedence

Operators	Operator Symbol	Precedence
Unary	`+ - ! ~`	Highest
Multiply, Divide, Modulus	`* / %`	
Add, Subtract	`+ -`	
Shift	`<< >>`	
Relational	`< <= > >=`	
Equality	`== != === !==`	
Reduction	`&, ~&`	
Logical	`^, ~^`	
	`|, ~|`	
	`&&`	
	`| |`	
Conditional	`? :`	Lowest

Outline

- Number Representation
- Data Types
- Dataflow Modeling with Continuous Assignments
- Operators
- Behavioral Modeling with `initial` and `always` Blocks
  - `initial` Block
  - `always` Block
  - Procedural Assignments: Blocking and Non-blocking
  - Timing Controls
  - `if-else` and `case` Statements
  - Loops: `while`, `for`, and `repeat`

Behavioral Modeling: `initial` Block

- Execute exactly once during simulation
- Start at simulation time 0 &
- Operate independently (Parallel)

```
reg  x, y, z;
initial  begin         // complex statement
         x = 1'b0;  y = 1'b1;  z = 1'b0;
#10      x = 1'b1;  y = 1'b1;  z = 1'b1;
end
initial x = 1'b0;  // single statement but not recommended
```

Behavioral Modeling: `initial` Block

```
module stimulus;
reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial
    #50 $finish;

endmodule
```

- Example: input stimulus

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

Behavioral Modeling: `always` Block

- Starts at simulation time 0
- Executes continuously during simulation
- Behavior modeling should be located inside of procedural constructs

```
reg  clock;
initial  clock = 1'b0;      //clock initialization
                            //with initial

always  #5 clock = ~clock; // cycle = 10, 100MHz

initial # 1000 $finish;     // $finish with initial
```

Behavioral Modeling: Procedural Assignments

- Syntax

  variable_lvalue = [timing_control] expression

  variable_lvalue <= [timing_control] expression

  [timing_control] variable_lvalue = expression

  [timing_control] variable_lvalue <= expression

- lvalue : reg, integer, real, time (**not net**!!)
- Two types
  - blocking
  - non-blocking

Behavioral Modeling: Procedural Assignments

- The bit widths
  - If expression is longer than lvalue → Keeping LSB
  - If expression is shorter than lvalue → Zero-extended in MSB

Behavioral Modeling: Procedural Assignments

- Type #1: Blocking assignments
  - Are executed in the specified order
  - Use the "=" operator
  - Similar with C code

```
// blocking assignments
initial begin
    x = #5 1'b0;     // at time 5
    y = #3 1'b1;     // at time 8
    z = #6 1'b0;     // at time 14
end
```

Behavioral Modeling: Procedural Assignments

- Type #1: Blocking assignments: An Example

```
//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1;     //Scalar assignments
        count = 0;               //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //initialize vectors

        #15 reg_a[2] = 1'b1;     //Bit select assignment with delay
        #10 reg_b[15:13] = {x, y, z}     //Assign result of concatenation to
                                         // part select of a vector
        count = count + 1;       //Assignment to an integer (increment)
end
```

Behavioral Modeling: Procedural Assignments

- Type #2: Non-blocking assignments
  - Use the "<=" operator
    - Same with operator "<=", distinguish by used position
  - Used to model several concurrent data transfers

```
reg  x, y, z;
// nonblocking assignments
initial begin
   x <= #5 1'b0;  // at time 5
   y <= #3 1'b1;  // at time 3
   z <= #6 1'b0;  // at time 6
end
```

Behavioral Modeling: Procedural Assignments

- Type #2: Non-blocking assignments: An Example

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1;              //Scalar assignments
        count = 0;                        //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a;     //Initialize vectors
        reg_a[2] <= #15 1'b1;             //Bit select assignment with delay
        reg_b[15:13] <= #10 {x, y, z};    //Assign result of concatenation
                                          //to part select of a vector
        count <= count + 1;               //Assignment to an integer (increment)
end
```

Timing Controls: Regular Delay Control

- Regular delay control
  - Defers the execution of the entire statement
  - For test bench

```
reg  x, y;
integer count;
    #25  y <= ~x;                 // at time 25
    #15  count <= count  + 1; // at time 40
```

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial
begin
     x = 0;              // no delay control
     #10 y = 1;          // delay control with a number. Delay execution of y = 1 by 10 units

     #latency z = 0;     // Delay control with identifier. Delay of 20 units
     #(latency + delta) p = 1;   // Delay control with expression
     #y x = x + 1;       // Delay control with identifier. Take value of
     #(4:5:6) q = 0;     // Minimum, typical and maximum delay values.
```

Timing Controls: Intra-Assignment Delay Control

- Intra-assignment delay control
  - Defers the assignment to the left-hand-side variable
  - Right side is already calculated

```
y = #25  ~x;                    // assign to y at time 25
count = #15 count  + 1;         // assign to count at time 40
```

```
y <= #25  ~x;                  // assign to y at time 25
count <= #15 count + 1;   // assign to count at time 15
```

```
//define register variables
reg x, y, z;

//intra assignment delays
initial
begin
    x = 0; z = 0;
    y = #5 x + z; //Take value of x and z at the time=0,
                  //evaluate x + z and then wait 5 time units
                  //to assign value to y
end
```

```
//Equivalent method with temporary
// variables and regular delay control
initial
begin
    x = 0; z = 0;
    temp_xz = x + z;
    #5 y = temp_xz;
end
```

Timing Controls: Event Timing Control

- Edge-triggered event control
- Level-sensitive event control
- What is an event? → Value change of register and net

Timing Controls: Edge-Triggered Event Control

- Edge-triggered event control
  - @(posedge clock)
  - @(negedge clock)

```
always @(posedge clock)  begin
       reg1 <= #25  in_1;
       reg2 <= @(negedge clock) in_2 ^ in_3;
       reg3 <= in_1;
end
```

Timing Controls: Event OR Control

- Event OR control
  - or
  - ,
  - * or (*) means any signals

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

Behavioral Modeling: `if … else` Statement

- Syntax

```
if (expr1)
   true_stmt1;
else if (expr2)
   true_stmt2;
     ..
  else
   default_stmt;
```

```
// e.g. 4-to-1 mux
module mux4_1 (out, in, sel);

    output        out;
    input  [3:0] in;
    input  [1:0] sel;

    reg           out;
    wire   [3:0] in;
    wire   [1:0] sel;

    always @(in or sel) begin
        if (sel == 2'b00)
            out = in[0];
        else if (sel == 2'b01)
            out = in[1];
        else if (sel == 2'b10)
            out = in[2];
        else
            out = in[3];
    end

endmodule
```

Behavioral Modeling: Synthesizable `if ... else`

- Usually synthesize into a 2-to-1 multiplexer
  - Nested if-else → priority-encoded, cascaded combination of multiplexers
- For combinational logic
  - Completely specified?

```
always @(enable or data)
    if (enable) y = data;  // infer a latch
```

- For sequential logic
  - Completely specified?

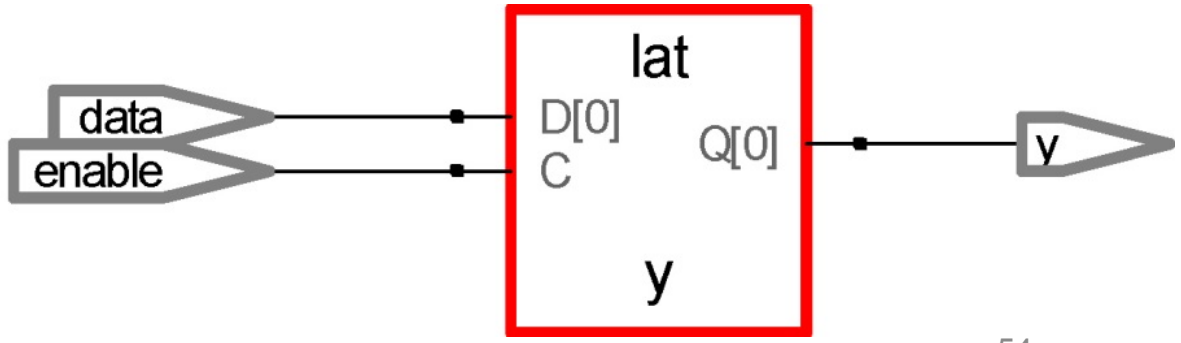
```
always @(posedge clk)
    if (enable) y <= data;
    else y <= y;      // a redundant expression
```

- Synthesis tool will remove the redundant expression

Behavioral Modeling: Synthesizable `if ... else`

- Latch Inference – Incomplete if-else Statements

```
// creating a latch
module latch_infer_if(enable, data, y);
input enable, data;
output reg y;

always @(enable or data)
    if (enable) y = data;  // infer a latch for y
```



Behavioral Modeling: case Statement

- Syntax

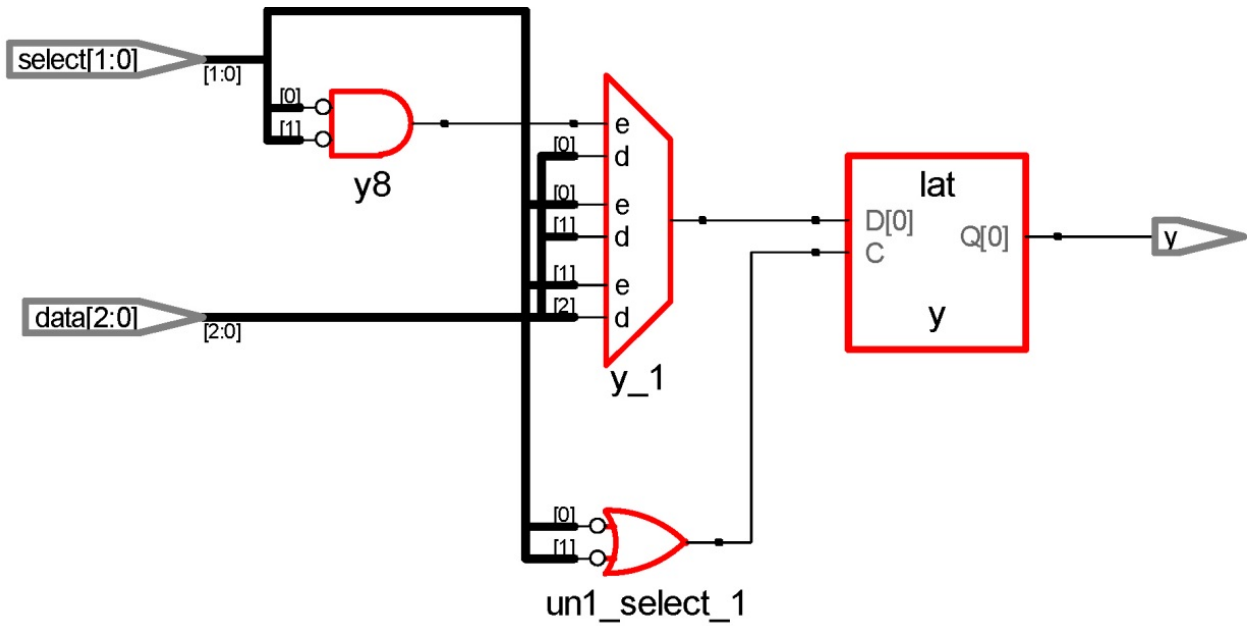
```
case (expr)
    item_1,… , item_n :    stmt1;
    item_n+1, …, item_m: stmt2;
    ..
    default:   default_stmt;
endcase
```

```
// e.g. 4-to-1 mux
module mux4_1 (out, in, sel);

    output       out;
    input  [3:0] in;
    input  [1:0] sel;

    parameter   SEL0 = 2'b00;
    parameter   SEL1 = 2'b01;
    parameter   SEL2 = 2'b10;

    reg          out;
    wire   [3:0] in;
    wire   [1:0] sel;

    always @(in or sel) begin
        case (sel)
            SEL0 : out = in[0];
            SEL1 : out = in[1];
            SEL3 : out = in[2];
            default: out = in[3];
        endcase
    end

endmodule
```

Behavioral Modeling: Synthesizable case

- Latch Inference – Incomplete case Statements

```
// Creating a latch
module latch_infer_case(select,
data, y);
…
output reg y;
always @(select or data)
    case (select)
        2'b00: y = data[select];
        2'b01: y = data[select];
        2'b10: y = data[select];
        //       default: y =
2'b11;
    endcase
```



Behavioral Modeling: `while` Statement

- Syntax

```
while (condition_expr) statement;
```

```
while (count < 12) count <= count + 1;
while (count <= 100 && flag) begin
    // put statements wanted to be carried out here
end
```

Behavioral Modeling: `while` Statement

- Example: while Statement

```
// count the zeros in a byte
integer i, out;
always @(data) begin
   out = 0; i = 0;
   while (i <= 7) begin  // simple condition, but complex
                         // conditions also possible
        if (data[i] == 0) out = out + 1;
        i = i + 1;
   end
end
```

Q: Can the if … be replaced with out = out + ~data[i] ? → Yes

Behavioral Modeling: `for` Statement

- Syntax

```
for (init_expr; condition_expr; update_expr) statement;
```

```
init_expr;
while (condition_expr) begin
      statement;
      update_expr;
end
```

- Simple, but range of use is narrow (only for fixed start and end)

Behavioral Modeling: for Statement

- Example: for Statement

```
// count the zeros in a byte
integer i;
always @(data) begin
    out = 0;
    for (i = 0; i <= 7; i = i + 1) // simple condition
         if (data[i] == 0) out = out + 1;
end
```

Behavioral Modeling: repeat Statement

- Syntax

```
repeat (counter_expr) statement;
```

- Only for fixed value

```
//Illustration 1: increment and display
//count from 0 to 127
integer count;
initial
begin
    count = 0;
    repeat(128)
    begin
         $display("Count = %d", count);
         count = count + 1;
    end
end
```

```
//Illustration 2 : Data buffer module example
//After it receives a data_start signal.
//Reads data for next 8 cycles.
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;

always @(posedge clock)
begin
  if(data_start) //data start signal is true
  begin
    i = 0;
    repeat(cycles) //for next 8 cycles
    begin
      @(posedge clock) buffer[i] = data; // wait until
                                         // next posedge
      i = i + 1;
    end
  end
end
endmodule
```

Behavioral Modeling: Synthesizable Loops

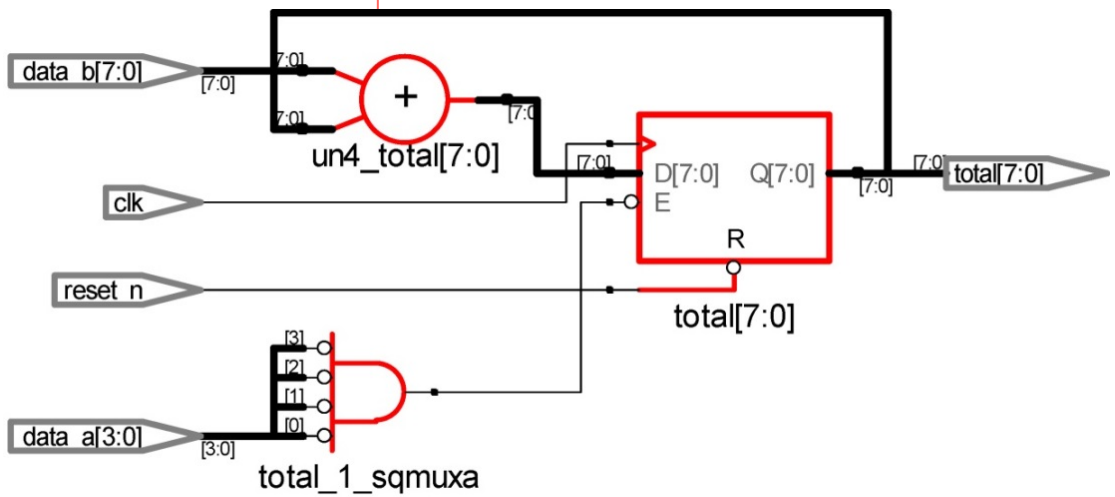
- Loop statements *for*, *while*, and *forever* are synthesizable except that *while* and *forever* must contain timing control @(posedge clk) or @(negedge clk)
- *repeat* is generally not synthesizable

```
// an N-bit adder using for loop.
module nbit_adder_for( x, y, c_in, sum, c_out);
parameter N = 4;      // default size
…
integer i;
…
always @(x or y or c_in) begin
    co = c_in;
    for (i = 0; i < N; i = i + 1)
         {co, sum[i]} = x[i] + y[i] + co; // At the elaboration
phase, the always block is expanded into the following statements
    c_out = co;
end
```

Behavioral Modeling: Synthesizable Loops

- Loop Structures – An Incorrectly Synthesizable Example

```
// a multiple cycle example --- This is an incorrect version.
…
parameter N = 8;
parameter M = 4;
input   clk, reset_n;
…
integer i;
// what does the following statement do?
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) total <= 0;
    else  for (i = 0; i < M; i = i + 1)
                if (data_a[i] == 1) total <= total + data_b;
end
```



Behavioral Modeling: Synthesizable Loops

- Loop Structures – An Incorrectly Synthesizable Example

```
// a multiple cycle example --- This is an incorrect version.
…
parameter N = 8;
parameter M = 4;
input   clk, reset_n;
…
integer i;
// what does the following statement do?
always @(posedge clk or negedge reset_n) begin
    if (!reset_n) total <= 0;
    else  begin
       if (data_a[0] == 1) total <= total + data_b;
       if (data_a[1] == 1) total <= total + data_b;
       if (data_a[2] == 1) total <= total + data_b;
       if (data_a[3] == 1) total <= total + data_b;
    end
end
```