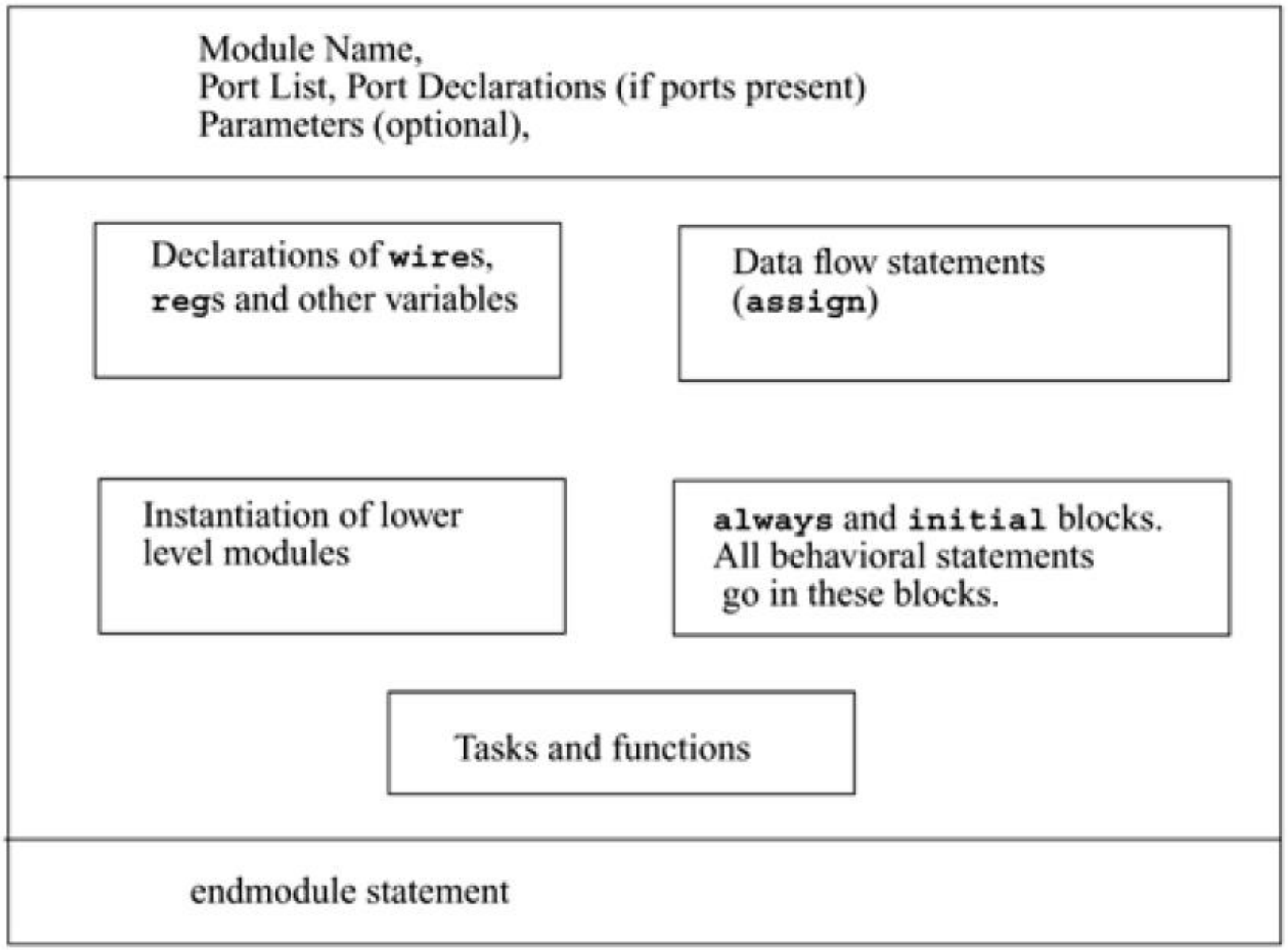
midterm exam. review

S

Basic Concept of Verilog

Components of a Verilog Module


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

```
module M (P1, P2, P3, P4);
input P1, P2;
output [7:0] P3;
inout P4;

reg [7:0] R1, M1[0:1023];
wire W1, W2, W3, W4;
wire [3:0] W5;
parameter C1=const;

initial
begin : blockname
  // statements
end

always
begin
  // behavioral level statements
end

// dataflow level assignments
// continuous assignment
assign W1 = expr ;

// structural level statements
// module instances
COMP U1 (.PP2(W2), .PP1(W1);
COMP U2 (W3, W4);

task T1; // common task in this module
input A1;
inout A2;
output A3;
begin
  // statements
end
endtask

function [7:0] F1; // common function in this module
input A1;
begin
  // statements
end
endfunction

endmodule
```

Structural, Dataflow, Behavioral Modeling

- structural modeling
  - instantiation of other modules
  - e.g. FA instance_name (sum, cout, a, b, cin);
- dataflow modeling
  - continuous assignment **assign**
  - e.g., **assign** {cout, sum} = a + b + cin;
- behavioral modeling
  - procedural assignments inside **always @(…)** block
  - e.g.

```
reg [31:0] sum;
reg cout;

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

Statements inside Behavioral Modeling

```
#delay  // neglected during synthesis
wait (expression)
@(A or B or C)
@(posedge clk)

Reg = expression;    // blocking assignment
Reg <= expression;  // non-blocking assignment

if (condition1)   … else if (condition2) … else …

case (selection)
  choice1 : begin … end
  choice2, choice 3: …
  …
  default : …
endcase

for (i=1; i<max; i=i+1) begin … end

repeat (8) …

while (condition)  …
```

Verilog Operators

a = **~**b // **not** b (unary operator); only one operand: b

 // output could be multiple bits, the same bit-width as the input

a = b **&&** c; // logical **AND** operation (binary operator)

  // two operands: b, c

  // output a is either TRUE (logic 1) or FALSE (logic 0)

// cp. a = b **&** c; // bitwise (bit-by-bit) operator

// output of bit-wise operator could be multi-bit

// with the same bit-width as that of inputs

a = b**?** c **:** d; // **MUX2** (ternary operator)

        // a=c if b=1,

        // a=d if b=0

  // three operands: b, c, d

Other operators

- arithmetic operators (+, -, *, /, %, **)
- relational operators (<, >, <=, >=)
- equality operators (==, !=, ===, !==)
- logical operators (&&, ||, !)
- bitwise operators (~, &, |, ^, ~^)
- reduction operators (&, ~&, , |, ~|, , ^, ~^)
- shift operators (>>, <<, >>>, <<<)
- concatenation operator ({ })
- conditional operator (? :)

Equality Operator: Logical vs. Case

- logical equality operator (==)

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

- case equality operator (===)

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

reduction operator

- perform bitwise operator on all bits of a single vector and yield a 1-bit result
- e.g.

```
X = 4'b1010
&X   // = 1&0&1&0 = 1'b0
|X    // = 1|0|1|0 = 1'b1
^X    // = 1^0^1^0 = 1'b0
```

Concatenation Operator Replication Operator

- A=1’b1, B=2’b01, C=2’b10

- concatenation operator

  Y={B,C} = 4’b0110
- replication operator

  Y={ 4{A}, 2{B} }=8’b1111_0101

Number

unsized numbers (decimal by default)

sized numbers:

<size>'<base format> <number>

<size>: the number of bits

'<base>: 'd (decimal),
'h (hexadecimal),
'b (binary),
'o (octal)

' is the single quote symbol, ***not*** back quote symbol **`** which will be used in compiler directive such as `include

e.g.

4'**b**1111 // 4-bit binary number $1111_2=15_{10}$

12'**h**abc // 12-bit hexadecimal number = $1010\_1011\_1100_2$

16'**d**255 // 16-bit decimal number = 0000000011111111_2

Four-Valued Logic System

- Signal and variable values represented using a 4-valued logic system
  - 1’b0: 1-bit binary logic 0 value
  - 1’b1: 1-bit binary logic 1 value
  - 1’bx: 1-bit binary ‘x’ or ‘X’ value
    - ✓ Unknown value, produced for uninitialized values or values driven to conflicting values by more than one signal source
  - 1’bz: 1-bit binary ‘z’ or ‘Z’ value
    - ✓ High impedance value, produced when a wire is disconnected from all signal sources driving that wire, i.e., floating node

data types (**wire** and **reg**)

- Nets (**wire**): connections between hardware elements
  - nets have values continuously driven by the outputs of devices that they are connected to

  eg.: **wire** a, b, c;

  wire a = b & c; // equivalent **to** **wire** a; **assign** a = b & c;
- Registers (**reg**): data storage element
  - a variable that can hold a value in procedural blocks starting with **always** or **initial**
  - *not necessarily* mean a flip-flop (one-bit register) in real circuit

  eg.: **reg** reset;
- SystemVerilog use **logic** to replace 4-value **wire** and **reg** to avoid confusion
  - not all **reg** data types synthesize into hardware registers

data types (vectors)

- Vectors: multiple bit widths (default is scalar)

```
wire [7:0] bus;       // 8-bit signal, bus[7] is the MSB, bus[0] is the LSB
wire [31:0] word;       // 32-bit word
reg [255:0] data;     // 256-bit word register, data[255] is MSB
reg [0:40] v_addr;  // v_addr[0] is MSB
```

- Vector Part Select

```
word[7:0]    // the least significant byte of the 32-bit vector word
v_addr[0:1]  // two most significant bits of the vector v_addr
```

- variable vector part select

[<starting_bit> **+:** width] : part-select *increments*

[<starting_bit> **-:** width] : part-select *decrements*

starting bit can be varied, but the width must be constant

```
eg1.: reg [255:0] data1;   // MSB->LSB with decreasing index order
      reg [0:255] data2;   // MSB->LSB with increasing index order
      reg [7:0] byte;

      byte=data1[31-:8]  // data1[31:24]
      byte=data1[24+:8] // data1[31:24]
      byte=data2[31-:8]  // data2[24:31]
      byte=data2[24+:8] // data2[24:31]

eg2.  reg [255:0] data1;
      for (j=0; j<=31; j=j+1) byte = data1[(j*8)+:8] ;
      // sequence is [7:0], [15:8], [23:16], …, [255:248]
```

register data types (integer, real, time)

- **integer**: register data type used for signed quantity
  - **reg** store a multi-bit signal which could mean a *unsigned* quantity, while **integer** store *signed* quantity

    eg., **integer** counter;   counter = -1; // used as a counter
- **real**: real number constant or register data type

  eg. **real** delta;

    delta = 4e10; delta = 2.14; // delta is a real variable
- **time**: register data type to store simulation time

  eg. **time** save_sim_time;

    save_sim_time = **$time**; // define a time variable

      // system task **$time** is invoked to get current simulation time
- **real** and **time** are non-synthesizable codes for simulation only

signed vs. unsigned

- unsigned (default Verilog data type)

```
wire [3:0] x;
wire [7:0] y;

assign y = {4’b0000, x};
```

```
wire [3:0] x;
wire [7:0] y;

 assign y = x;   // zero padded
```

- signed (new data type added in 2001)

```
wire signed [3:0] x;
wire signed [7:0] y;

assign y = {4{x[3]}, x};
```

```
wire signed [3:0] x;
wire signed [7:0] y;

assign y = x; // sign-extended
```

```
wire [11:0] s1;
wire signed [11:0] s2;

assign s2 = $signed(s1);   // convert to signed number
// assign s1 = $unsigned(s2);  // convert to unsigned numer
```

data types (array and memory)

- Arrays

  reg [4:0] port_id **[0:7];**

  // array of 8 elements, with each element 5-bit wide

  port_id[6] = 5`b00000; // the array element with index=6

  reg [63:0] array_4d **[15:0] [7:0] [7:0] [255:0];**

  // 4-d array, new Verilog, IEEE1364-2001

- Memory (register files, RAM, ROM)

  reg [7:0] membyte [0:1023]; // 1K bytes (1024x8) memory

  data = membyte[511] // fetch 1 byte whose address id=511

- Use tool-supported memory generator/compiler (instead of registers) to create hardware SRAM memory instead of flip-flop-based memory array

bit-vector vs. array

- wire [31:0[x;
  - declare a 32-bit signal x
  - all 32 bits are packed into a word x
  - more efficient in simulation
- wire x [31:0];
  - declare an array x with 32 elements x[31], x[30], …, x[1], x[0] where each element is 1-bit
  - not efficient in simulation
- wire [31:0] x[31:0];
  - declare an array x with 32 elements x[31], x[30], …, x[1], x[0] where each element is 32-bit

data type (parameter)

- Parameters
  - allows constants to be defined in a module
  - parameter values for each module instance can be overridden individually at compile time

parameter cache_line_width = 256;

// constant defined inside a module

// parameter values can be changed

// *at module instantiation* ***#()*** or *by using* ***defparam***

localparam // parameter values cannot be changed

state1=4’b0001,

state2=4’b0010,

state3=4’b0100,

state4=4’b1000;

Notes on Dataflow Statements (1/2)

- dataflow statements with continuous assignment usually describe “combinational logic”
  - pure feed-forward datapath
  - LHS signal should be different from the RHS signals
  - e.g., assign c=a+b; // describe an adder
  - e.g., assign d=c+1; // describe an adder with one input=1
  - e.g., assign c=c+1; // c in LHS and RHS !!! not OK
    - ✓ a loop in the datapath
  - use register in behavioral modeling to describe a counter
  - e.g.,

```
// behavioral statements are inside always block
// draw the hardware of the following statements
reg [7:0] c;
always @ (posedge clk)
   c <= c+1;
```

Notes on Dataflow Statements (2/2)

- all dataflow statements are concurrent
  - different order of statements results in same hardware
  - e.g.,

```
// an adder, a subtractor,
// and a multiplier
wire [7:0] a, b, c, d
assign c=a+b;
assign d=a-b;
assign e=c*d;
```

```
// a subtractor, a multiplier,
// and a adder
wire [7:0] a, b, c, d
assign d=a-b;
assign e=c*d;
assign c=a+b;
```

- similar behavioral statements could describe the same combinational logic
  - e.g., inside always block

```
reg [7:0] a, b, c, d;
always @(a, b, c, d)
begin
   c=a+b;
   d=a-b;
   e=c*d;
end
```

Verilog System Tasks

System Tasks $task_name

- **$display**
  - display values of variables or strings or expressions

  $display(“at time **%d**   address is **%h**”, **$time**, addr);
- **$monitor**
  - continuously monitor a signal *when its value is changed*
  - unlike **$display**, **$monitor** only needs to be invoked once
  - use **$monitoroff** and **$monitoron** to disable/enable

  $monitor (**$time**, “clock = **%b** reset = **%b**”, clk, rst);
- **$stop**
  - suspend the simulation for debug
- **$finish**
  - terminate simulation

$strobe vs. $display

- **$display**
  - if many other statements are executed in the same time unit as the task **$display**, the execution order of the statements and **$displa**y task is non-deterministic (depending on all the scheduled events at that time)
- **$strobe**
  - task **$strobe** is always executed after all other assignment statements in the same time unit have executed
  - provide synchronization mechanism to display data only after all other assignments are executed

Compiler Directives ` (back quote)

- **`define**
  - define text macro for later text macro substitution

  `define WORD_SIZE 32

  // used as **`WORD_SIZE** in the code such as

  // **wire** [**`WORD_SIZE**-1:0] a, b, c;
- **`include**
  - include a Verilog source file in another file

  `include header.v   //include the file header.v
- **`ifdef**
  - conditional compilation

  `ifdef TEST **module** test;

  // compile module test only if text macro TEST

  // is defined using **`define** TEST
- **`timescale**

  `timescale 100ns/1ns

`define `ifdef `else `endif

```
`define TEXT_MACRO_NAME // comment out this line if do not want this definition
…
// do not confuse with if … else in always procedure block which synthesize MUX
// compiler directive `ifdef … `else … does not synthesize MUX hardware
`ifdef TEXT_MACRO_NAME   // conditional compilation with `else
  // compile the following statements here if TEXT_MACRO_NAME is defined
  …
`else     // `else compiler directive is optional
  // compile the following statements here if TEXT_MACRO_NAME is not defined
  …
`endif

`ifdef TEXT_MACRO_NAME  // conditional  compilation without `else
  // compile the following statements here if TEXT_MACRO_NAME is defined
  // the following statements are skipped if TEXT_MACRO_NAME is not defined
  …
`endif
```

Structural-Level (Gate-Level) Modeling

structural modeling

- component instantiation
  - Verilog *built-in* gate primitives (**not, and, or, nand, nor, xor, xnor**, …)
  - module instantiation (from previously user defined modules)
- component connectivity
  - connecting by ordered lists (ordered by position)

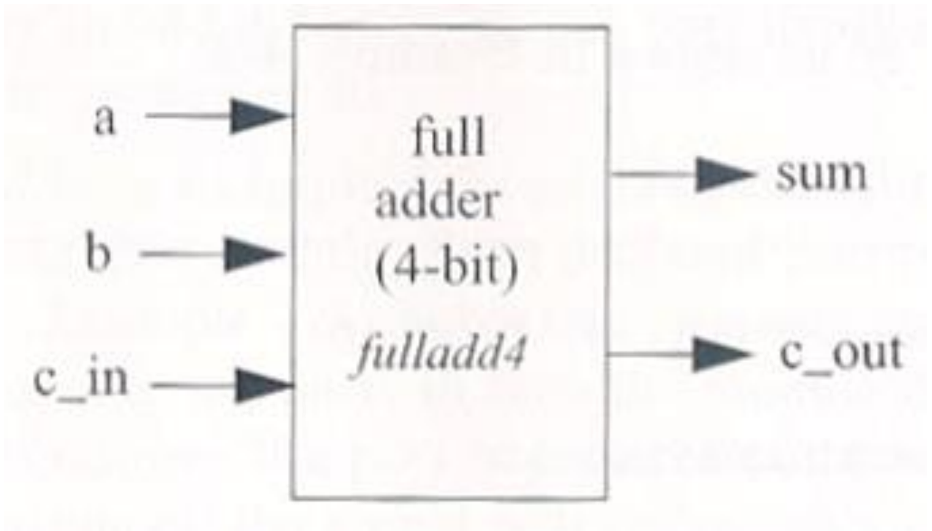
    ```
    fulladd4   fa_ordered  (EXT_SUM, EXT_C_OUT, EXT_A,  EXT_B, EXT_C_IN)
    ```
  - connecting by name (order by name)

    ```
    fulladd4   fa_byname
    (.c_out(EXT_C_OUT), .sum(EXT_SUM),  .b(EXT_B), .c_in(EXT_C_IN), .a(EXT_A));
    ```

Port Declaration (1995 vs. 2001)

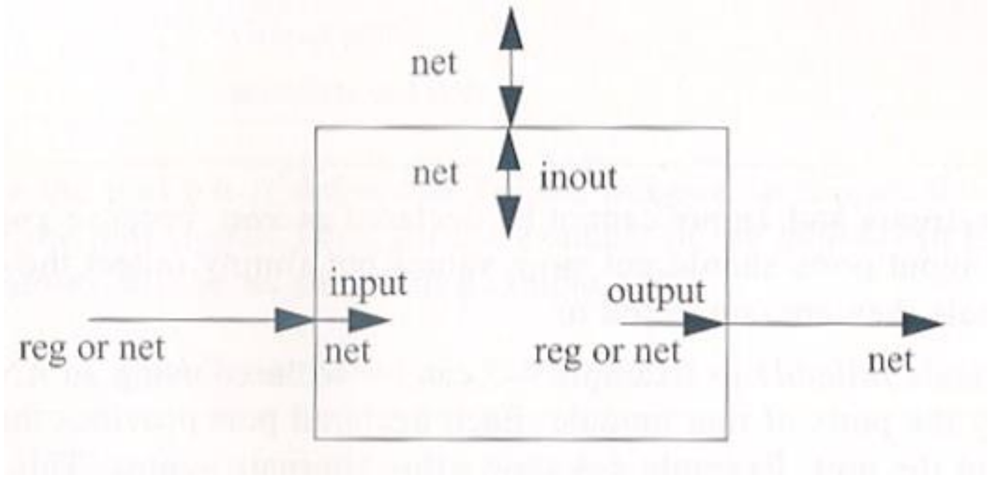
```
// IEEE 1364-1995 (old)
module fulladd4(sum, c_out, a, b, c_in);
parameter width = 4;
output [width-1:0] sum;
output c_out;
input [width-1:0] a, b;
input c_in;
reg [width-1:0] sum;
reg c_out;
…
endmodule
```

```
// ANSI C style (IEEE 1364-2001)
module
    # (parameter width =4)
    fulladd4 (
    output reg [width-1:0] sum,
    output reg c_out,
    input [width-1:0] a, b,
    input c_in
);
…
endmodule
```

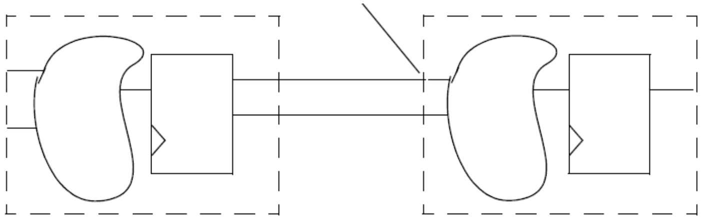


Port Connection Rules

- all port declarations are implicitly declared as net **wire**
- **input** and **inout** cannot be declared as **reg**
- **input**
  - internally (inside the module), always be of the type **wire**
  - externally (outside the module), can be connected to **reg** or **wire**
- **output**
  - internally, can be **reg** or **wire**
  - externally, always connected to **wire**
- **inout**
  - always be **wire** internally or externally

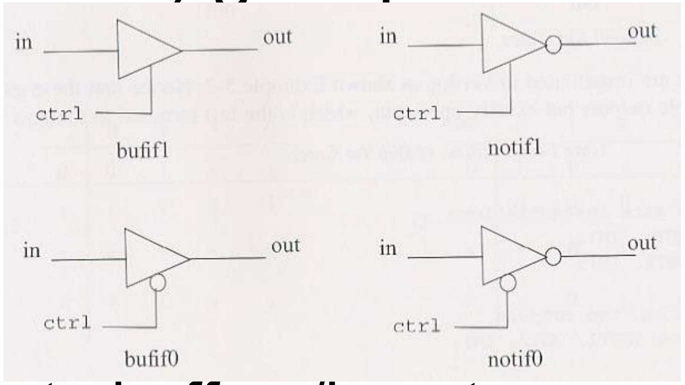


Latched module provides Enough driving strength, And break critical path



Verilog predefined gates

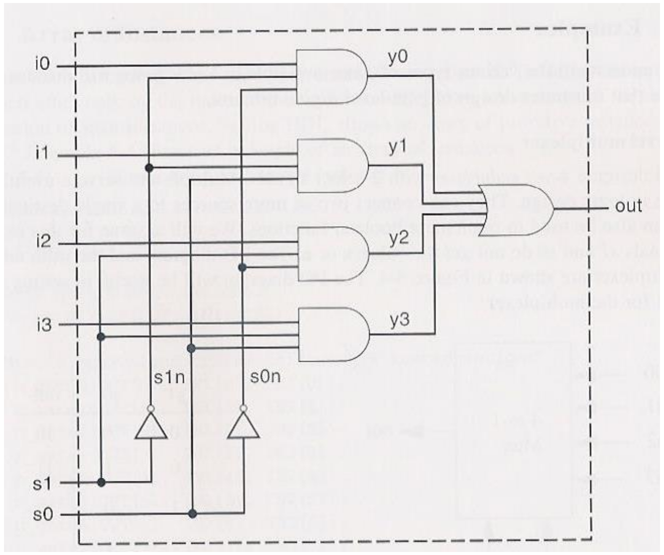
- basic logic gate as predefined (built-in) gate primitives
  - **and, or, nand, not, xor, xnor**
    - ✓eg., **and** a1(out, in1, in2);
  - **buff, not**
    - ✓eg., **buff** b1(out1, in1);
  - **bufif1, bufif0, notif1, notif0** // tristate buffers/inverters
    - ✓eg., bufif1 b1(out, in, ctrl);

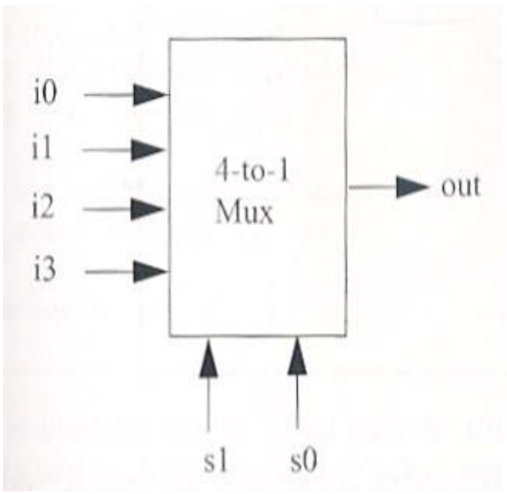


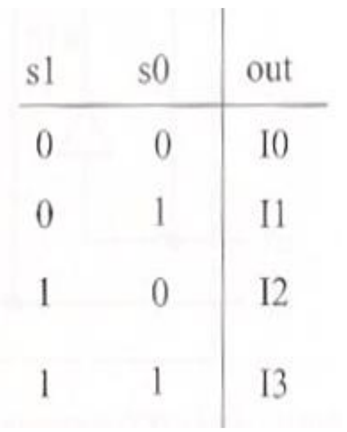
- array of instances

```
wire [7:0] out, in1, in2;

nand n_gate [7:0] (out, in1, in2);
/* equivalent to following 8 instances
nand n_gate0 (out[0], in1[0], in2[0];
nand n_gate1 (out[1], in1[1], in2[1];
…
nand n_gate7 (out[7], in1[7], in2[7];   */
```

4:1 Multiplexer (MUX4)







s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

```
// structural level modeling of MUX
module mux4_to_1
          (out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3, s0, s1;
wire s1n, s0n, y0, y1, y2, y3;
      not (s1n, s1);
      not (s0n, s0);
      and (y0, i0, s1n, s0n);
      and (y1, i1, s1n, s0);
      and (y2, i2, s1, s0n);
      and (y3, i3, s1, s0);
      or (out, y0, y1, y2, y3);
endmodule
```

```
// behavioral level modeling of MUX
module mux4_to_1
          (output reg out, input i0, i1, i2, i3, s1, s0);

// output wire out,  assign out = s1 ? (s0? I3 : I2) : (s0 ? I1:i0);
//
always @ (i0, i1, i2, i3, s0, s1) // sensitivity list
begin
     case ({s1, s0})
       2'b00: out = i0;
       2'b01: out = i1;
       2'b10: out = i2;
       2'b11: out = i3;
       default: $display("invalid control signals");
     endcase
end
endmodule
```

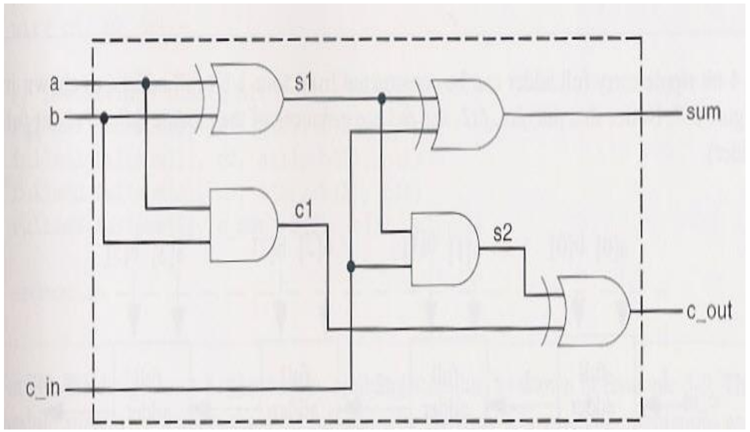
structural MUX4 using bufit1

- use Verilog built-in gate primitives bufif1 and bufif0 to construct MUX2

- use three MUX2 to construct MUX4

1-bit Full Adder

```
// structural (gate)level modeling

module fulladd(sum, c_out, a, b, c_in);
output sum, c_out;
input a, b, c_in;
wire s1, c1, c2;

      xor (s1, a, b); // assign s1=a^b;
      and (c1, a, b); // assign c1=a&b;
      xor (sum, s1, c_in);
      and (c2, s1, c_in)
      xor (c_out, c2, c1);

endmoudle
```

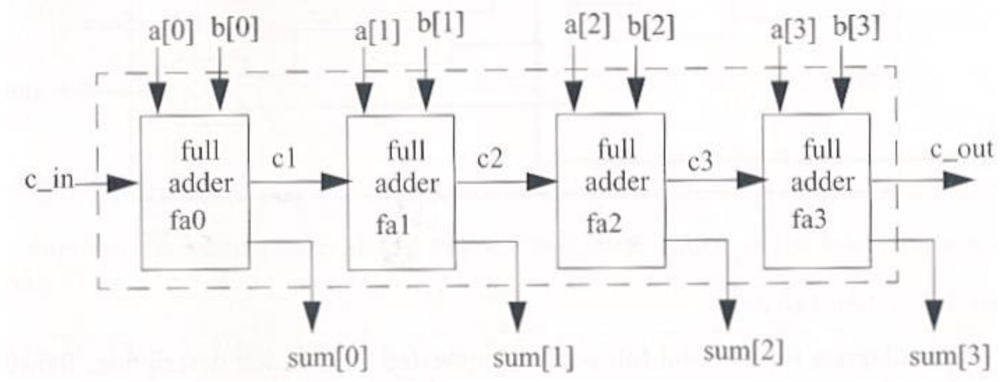


```
// dataflow modeling,  continuous assignment

module fulladd (sum, c_out, a, b, c_in);
output sum, c_out;
input a, b, c_in;

assign {c_out, sum} = a + b + c_in;
// synthesized gate netlist depends on
// area/delay constraints from users

endmoudle
```

```
// behavioral level modeling

module fulladd (sum, c_out, a, b, c_in);
output  reg  sum, c_out;
input a, b, c_in;

always @ (a or b or c_in)
    {c_out, sum} = a + b + c_in;

endmoudle
```

4-bit Ripple Carry Adder (RCA)

```
// structural level modeling
module fulladd4 (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in

wire c1, c2,c3;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

      fulladd   fa0 (sum[0], c1, a[0], b[0], c_in);
      fulladd   fa1 (sum[1], c2, a[1], b[1], c1);
      fulladd   fa2 (sum[2], c3, a[2], b[2], c2);
      fulladd   fa3 (sum[3], c_out, a[3], b[3], c3);

endmoudle
```



```
// dataflow level modeling
// using continuous assignment
module FA_dataflow
    (output [3:0] sum,
     output  c_out,
     input [3:0] a, b,
     input c_in);

     assign {c_out, sum} = a+b+c_in;
// synthesized gate netlist depends on
// area/delay constraints from users

endmodule
```

```
// behavioral level modeling
module FA_behavior
    (output reg [3:0] sum,
     output reg c_out,
     input [3:0] a, b,
     input c_in);

     always @(a, b, c_in)
            {c_out, sum} = a+b+c_in;

endmodule
```

RCA with generate loop

- use **generate** loop to duplicate module instances
  - e.g. in FA-cascaded RCA
- or simple
  - **array of instances**

```
// structural level modeling
module RCA (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in

wire c1, c2,c3;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

fulladd   fa0 (sum[0], c1, a[0], b[0], c_in);
fulladd   fa1 (sum[1], c2, a[1], b[1], c1);
fulladd   fa2 (sum[2], c3, a[2], b[2], c2);
fulladd   fa3 (sum[3], c_out, a[3], b[3], c3);

endmoudle
```

```
module RCA_generate (sum, c_out, a, b, c_in);
parameter N=4;
output [N-1:0] sum;
output c_out
input [N-1:0] a, b;
input c_in;
wire [N:0] c;

assign c[0]=c_in;

genvar i;
generate
   for (i=0; i<=N-1; i=i+1)
      begin: FA_loop   // block named “FA_loop”
         fulladd  fa (sum[i], c[i+1], a[i], b[i], c[i]);
         // fulladd FA_loop[0].fa ( sum[0], c[1], a[0], b[0], c[0] );
         // fulladd FA_loop[1].fa ( sum[1], c[2], a[1], b[1], c[1] );
         // fulladd FA_loop[2].fa ( sum[2], c[3], a[2], b[2], c[2] );
         // fulladd FA_loop[3].fa ( sum[3], c[4], a[3], b[3], c[3] );
      end
endgenerate

// fulladd FA_loop[0:3] (sum[0:3], c[1:4], a[0:3], b[0:3], c[0:3]);

assign  c_out = c[N];

endmodule
```

generate blocks

- allow Verilog code to be generated *dynamically* at the elaboration time (before simulation)
  - HDL compiler usually contain two stages: compilation and elaboration
- help the creation of parameterized models
- applications
  - same operation or module instance is repeated
    - ✓ eg., 64-b ripple carry adder (cascaded of 64 FA cells)
  - certain Verilog code is conditionally included based on parameter definitions
    - ✓ e.g., Booth or non-Booth multiplier based on bit-width

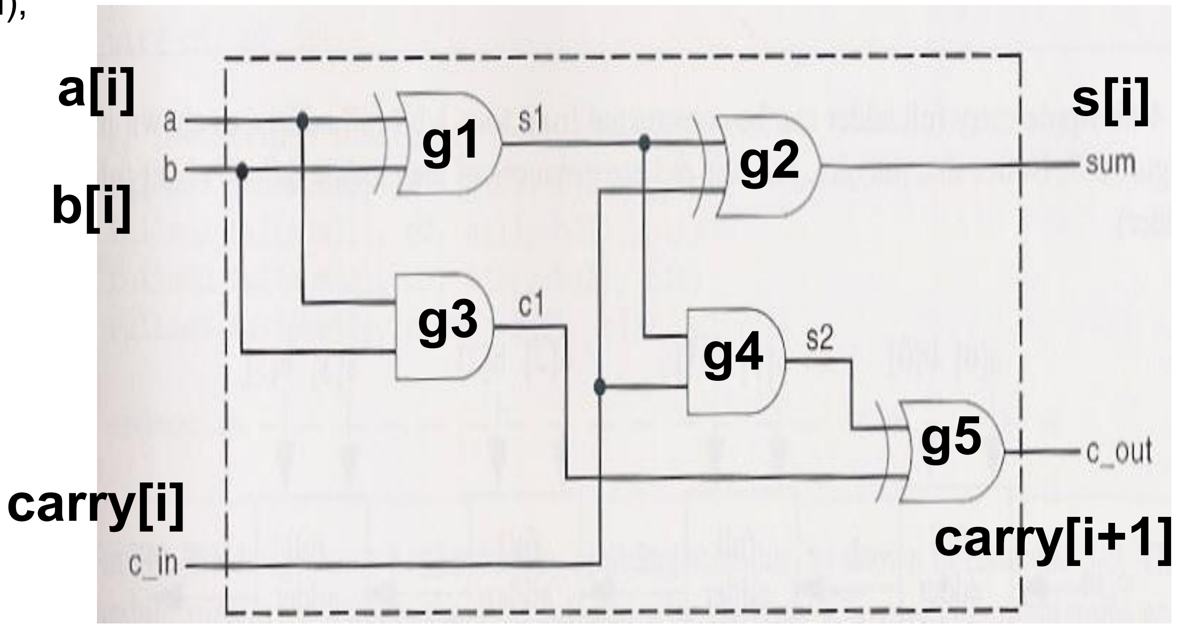
generate loop

- allow multiple instantiation, or duplication of statements
- note that **generate** should be used outside of procedural blocks (**always**, **initial**)

```
module bitwise_xor(out, i0, i1);
parameter N=32;
output [N-1:0] out;
input [N-1:0] i0, i1;

genvar j; // var. j is local
generate
      for (j=0; j<N; j=j+1)
          begin: xor_loop
             xor g(out[j], i0[j], i1[j]);
          end
// hierarchical name referencing:
// xor_loop[0].g, xor_loop[1].g,  ...,
endgenerate
// xor xor_loop[0].g(out[0], i0[0], i1[0]);
// xor xor_loop[1].g(out[1], i0[1], i1[1]);
…
// xor xor_loop[31].g(out[31], i0[31], i1[31]);
endmodule
```

```
// alternate style (behavioral level)
reg [N-1:0]  out;

genvar j;
generate
      for (j=0; j<N; j=j+1)
      begin: bit
        always @ (i0[j] or i1[j])   out[j] = i0[j] ^ i1[j];
      end
// always @(i0[0], i1[0])     out[0]=i0[0]^i1[0];
// always @(i0[1], i1[1])     out[1]=i0[1]^i1[1];
…
// always @(i0[31], i1[31]) out[31]=i0[31]^i1[31];
endgenerate
```

generated ripple carry adder



```
module ripple_adder(co, sum, a0, a1, ci);
parameter N=4;
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
wire [N:0] carry;

assign carry[0]=ci;

genvar i;
generate
        for (i=0; i<N; i=i+1)
        begin: r_loop
                wire s1, c1, t3;               // the following hierarchical instance names are generated
                xor g1(s1, a0[i], a1[i]);        //  xor r_loop[0].g1(…);   xor r_loop[1].g1(…); …
                xor g2(sum[i], s1, carry[i]);    //  xor r_loop[0].g2(…);   xor r_loop[1].g2(…);  …
                and g3(c1, a0[i], a1[i]);        //  and r_loop[0].g3(…);   and r_loop[1].g3(…);  …
                and g4(s2, s1, carry[i]);        //  and r_loop[0].g4(…);   and r_loop[1].g4(…);  …
                xor g5(carry[i+1], c1, s2);      //  xor r_loop[0].g5(…);   xor r_loop[1].g5(…);  …
        end
endgenerate

assign co = carry[N]

endmodule
```

RCA with generate loop

- use generate loop to duplicate module instances
  - e.g. in FA-cascaded RCA
- or simple
  - array of instances

```
// structural level modeling
module RCA (sum, c_out, a, b, c_in);

output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in

wire c1, c2,c3;
// construct 4-bit adder fullladd4 from
// previously defined module fulladd

fulladd   fa0 (sum[0], c1, a[0], b[0], c_in);
fulladd   fa1 (sum[1], c2, a[1], b[1], c1);
fulladd   fa2 (sum[2], c3, a[2], b[2], c2);
fulladd   fa3 (sum[3], c_out, a[3], b[3], c3);

endmoudle
```

```
module RCA_generate (sum, c_out, a, b, c_in);
parameter N=4;
output [N-1:0] sum;
output c_out
input [N-1:0] a, b;
input c_in;
wire [N:0] c;

assign c[0]=c_in;

genvar i;
generate
  for (i=0; i<=N-1; i=i+1)
    begin: FA_loop  // block named “FA_loop”
      fulladd  fa (sum[i], c[i+1], a[i], b[i], c[i]);
      // fulladd FA_loop[0].fa ( sum[0], c[1], a[0], b[0], c[0] );
      // fulladd FA_loop[1].fa ( sum[1], c[2], a[1], b[1], c[1] );
      // fulladd FA_loop[2].fa ( sum[2], c[3], a[2], b[2], c[2] );
      // fulladd FA_loop[3].fa ( sum[3], c[4], a[3], b[3], c[3] );
    end
endgenerate

// fulladd FA_loop[0:3] (sum[0:3], c[1:4], a[0:3], b[0:3], c[0:3]);

assign  c_out = c[N];

endmodule
```

generate case

```
module adder (co, sum, a0, a1, ci);

parameter N=4;

output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;

// instantiate different modules depending on input bitwidth
generate
        case (N)
                1 :  adder_1bit       adder1(co, sum, a0, a1, ci);
                2 :  adder_2bit       adder2(co, sum, a0, a1, ci);
          default :  adder_cla #(N) adder3(co, sum, a0, a1, ci);
        endcase
endgenerate

endmodule
```

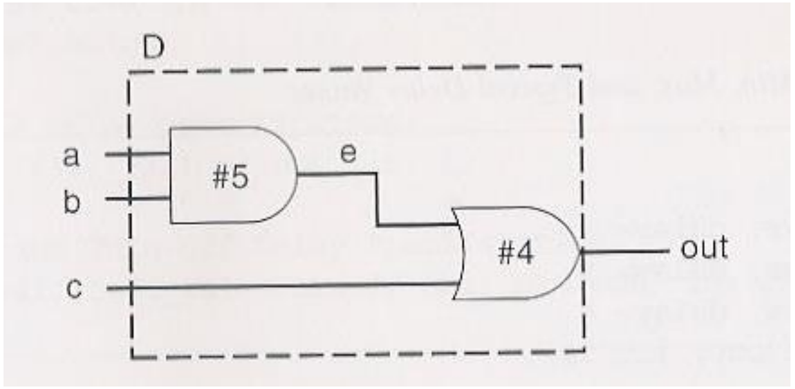
```
module top ( ….);
…
adder  #(1) add1 (co1, s1, a1, b1);
// adder_1bit add1 (co1, s1,, a1, b1);
…
adder  #(2) add2 (co2, s2, a2, b2);
// adder_2bit  add2 (co2, s2, a2, b2);
...
adder  #(8) add3 (co3, s3, a3, b3);
// adder_cla  add3 (co3, s3, a3, b3);
…
adder  #(32) add4 (co4, s4, a4, b4);
// adder_cla  #(32) add4 (co4, s4, a4, b4);
……
endmodule
```

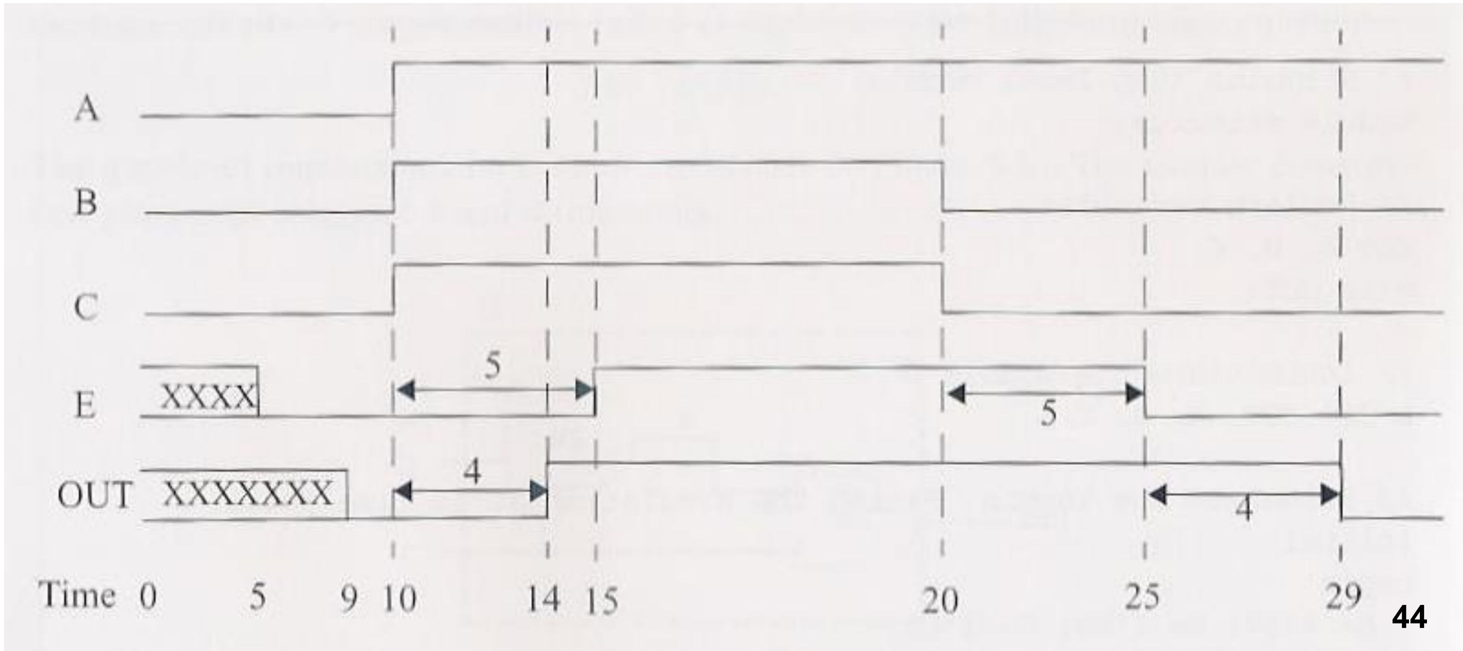
min/typ/max delays

```
// with Verilog-XL option +mindelays, delay=4
// with Verilog-XL option +typdelays, delay=5
// with Verilog-XL option +maxdelays, delay=6
and #(4:5:6) a1(out, i1, i2);

// if +mindelays, rise=3, fall=5, turn-off=min(3,5)
// if +typdelays, rise=4, fall=6, turn-off=min(4,6)
// if +maxdelays, rise=5, fall=7, turn-off=min(5,7)
and #(3:4:5, 5:6:7) a2(out, i1, i2);

// in fact, #() overwrite default parameters originally defined
// in a module during module instantiation
```

Delay Example

```
module D (out, a, b, c);
output out;
input a, b, c;
wire e;
    and #(5) a1(e, a, b);
    or #(4) o1(out, e, c);
endmodule
```





Timing Analysis Tools

- Static timing analsysi (STA)
  - Generate critical path delay without providing inputs
  - e.g., Synopsys Design
  - Might have false path delay which does not exist for any input combination
  - Can also generate power information, but not accurate for clock gating circuits
- Dynamic timing analysis
  - Generate the longest delay based on a sequence of user-specified inputs
  - e.g., Synopsys PrimeTime
  - Might not extract the crtical path delay if the provided input changes does not activate the critical path
  - Could also generate power information with clock gating

Dataflow Modeling
(continuous assignment: assign)

Continuous Assignment (assign)

- most basic statement in dataflow modeling
- Continuously drive a value onto a **net (wire)**
  - Compared with *procedural assignment* in a behavioral block (**always**)
- replace gates in the description of circuits at a higher level of abstraction (dataflow model)
  - Describe **combinational logic**

```
wire  [31:0] i1, i2, out;
assign out = i1 | i2;    // LHS is wire data type
// or (out, i1, i2)

//  operator | denotes bitwise OR operation
// out must be a net (wire)
```

Implicit Continuous Assignments

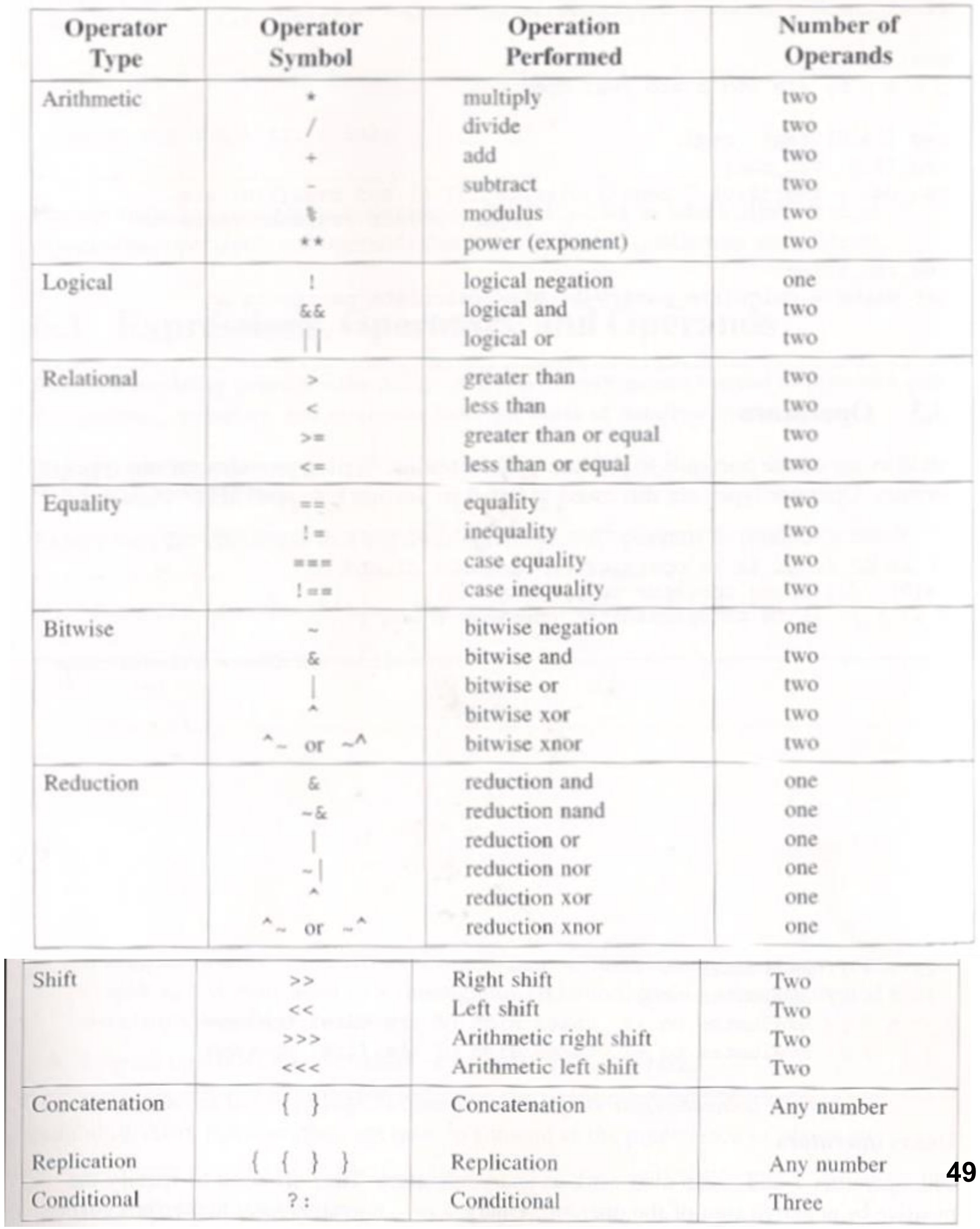
- // regular continuous assignment
  wire out;   // wire is default data type
  assign out **=** in1 **&** in2;\

- // implicit continuous assignment
  wire out **=** in1 **&** in2;   // no keyword **assign**

- // implicit net declaration
  wire in1, in2;
  assign #10 out = in1 **&** in2;
  // implicit net declaration of out as **wire**
  // wait for 10 time units (#10, delay control), then
  // sample values of in1, in2, calculate in1&in2,
  // and assign to out

Operator Types

- arithmetic (+, -)
- logical (!, &&, ||)
- relational (>, <)
- equality (==, ===, !=, !==)
- bitwise (~, &, |, ^)
- reduction (&, ~&)
- shift (<<, <<<, >>, >>>)
- concatenation ({ })
- conditional (? :)



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	? :	Conditional	Three

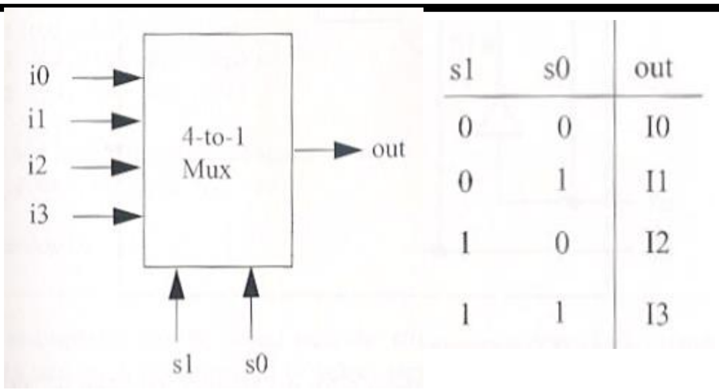
Logical Shift (<<, >>) versus Arithmetic Shift (<<<, >>>)

```
// Verilog-2001

integer  data_value,
         data_value_1995,
         data_value_2001;
…
data_value = -9;                          // stored as 1111_…_1111_0111
…
data_value_1995 = data_value  >> 3;   // stored as  0001_..._1111_1110
…
data_value_2001 = data_value  >>> 3; // stored as  1111_...._1111_1110
                                      // repeat sign bit during arithmetic right shift
…
data_value_1995 = data_value << 3;    // stored as 1111_..._1011_1000
…
data_value_2001 = data_value <<< 3;  // stored as 1111_..._1011_1000
                                      // repeat sign bit during arithmetic left shift
// give an example where the logical left shift << 3 is different from arithmetic
shift
// e.g., data_value2 = 32'b 1110_xxx ….
```

Example (4-to-1 Multiplexer)

```
// using logic equations
module mux4_logic (out, i0,
    i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3, s0, s1;

assign out =
(~s1 & ~s0 & i0) |
(~s1 & s0 & i1) |
(s1 & ~s0 & i2) |
(s1 & s0 & i3);

endmodule
```

```
// using conditional operator
module mux4_cond (out, i0,
    i1, i2, i3, s0, s1);

output out;
input i0, i1, i2, i3, s0, s1;

assign out = s1 ?
( s0 ? i3 : i2) : (s0 ? i1 : i0);

endmodule
```



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Example (4-bit adder)

```
// using dataflow
// with proper synthesis constraint

module add4 (sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;

// gate-level structure
// depends on  synthesis constraints
assign {c_out, sum} = a + b+ c_in;

endmodule
```

$p_i = a_i ? b_i, \quad g_i \quad a_i b_i, \quad c_{i+1} = g_i + p_i c_i$

$c_1 = g_0 + p_0 c_{in}$

$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$

$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_{in}$

$c_4 = g_2 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$

$s_i = p_i ? c_i$

```
// explicit carry lookahead adder using p and g

module add4 (sum, c_out, a, b, c_in);
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;

assign p0=a[0]^b[0], p1=a[1]^b[1], p2=a[2]^b[2],
      p3=a[3]^b[3];
assign g0=a[0]&b[0], g1=a[1]&b[1], g2=a[2]&b[2],
      g3=a[3]&b[3];
assign c1 = g0|(p0&c_in),
     c2=g1|(p1&g0)|(p1&p0&c_in),
     c3=g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&c_in),
     c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)
                  |(p3&p2&p1&p0&c_in);
assign sum[0] = p0 ^ c_in, sum[1] = p1 ^ c1,
    sum[2]=p2 ^ c2, sum[3] = p3 ^ c3;
assign c_out = c4;

endmodule
```

Behavioral Modeling

Procedural Assignments

- two structured procedures
  - **always** (SystemVerilog uses **always_ff, always_comb, always_latch**)
  - **Initial** (not synthesizable)
- procedural assignments update LHS values of
  - **reg** (SystemVerilog uses **logic** to replace **reg** for avoiding confusion)
  - **integer**
  - **real** (non-synthesizable)
  - **time** (non-synthesizable)
- The value placed on a variable remain unchanged until another procedural assignment updates the variable with a different value
  - unlike continuous assignments (using **assign**) in dataflow where one assignment can cause the value of RHS expression to be continuously placed on the LHS expression

always statement

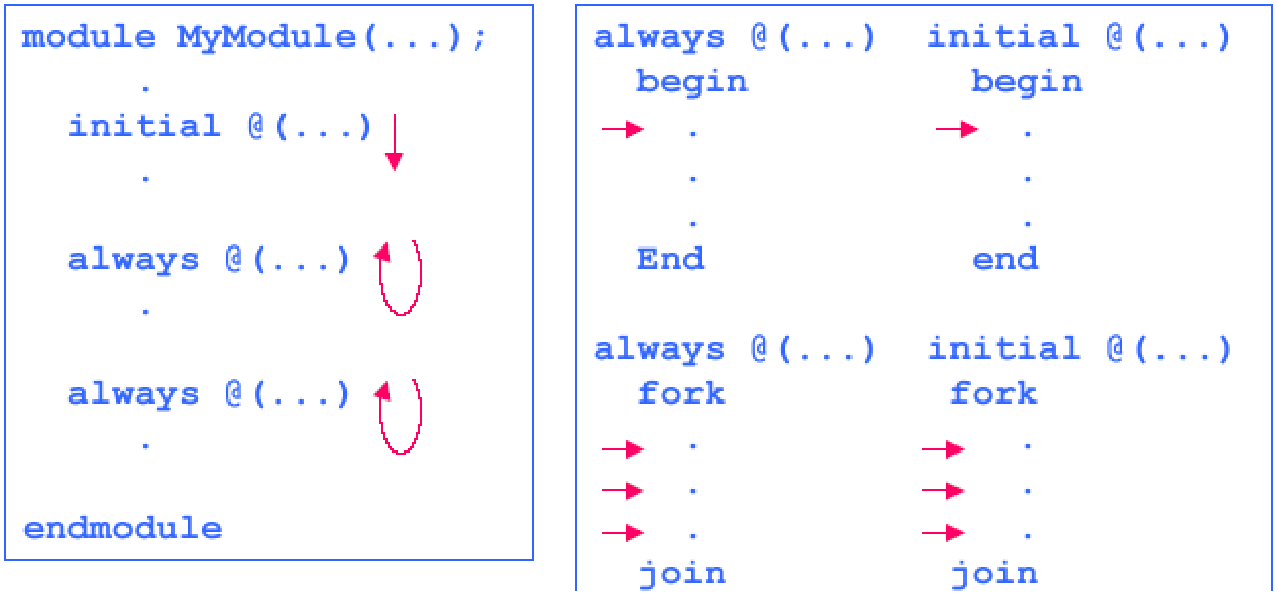
- starts at time 0 and executes the statements in the always block repeated*ly in a looping fashion*
- used to model a block of activity that is repeated continuously in a digital circuit
- usually include timing/event control (e.g., @posedge) or sensitivity list (e.g., @(a, b, c, …))

```
module clock_gen (output reg clock);
  initial clock = 1'b0;            // executed only once
  always #10 clock = ~clock;   // executed continuously
  initial #1000 $finish;           // finish execution at time step = 1000
endmodule
```

behavioral modeling

- statements inside **begin** … **end** are executed *sequentially*
- statements inside **fork** … **join** are executed *in parallel*
  - **fork** … **join** block is not synthesizable

- Event-driven procedures: **always, initial**
- Sequential blocks: **begin...end**
- Parallel blocks: **fork...join**



```
module MyModule(...);
        .
   initial @(...)
        .

   always @(...)
        .

   always @(...)
        .

endmodule
```

```
always @(...)   initial @(...)
  begin             begin
     .                 .
     .                 .
     .                 .
  End               end

always @(...)   initial @(...)
  fork             fork
     .                 .
     .                 .
     .                 .
  join             join
```

behavioral statements (used inside initial or always blocks)

- procedural assignments
  - blocking (**=**) vs. non-blocking (**<=**)
  - operators (such as +, -, *, &, |, ^, ?:, …)
  - timing control **#**, event control **@**
- conditional statements (**if … else**…)
- multi-way branching (**case … endcase**)
- looping statement
  - **while, repeat, for, forever**
- sequential and parallel blocks
  - sequential block (**begin … end**)
  - parallel block (**fork … join**

Sequential logic

- flip-flop using non-blocking assignment <=

```
module FF (q, d, clk);
output q;
input d, clk;
reg q;  // LHS of procedural statement should be data type reg

always @(posedge clk)
   q <= d;  // non-blocking assignment <=

endmodule
```

- flip-flop with active-low asynchronous reset

```
Module FF_asyn_rst (q, d, clk, reset);
output q;
input d, clk, reset;
reg q;

always @(posedge clk or negedge reset)  // use negedge for active-low control signal
   if (~reset )    q <= 1'b0;
   else            q <= d;

endmodule
```

Combinational logic

- Use blocking assignment **=**

```
module adder32 (a, b, sum);
output [31:0] sum;
input [31:0] a, b ;
reg sum;

always @( a or b)
   sum = a + b;  // blocking assignment  =

endmodule
```

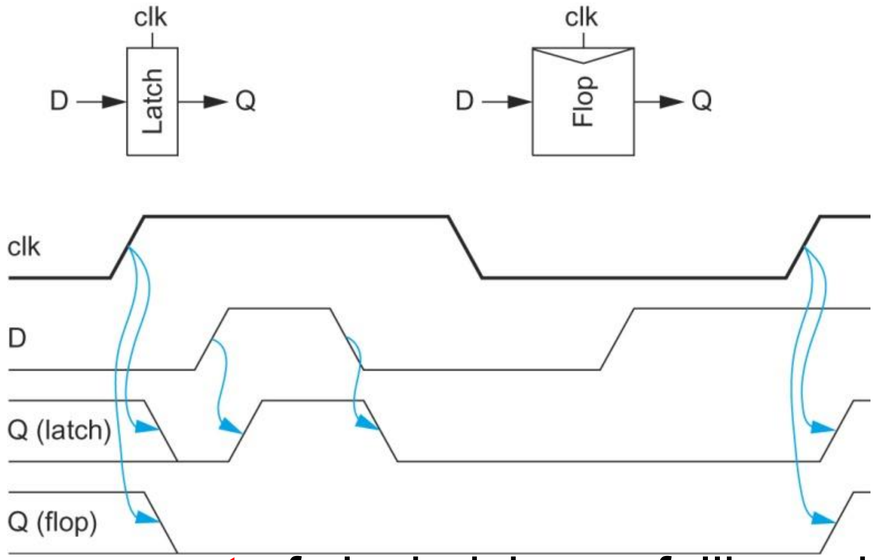
- could use similar continuous assignment assign

```
module adder32 (a, b, sum);
output [31:0] sum;
input [31:0] a, b ;
wire sum;

assign sum = a + b;  // continuous assignment assign

endmodule
```

Sequencing Elements (Latch and Flip-Flop)

- **Latch**: *Level sensitive*
  - pass input to output at **time interval** when level of control is logic 1
  - a.k.a. transparent latch

```
// Verilog code for latch
always @ (CLK or D)
   if (CLK)  Q <= D;
```



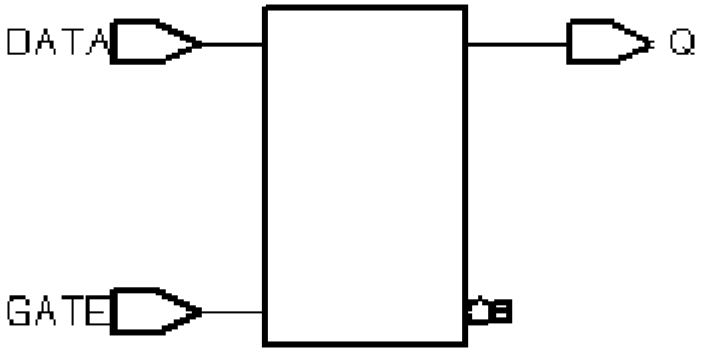
- **Flip-Flop**: *edge triggered*
  - pass input to output at the **moment** of clock rising or falling edge
  - a.k.a. opaque edge-triggered flip-flop

```
// Verilog code for DFF
always @ (posedge CLK)
   Q <= D;
```

inferring a D-latch

- Model a latch using not fully specified **if ..**
- Model a multiplexer using fully specified **if … else …**

```
module d_latch (GATE, DATA, Q);
   input GATE, DATA;
   output Q;
   reg Q;

always @(GATE or DATA)
   if (GATE)
      Q = DATA;

endmodule
```

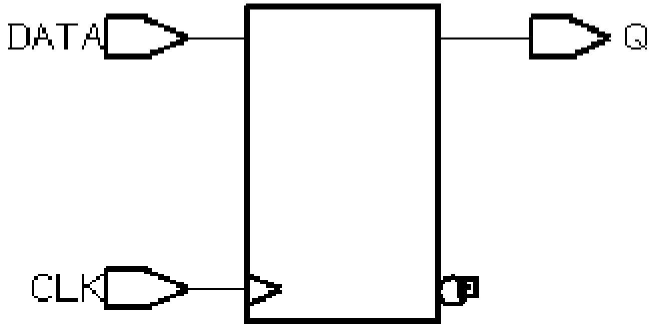


```
module d_CL (
input gate, data,
output reg q );

// infer a combinational logic,
// not a latch
always @ (gate or data)
   if (gate)
       q = data;
   else
      q = ‘b0;

endmodule
```

inferring a D-flip-flop

- Model a flip-flop (FF) using non-blocking assignment (**<=**) with *edge-triggered* event (**@posedge** or **@negedge**)
  - c.p. latch modeling using l*evel-sensitiv*e event

```
module dff_pos (DATA, CLK, Q);
   input DATA, CLK;
   output Q;
   reg Q;

always @(posedge CLK)
   Q <= DATA;
endmodule
```



Shift registers using non-blocking statements

- each flip-flop shift to its neighbor, except for the two boundary FFs that acts as input and output registers
- line buffer (in CNN hardware) is realized by shift registers
- quiz: how to model a long shift registers ?
  - { sr[1:n-1], out } <= {d, sr[1:n-1] }; // n shift registers: { sr[1:n-1], out }

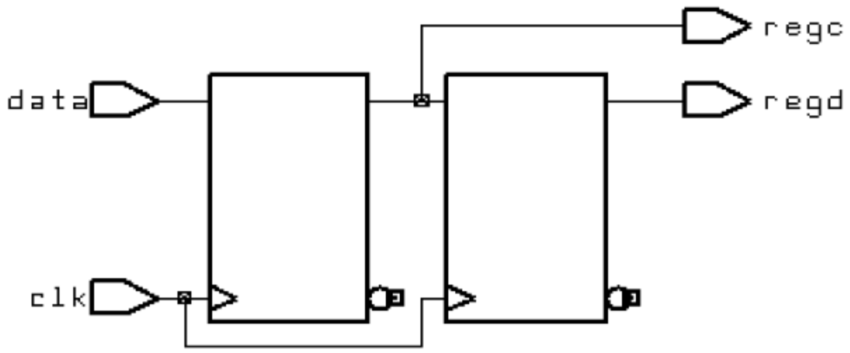
```
// shift register, OK
always @(posedge clock)
begin
   reg1 <= d;
   reg2 <= reg1;
   reg3 <= reg2;
   out   <= reg3;
end
```

```
// 4 FFs with same input d
// problematic ?
always @(posedge clock)
begin
   reg1 = d;
   reg2 = reg1;
   reg3 = reg2;
   out   = reg3;
end
```

synthesis of non-blocking and blocking (shift registers vs. multi-bit register with same input)

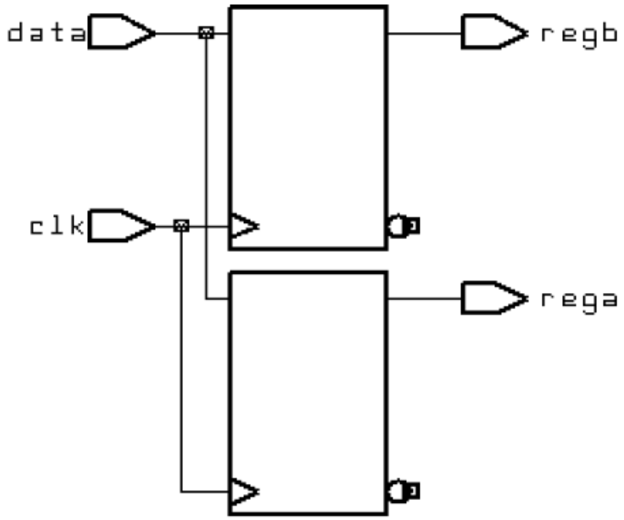
- non-blocking assignment (**<=**)

```
module rtl (clk, data, regc, regd);
input data, clk;
output regc, regd;

reg regc, regd;

always @(posedge clk)
begin
     regc <= data;
     regd <= regc;
end
endmodule
```



- blocking assignment (**=**)

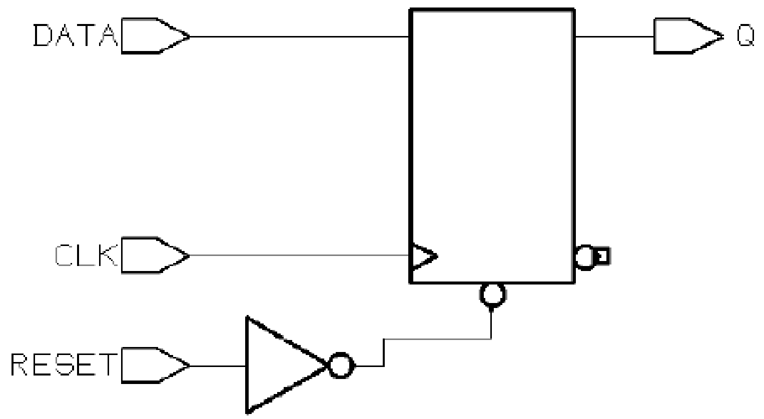
```
module rtl (clk, data, rega, regb);
input data, clk;
output rega, regb;

reg rega, regb;

always @(posedge clk)
begin
     rega = data;
     regb = rega;
end
endmodule
```



FF with Asynchronous reset (active-high)

- Reset FF content whenever reset=high (active high), independent of clock signal (asynchronous)
  - Event of edge-triggered clk and reset signals
- remember to Initialize register content before execution using resettable DFFs
  - **initial** is not synthesizable, only used in simulation

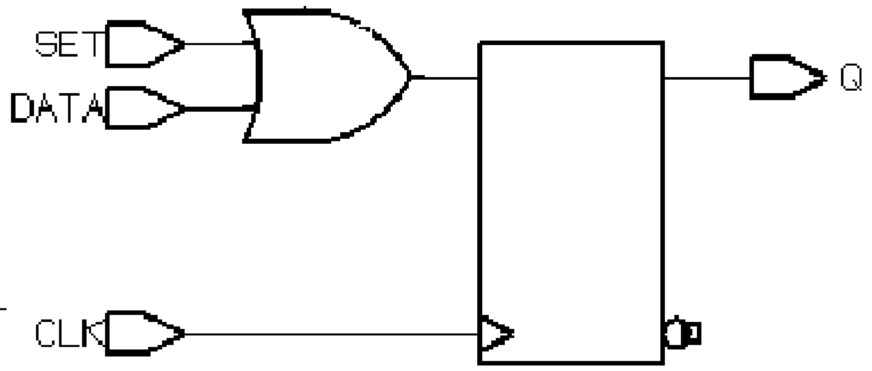
```
module dff_async_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

always @(posedge CLK or posedge RESET)
  if (RESET)
    Q <= 1'b0;
  else
    Q <= DATA;
endmodule
```



DFF with Synchronous set (active-high)

- sensitive list includes only clock signal
  - synthesized into normal DFF with extra combinational logic for the input data

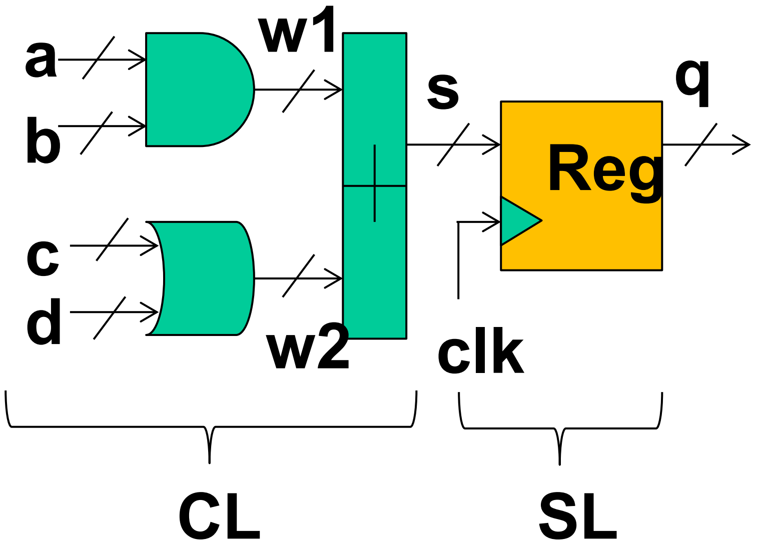
```
module dff_sync_set (DATA, CLK, SET, Q);
   input DATA, CLK, SET;
   output Q;
   reg Q;

//synopsys sync_set_reset "SET"
always @(posedge CLK)
   if (SET)
      Q <= 1'b1;
   else
      Q <= DATA;
endmodule
```



Separation of Combinational and Sequential Logic

- use blocking assignment (=) for combinational logic
- use non-blocking assignment (<=) for sequential logic
- do not mix blocking and non-blocking assignments in a single always block

```
// model CL using =
always @ (a or b or c or d) begin
   w1 = a & b;
   w2 = c | d;
   s = w1 + w2;
end

// model SL using <=
always @(posedge clk) begin
   q <= s;
end
```



```
// merged CL and SL
always @(posedge clk) begin
   q <= (a & b) + (c | d);
end
```

Pipelined A*B+C

```
module pipelined (
input [31:0] A, B;
input [63:0] C;
input clk;
output [63:0]  final_out);

reg [63:0] pipe1_C, pipe1_AB, pipe2_out;

always @ (posedge clk) begin
   pipe1_AB <= A*B;
   pipe1_C <= C;
end

always @(posedge clk)
   pipe2_out <= pipe1_AB + pipe1_C;

assign final_out = pipe2_out;

endmodule
```

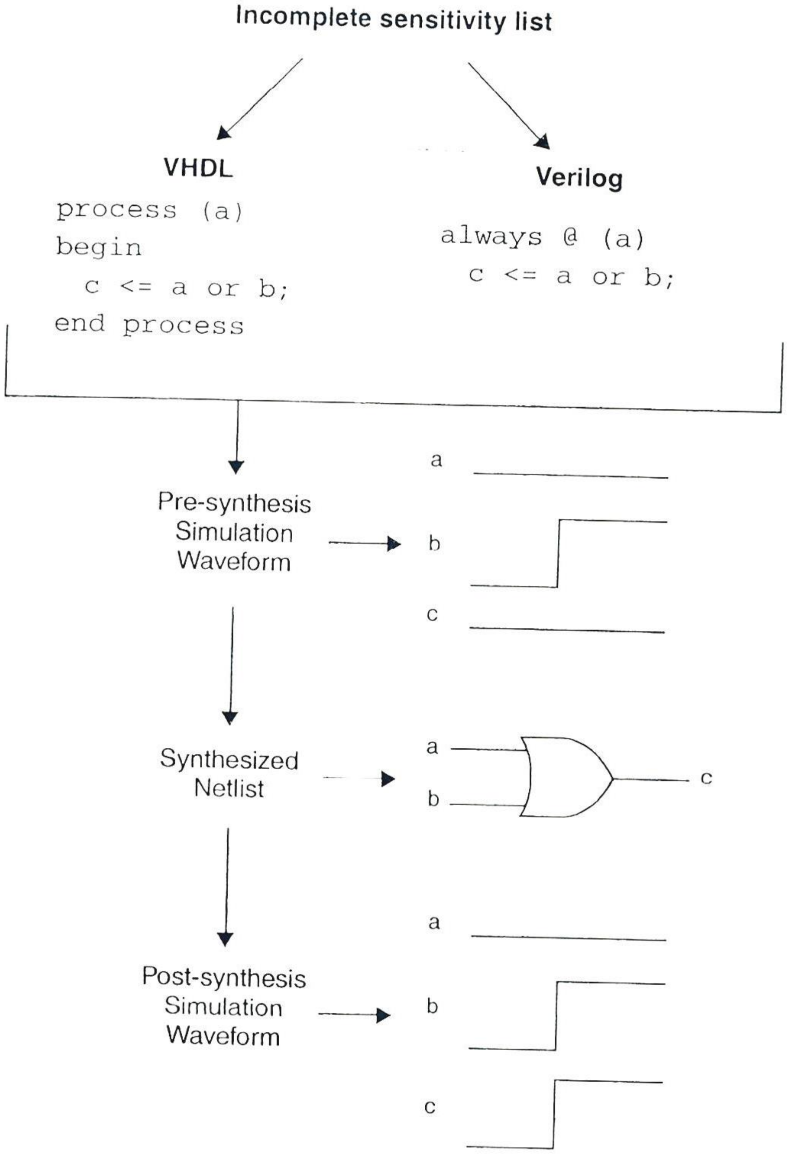
Use of @*, or @(*) for CL

```
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c: d+e;      // out1, out2 should be reg
out2 = f ? g+h: p+m;
end
```

```
// all input variables are included automatically
always @(*)   // IEEE 1364-2001
begin
out1 = a ? b+c: d+e;
out2 = f ? g+h: p+m;
end
```

simulation mismatch due to incomplete sensitivity list

- make sure that sensitivity lists contain only necessary signals
  - e.g. always @(*) … for pure CL
  - missing signals in sensitivity list cause mis-match in RTL (pre-synthesis) simulation and gate-level (post-synthesis) simulation
- adding unnecessary sensitivity list slows down simulation

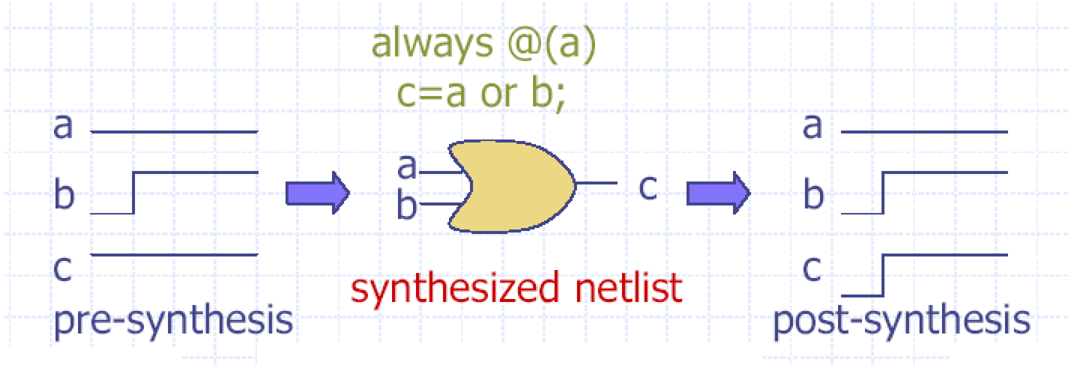

Incomplete sensitivity list
VHDL
Verilog
Pre-synthesis Simulation Waveform
a
b
c
Synthesized Netlist
a
b
c
Post-synthesis Simulation Waveform
a
b
c

VHDL:

```
process (a)
begin
  c <= a or b;
end process
```

Verilog:

```
always @ (a)
  c <= a or b;
```



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

conditional statements (if … else …)

```
if (!lock) buffer = data;
if (enable) out = in;  // infer a latch
if (enable) out = in  else out = d;  // infer a MUX2
```

```
if (number_queued < MAX_Q_DEPTH)
        begin    data_queue = data;
                 number_queued = number_queued + 1;
        end
else
        $display ("Queue Full. Try Again");
```

```
if (alu_control == 0)        y = x + z;
else if (alu_control == 1)   y = x – z;
else if (alu_control == 2)   y = x * z;
else $display ("Invalid ALU control sign");
```

Unspecified Condition: Inferring Latch

- Fully specified condition
  - Describe combinational logic

```
always @ (a, b, c) begin
   if (c == 1)
      d=a+b;
   else
      d=0;
end
```

```
always @ (a, b, c) begin
   d=0;
   if (c == 1)
      d=a+b;
end
```

- Unspecified condition
  - Infering latch to store the the previous value

```
always @ (a, b, c) begin
   if (c == 1)
      d=a+b;   // d is stored in a latch controlled by c
end
```

Multiway Branching (case Statements)

```
reg [1:0] alu_control;
…
case (alu_control)
      2'd0     : y = x + z;
      2'd1     : y = x – z;
      2'd2     : y = x * z;
      default : $display("Invalid ALU control signal");
endcase
```

```
if (alu_control == 0)          y = x + z;
else if (alu_control == 1)   y = x – z;
else if (alu_control == 2)   y = x * z;
else $display ("Invalid ALU control sign");
```

4-to-1 MUX with case

```
// 4-to-1 multiplexer (i1, i2, i3, s[1:0], out)

module mux4 (out, i0, i1, i2, i3, s0, s1);
output out;
input i0, i1, i2, i3, s0, s1;
reg out;

always @(s1, s0, i0, i1, i2, i3)
case ({s1, s0})
    2'd0: out = i0;
    2'd1: out = i1;
    2'd2: out = i2;
    2'd3: out = i3;
    default: $display ("Invalid control signals");
endcase

endmodule
```

full case and parallel case

- full case
  - all possible branches are specified
  - otherwise, latches are synthesized
- parallel case
  - no cases overlap (one and only one of the branches is executed each time)
  - hardware multiplexers are synthesized in parallel case
  - if not determined, hardware *priority encoder* is synthesized (such as 1xxx, x1xx, xx1x, xxx1)

both full case and parallel case

- a hardware multiplexer is synthesized

```
// full case, and parallel case
// infer multiplexer

input [1:0] a;
always @ (a, w, x, y, z) begin
   case (a)
      2’b11: b=w;
      2’b10: b=x;
      2’b01: b=y;
      2’b00: b=z;
   endcase
end
```

casex, casez

- **casez**
  - treat all **z** values as don't care, **z** can also be represented by **?**
- **casex**
  - treat all **z** and **x** values as don't care

```
reg [3:0] encoding;
integer state;

// not parallel case, but full case,
// infer priority encoder
casex (encoding)
     4'b1xxx : next_state = 3;  // this case has highest priority
     4'bx1xx : next_state = 2;
     4'bxx1x : next_state = 1;
     4'bxxx1 : next_state = 0;
     default : next_state = 0;
endcase
```

4-to-1 multiplexer (4:1 MUX)

- input
  - in0, in1, in2, in3, s[1:0]
- output
  - out

```
reg out;

always @(s, in0, in1, in2, in3)
case (s)
    2’b00: out = in0;
    2’b01: out = in1;
    2’b10: out = in2;
    2’d11: out = in3;
    default: $display (“Invalid control signals”);
endcase
```

1-to-4 demultiplexer (1:4 DEMUX)

- input
  - in, s[1:0]
- output
  - out0, out1, out2, out3
  - all outputs except the selected one are z

```
reg  out0, out1, out2, out3;

always @(s, in0, in1, in2, in3)
case (s)
    2’b00: begin out0 = in;      out1=1’bz;  out2=1’bz;  out3=1’bz;   end
    2’b01: begin out0 = 1’bz;  out1=in;      out2=1’bz;  out3=1’bz;   end
    2’b10: begin out0 = 1’bz;  out1=1’bz;   out2=in;     out3=1’bz;   end
    2’d11: begin out0 = 1’bz;  out1=1’bz;   out2=1’bz;  out3=in;      end
    default: $display (“Invalid control signals”);
endcase
```

4:2 priority encoder using casex

- input
  - in0, in1, in2, in3
- output
  - out[1:0]

```
reg [1:0] out;

always @(in3, in2, in1, in0)
casex ({in3, in2, in1, in0})
    4'b1xxx: out = 2'b11;
    4'b01xx: out = 2'b10;
    4'b001x: out = 2'b01;
    default  : out = 2'b00;
endcase
```

4:2 encoder

- input
  - in0, in1, in2, in3
- output
  - out[1:0]

```
reg [1:0] out;
reg       valid;

always @(in3, in2, in1, in0)
valid = 1;
case ({i3, i2, i1, i0})
    4'b1000: out = 2'b11;
    4'b0100: out = 2'b10;
    4'b0010: out = 2'b01;
    4'b0001: out = 2'b00;
    default  : begin out = 2'bxx;  valid = 0; end
endcase
```

2:4 decoder

- input
  - in[1:0]
- output
  - out0, out1, out2, out3

```
wire  [1:0] in;
reg  out0, out1, out2, out3;

always @(in)
case (in)
    2’b00: begin out0=1;   out1=0;    out2=0;   out3=0;   end
    2’b01: begin out0=0;   out1=1;    out2=0;   out3=0;   end
    2’b10: begin out0=0;   out1=0;    out2=1;   out3=0;   end
    2’d11: begin out0=0;   out1=0;    out2=0;   out3=1;   end
endcase
```

while Loop

```
integer count;

initial
begin
count = 0;
while (count < 128)
     count = count +1;
end
```

```
`define TRUE 1’b1
`define FALSE 1’b0
reg [15:0] flag;
integer i;
reg continue;

// find the leading 1 in flag
initial
begin
flag = 16’b 0010_0000_0000_0000
i = 0;
continue = `TRUE;
while ((i<16) && continue)
   begin
      if (flag[i]) continue = `FALSE;
      i=i+1;
   end
end
```

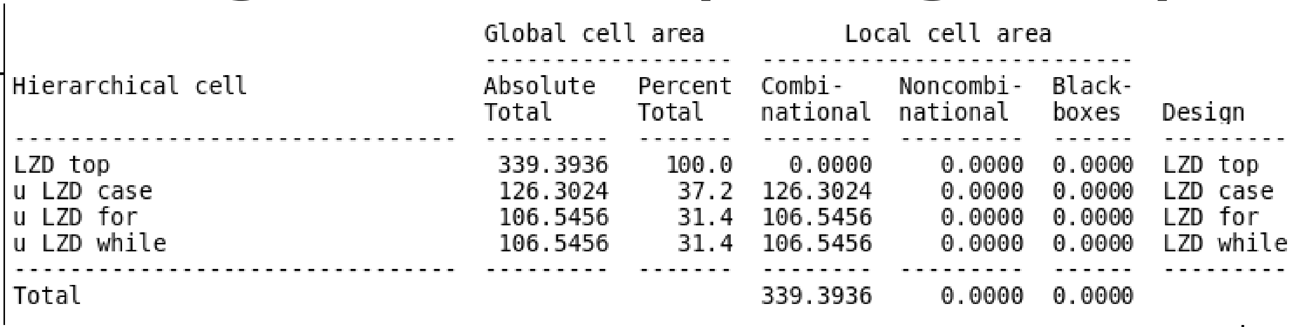
for Loop

- **for** loops are generally used when there is a ***fixed*** beginning and end to the loop
- If the loop is simply looping on a certain condition, it is better to use the **while** loop

```
integer count;

initial
for (count=0; count<128;  count=count+1)
   $display("Count=%d", count);
```

```
`define MAX_STATES 32
integer state[0:`MAX_STATES-1];
integer i;

initial
begin
for (i=0; i<32; i=i+2)
   state[i]=0;      //initialize even locations
for (i=1; i<32; i=i+2)
   state[i]=1;      //initialize odd locations
end
```

16-bit LOD using **for** loop (~38 gates)



```
                                    Global cell area        Local cell area
                                    ------------------  -------------------------
Hierarchical cell                   Absolute   Percent  Combi-    Noncombi-  Black-
                                    Total      Total    national  national   boxes   Design
--------------------------------    ---------  -------  --------  ---------  ------  ---------
LZD top                              339.3936    100.0    0.0000     0.0000  0.0000  LZD top
u LZD case                           126.3024     37.2  126.3024     0.0000  0.0000  LZD case
u LZD for                            106.5456     31.4  106.5456     0.0000  0.0000  LZD for
u LZD while                          106.5456     31.4  106.5456     0.0000  0.0000  LZD while
--------------------------------    ---------  -------  --------  ---------  ------  ---------
Total                                                   339.3936     0.0000  0.0000
```

```
`define TRUE 1'b1
`define FALSE 1'b0

module LZD_for (
input [15:0]din, output reg [3:0]zero_cnt); input [15:0] din, output reg [3:0] zero_cnt);
reg continue;
integer i;

// zero_cnt is the number of leading zeros
// continue is false, if leading one is found
always@(*) begin
   continue = `TRUE;
   for (i=15; continue && (i>=0); i=i-1) begin
      zero_cnt = din[i] ? 4'd15 - i : 4'd0;
      continue = din[i] ? `FALSE  : `TRUE;
   end
end

endmodule
```

Delay=0.33ns

Area=106 um x um (um²)
~=(106/2.82) NAND2 gates
~=38 gate equivalent

Quiz

- what is value of cnt if din=0000_0000_0000_0000?

```
 continue = `TRUE;
for (i=15; continue && (i>=0); i=i-1) begin
     cnt = din[i] ? 4'd15 - i : 4'd0;
     continue = din[i] ? `FALSE  : `TRUE;
end
```

- How to change the above code to detection # of leading bits before bit-inverse ?
  - if din=0001_xxxx_xxxx_xxxx, cnt=3 (din[13[=0, din[12]=1)
  - if din=1110_xxxx_xxxx_xxxx, cnt=4 (din[13[=1, din[12]=0)
  - if din=0000_0000_0000_0000, din=16
  - if din=0000_0000_0000_0000, din=16

example: ripple carry adder (RCA) using for loop in behavioral modeling

```
always @ (*) begin
   c[0] = cin;
   for (i=0; i<=31; i++)
      {c[i+1], s[i]} = a[i] + b[i] + c[i];
   cout = c[32];
end
// the for loop will be unrolled as follows
// {c[1],s[0]}=a[0]+b[0]+c[0];
// {c[2],s[1]}=a[1]+b[1]+c[1];
…
// {c[32],s[31]}=a[31]+b[31]+c[31];
```

shift registers (many registers)

```
x[0] <= d
x[1] <= x[0]
x[2] <= x[1]
…
x[N-2] <= x[N-3]
out <= x[N-2]
```

```
// a total of N shift registers, each bw bits
reg [bw-1:0] x[0:N-2];
reg [bw-1:0] d, out;
always @ (posedge clk) begin
  x[0] <= d;
  for (i=1; i<=N-2; i=i+1)
    x[i] <= x[i-1];
  out <= x[N-2];
end
```

```
// a total of N shift registers
reg [bw-1:0] x[0:N-2];
reg [bw-1:0] d, out;
always @ (posedge clk) begin
  {x[0:N-2], out} <= {d, x[0:N-2]};
end
```

Dopt Product 4 (DP4)

- 4-D dot-product: a[0]*b[0]+ a[1]*b[1]+ a[2]*b[2]+ a[3]*b[3]

```
module DP4 (input[16*4-1:0]in_a, in_b, output [31:0]out);
wire [15:0] a[3:0],  b[3:0];

// convert two long bit-vectors in_a, in_b into two arrays a, b
assign {a[3],a[2],a[1],a[0]}=in_a;
assign {b[3],b[2],b[1],b[0]}=in_b;

integer i;
reg [31:0] tmp[0:3];
always @ (*) begin
   tmp[0] = a[0]*b[0];
   for (i=1; i<=3; i=i+1) tmp[i] = a[i]*b[i] + tmp[i-1];
end

assign out = tmp[3];

endmodule
```

DPn

- n-D dot-product

```
module #parameter n=4 DPn (input[16*n-1:0]in_a, in_b, output [31:0]out);
reg [15:0] a[n-1:0];
reg [15:0] b[n-1:0];
integer i;
always@ (*) begin   // convert two long bit-vectors into two arrays
            for (i=0; i<n; i=i+1) begin
                        a[i]= in_a[16*i +:16];
                        b[i]= in_b[16*i +:16];
            end
end

reg [31:0] tmp[0:n];
always @ (*) begin
  tmp[0] = 0;
  for (i=0; i<=n-1; i=i+1) tmp[i+1] = a[i]*b[i] + tmp[i];
end
assign out = tmp[n];
endmodule
```

Example: Saturation Adder

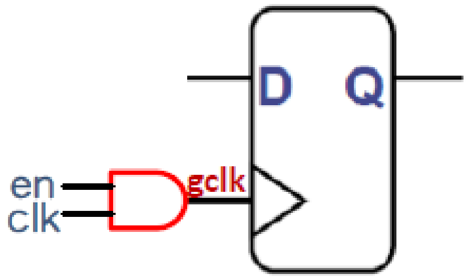
- clip the computation results to a fixed range

```
signed  [7:0] a, b, sum, tmp, maxk min;
// wire [8:0] carry;  // if carry out bits are known

assign max = 8’d127;
assign min = 8’d-128;
assign tmp = a + b;
always @ (*) begin
   if ( a[7] == b[7] && tmp[7] != a[7] )     // carry[ [8] != carry]7]
      sum = a[7] ? min : max; // sum = { a[7], 7{~a[7]} };
   else
      sum = tmp;
end
```

Clock Gating

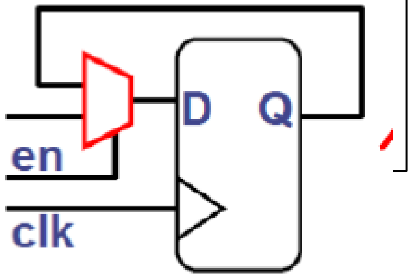
clock gating

```
module dff(Q, D, clk);
input D, clk;
output Q;
reg Q;
wire gclk, en;
// clock signal is from the output of AND
// glitch might cause extra clock edges
assign gclk = clk & en;
always @(posedge gclk)
   Q <= D;
endmodule
```



gclk might have glitches !!!
cause unexpected latching

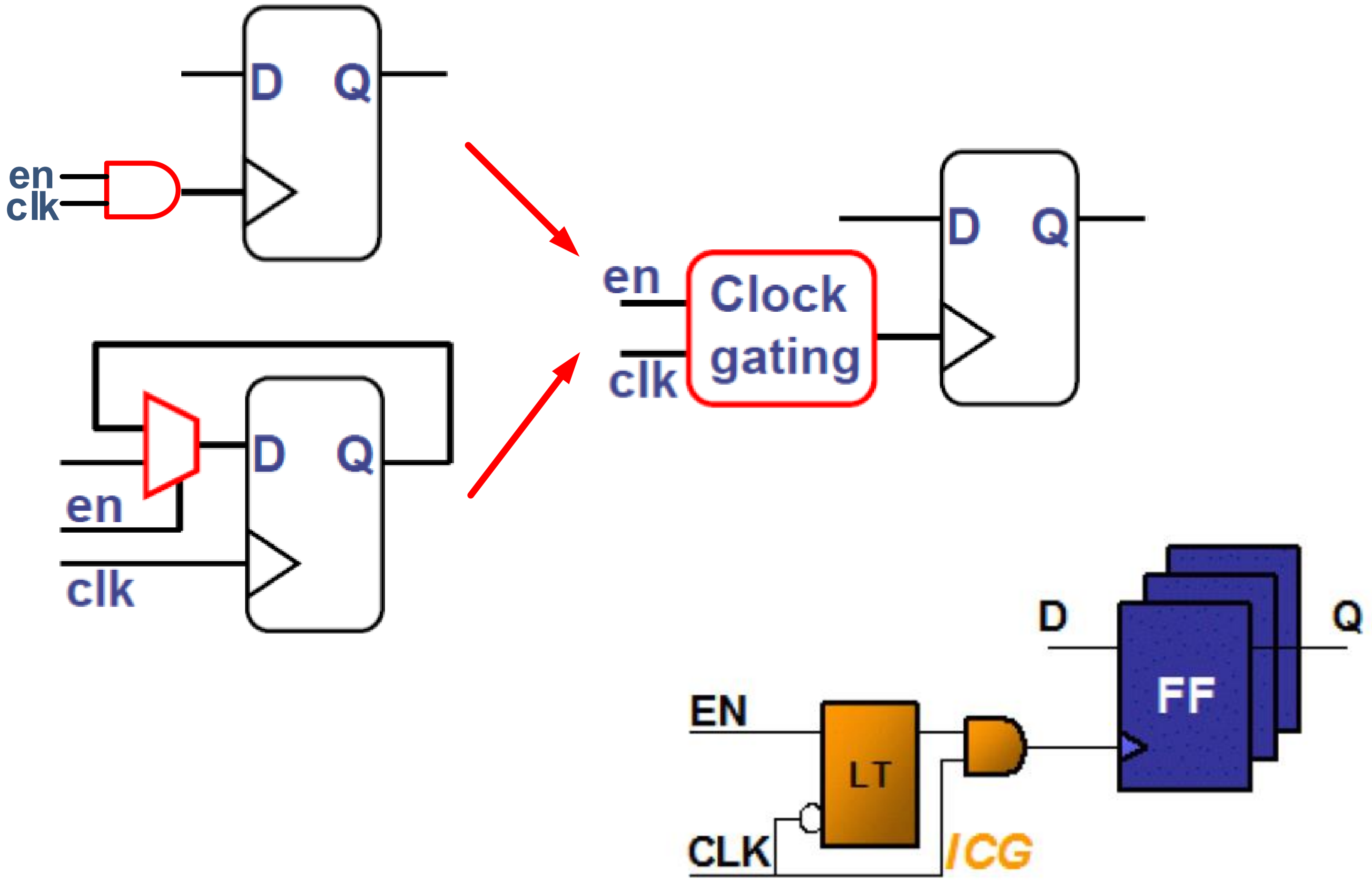
```
module dff(Q, D, clk);
input D, clk;
output Q;
reg Q;
wire en;

// data input from MUX
always @(posedge clk)
if (en) begin
Q <= D;
end
endmodule
```



The clk port might still have switching power!
not efficiently reduce dynamic power

Synthesized Clock Gating

- When synthesis is done with proper commands
  - Both designs lead to safe clock gating circuit (with latch)



Tasks and Functions

Tasks vs. Functions

- task is similar to subroutine in FORTRAN
- function is similar to function in FORTRAN

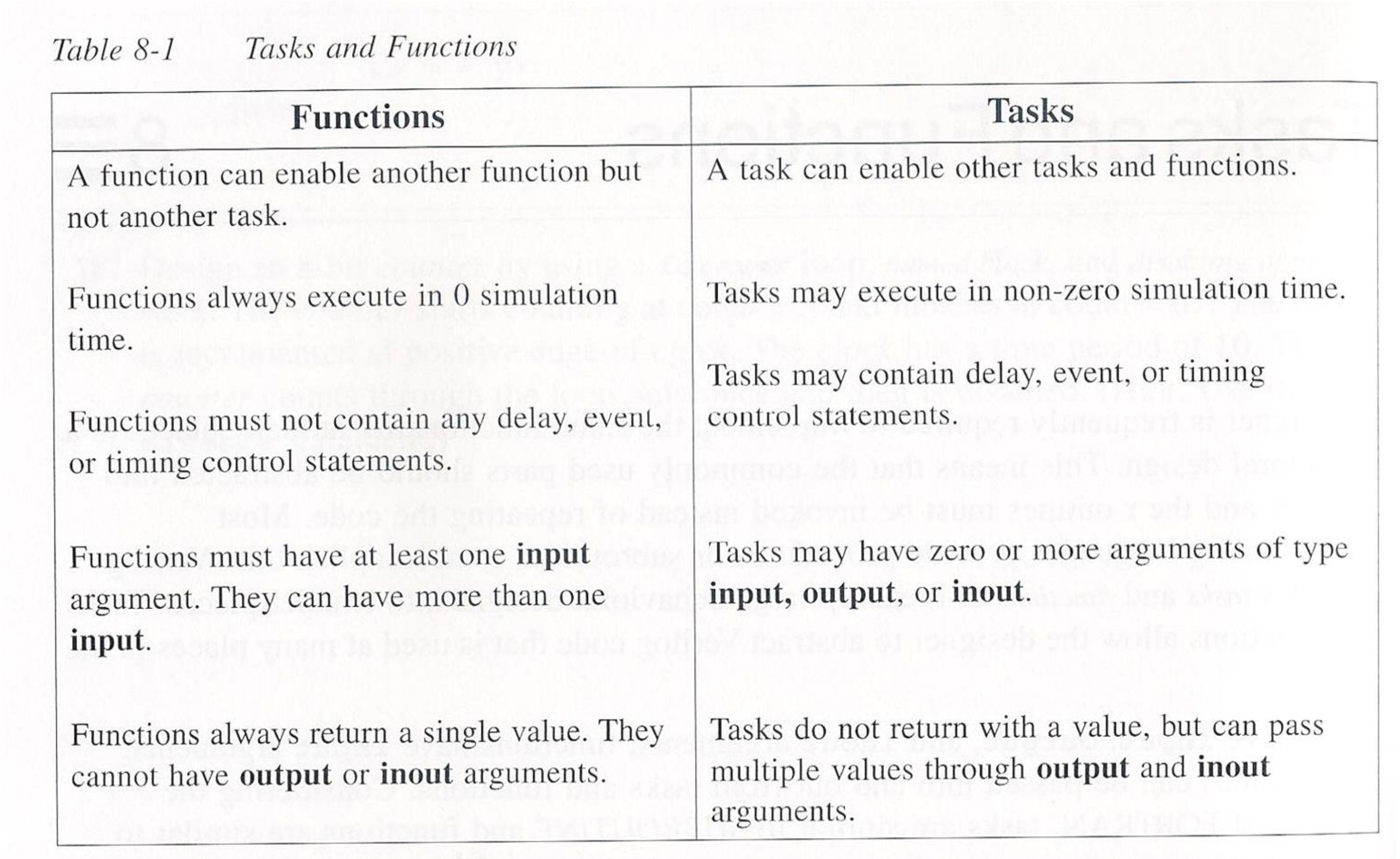


Table 8-1 Tasks and Functions

Functions	Tasks
A function can enable another function but not another task.	A task can enable other tasks and functions.
Functions always execute in 0 simulation time.	Tasks may execute in non-zero simulation time.
Functions must not contain any delay, event, or timing control statements.	Tasks may contain delay, event, or timing control statements.
Functions must have at least one **input** argument. They can have more than one **input**.	Tasks may have zero or more arguments of type **input, output**, or **inout**.
Functions always return a single value. They cannot have **output** or **inout** arguments.	Tasks do not return with a value, but can pass multiple values through **output** and **inout** arguments.

Tasks vs. Functions

- defined in a module and local to the module
  - called from the **always** or **initial** blocks, or other tasks and functions
- tasks are used for commonly Verilog code
  - contain delays, timing, event constructs, or multiple output arguments
  - ***can have input, output and inout arguments***
- functions are used for code that
  - is purely **combinational**
  - executes in zero simulation time
  - provides ***exactly one output***
  - can have input arguments

Tasks (task … endtask)

```
module operation;
…
parameter delay=10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;
always @ (A or B)
begin
bitwise_oper(AB_AND, AB_OR, AB_XOR,
     A, B);
end
…
task bitwise_oper;
output [15:0] ab_and, ab_or, ab_xor;
input [15:0] a, b;
begin
   #delay ab_and = a & b;
   ab_or = a | b;
   ab_xor = a ^ b;
end
end task
…
endmodule
```

task could have
(1) multiple outputs
(2) delay/timing constructions

```
// another task declaration
//
task bitwise_oper (
output [15:0] ab_and, ab_or, ab_xor,
input [15:0] a, b);
begin
   #delay ab_and = a & b;
   ab_or = a | b;
   ab_xor = a ^ b;
end
endtask
```

Example: Asymmetric Sequence Generator

```
module sequence;
reg clk;

initial  init_sequence;                // invoke task init_sequence
always asymmetric_sequence; // invoke another task

task init_sequence; begin  clk= 1'b0; end endtask

task asymmetric_sequence;
begin
   #12 clk = 1'b0;
   #5 clk = 1'b1;
   #3 clk = 1'b0;
   #10 clk = 1'b1;
end endtask

endmodule
```

Automatic (Re-entrant) Tasks

```
// all normally declared task items are statically allocated and shared.
// automatic task allows for dynamic allocation for each invocation
// where each task call operates in an independent memory space.

module top;
reg [15:0] cd_xor, ef_xor;
reg [15:0] c, d, e, f;

// this task can invoked concurrently with individual memory space
task automatic bitwise_xor
output [15:0] ab_xor;
input [15:0] a, b;
begin
   ab_xor = a ^ b;
end endtask

// two tasks are called concurrently in two procedural blocks
always @(posedge clk)   bitwise_xor (ef_xor, e, f);
always @(posedge clk2) bitwise_xor (cd_xor, c, d);  // clk2 has twice frequency
endmodule
```

Function (function … endfunction)

```
module parity;
reg [31:0] addr;
reg parity;
always @ (addr)
   parity = calc_parity(addr);

function calc_parity;
input [31:0] address;
begin
   calc_parity = ^address;
end
endfunction


endmodule
```

function could have
(1) only one returned output
(2) no delay/timing constructions
(3) purely combinational

```
// alternate function definition
function calc_parity
(input [31:0] address);
begin
   calc_parity = ^address;
end
endfunction
```

Example: Left/Right Shifter

```
module shifter;
‘define LEFT_SHIFT 1’b0;
‘define RIGHT_SHIFT 1’b1;
reg [31:0] addr, left_addr, right_addr;
reg control;
always @ (addr)
begin
   left_addr = shift (addr, ‘LEFT_SHIFT);
   right_addr = shift (addr, ‘RIGHT_SHIFT);
end

function [31:0] shift; // function output is a 32-bit value
input [31:0] address;
input control;
begin
   shift = (control == ‘LEFT_SHIFT) ? (address << 1) : (address >>1);
end
endfunction

endmodule
```

Automatic (Recursive) Functions

```
// functions are normally used non-recursively
// automatic function allows all function declarations allocated dynamically
// for each recursive call
// each call to an automatic function operates in an independent variable space.

module top;

function automatic integer factorial;
input [31:0]  oper;
integer i;
begin
   if (oper >=  2 ) factorial = factorial(oper-1) * oper;  // recursive functional call
   else factorial = 1;
end endfunction

integer result;
initial  result = factorial(4);

endmodule
```

constant Function (input arguments are constant)

```
// the arguments to a constant function are constant expressions

module ram_model (address, write, chip_select, data);
parameter data_width = 32;
parameter ram_depth = 256;
localparam addr_width = clogb2 (ram_depth);
input [addr_width – 1 : 0] address;
input write, chip_select;
inout [data_width – 1:0] data;

function integer clogb2 (input integer depth);
begin
   for (clogb2=0; depth >0; clogb2=clogb2+1);
   depth = depth >> 1;
end endfunction

reg [data_width – 1:0]  data_store [0:ram_depth – 1];
…..
endmodule
```

signed function (returned value is signed)

```
module top;
...

function automatic signed [63:0] compute_signed (input [63:0] vector);
...          // returned function value is signed
endfunction
…
always @ … begin
…
if (compute_signed(vector) < -3)
begin ... end
...
end
…
endmodule
```