Verilog: behavioral modeling, task and function

S. Palnitkar, "***Verilog HDL, A Guide to Digital Design and Synthesis***", 2nd ed., , Sun Microsystems, Inc, 2003.

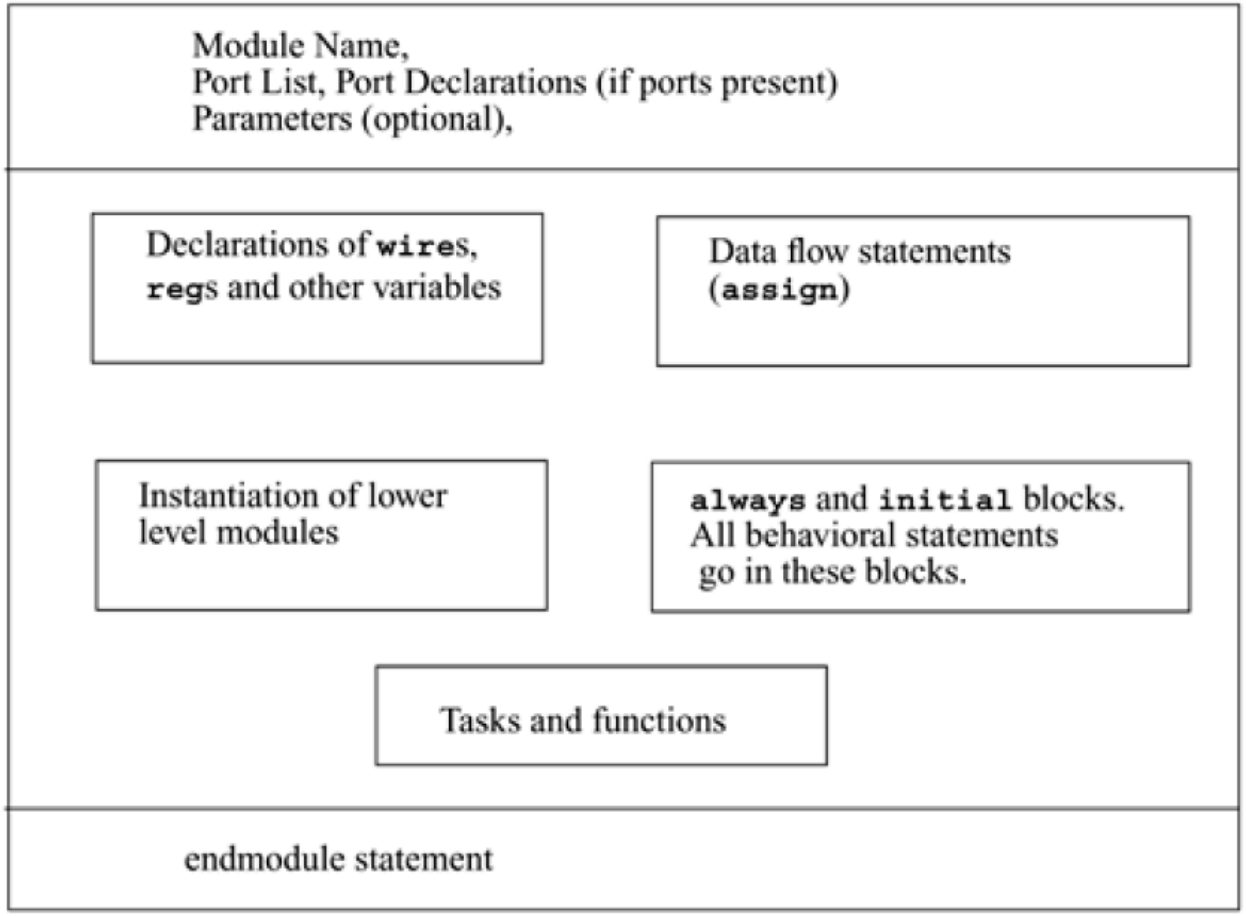
Outlines

- Behavioral Modeling
  - Blocking and non-blocking assignments
  - Asynchronous/synchronous reset DFF
- Clock Gating
- Tasks and Functions

Behavioral Modeling

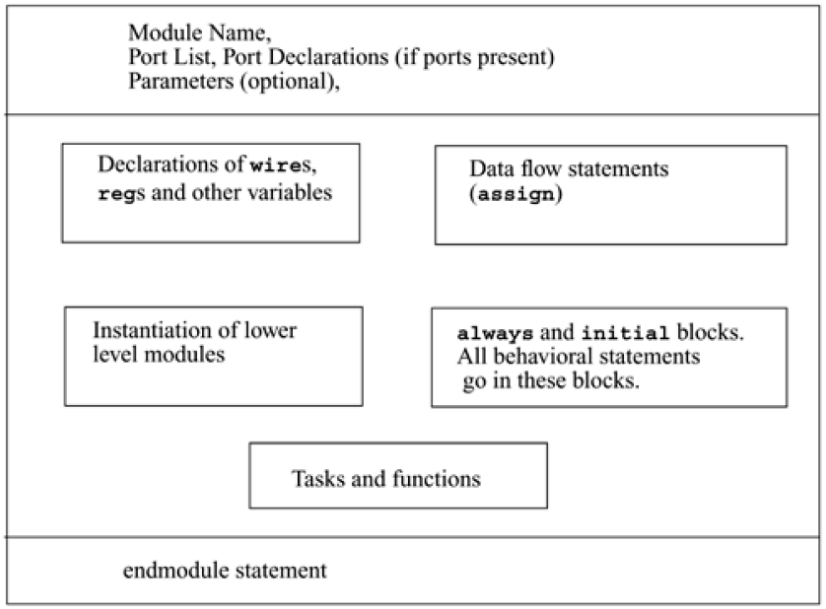
Components of a Verilog Module

- declaration
  - inputs, outputs
  - parameters
  - data types
- structure model
- module instantiation
- dataflow model
  - continuous assignment
- behavior model
  - procedural assignment



```
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;

// non-synthesizable coding
// used in testbench
initial
begin : blockname
  // statements
end

// behavioral modeling
always
begin
  // statements
end
```

```
// dataflow: continuous assignment
assign W1 = expr ;  // use operators

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

task T1;
input A1, A2;
output A3, A4;
begin
  // statements
end
endtask

function [7:0] F1;
input A1;
begin
  // statements
end
endfunction
```



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

initial statement

- start at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute *concurrently* at time 0
- Multiple sequential behavioral statements must be grouped using keywords **begin** and **end**
- typically used for initialization (for simulation, ***not for hardware design***), monitoring waveforms, …
  - **initial** statement is NOT synthesizable, cannot be used to initialize values of DFF in hardware design
    - ✓ instead, use synchronous or asynchronous set/reset, to be discussed later

always statement

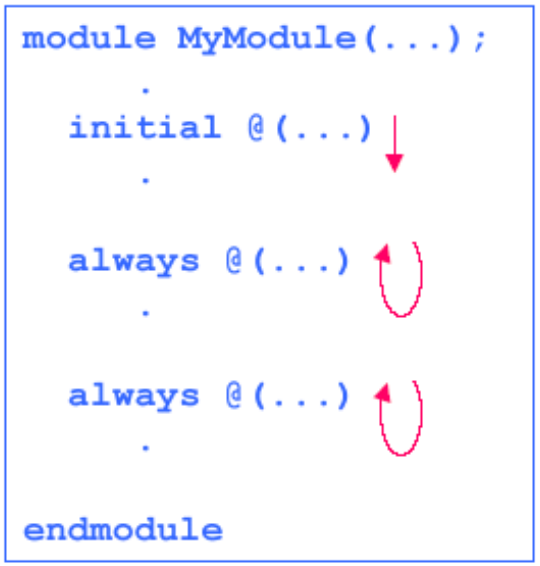
- starts at time 0 and executes the statements in the always block repeated*ly in a looping fashion*
- used to model a block of activity that is repeated continuously in a digital circuit
- usually include timing/event control (e.g., @posedge) or sensitivity list (e.g., @(a, b, c, …))

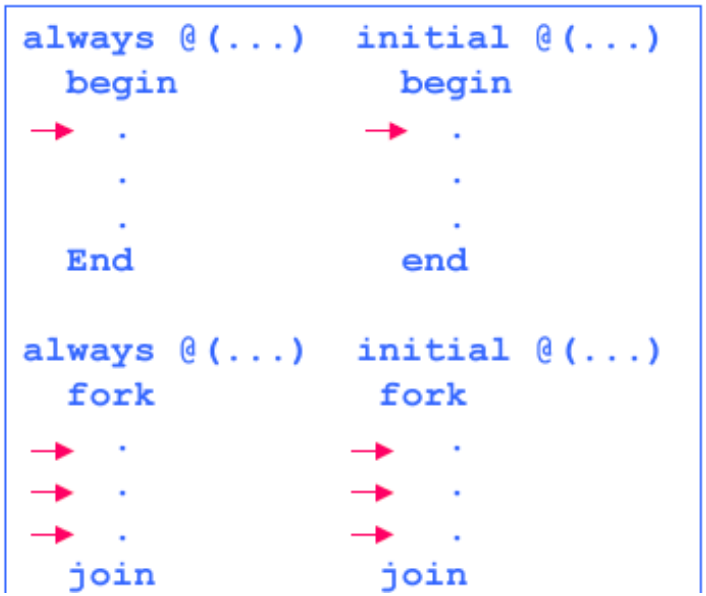
```
module clock_gen (output reg clock);
   initial clock = 1'b0;              // executed only once
   always #10 clock = ~clock;   // executed continuously
   initial #1000 $finish;             // finish execution at time step = 1000
endmodule
```

behavioral modeling

- statements inside **begin** … **end** are executed *sequentially*
  - but *non-blocking* assignments **<=** are executed in parallel
- 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`





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
```

Blocking Assignments (=)

- executed in the order they are specified in a sequential block (**begin** …. **end**)
  - Blocking statements are executed ***sequentially***

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

initial begin
        x=0;  y=1;  z=1;                    // scalar assignments
        count = 0;
        reg_a = 16'b0;   reg_b = reg_a;  // reg_b = 16`b0
#15   reg_a[2] =1'b1;                     // bit select assignment at time=15
# 10  reg_b[15:13] = {x,y,z};             // at time=25
        count = count +1;                 // at time=25
end
```

Combinational logic using blocking statements (=)

- it is reasonable to describe *combinational logic* using blocking statements
  - model forward cascade of combinational logic gates without feedback datapath

```
reg a, b, cout, sum

always @(in1 or in2)
begin
   a = in1 & in2;       // no assign keyword
   b = in1 | in2;
   {cout, sum} = a + b;
end
```

Non-blocking Assignments (<=)

- allow scheduling of assignments ***without blocking*** execution of the statements that follow in a sequential block
  - non-blocking statements are executed ***concurrently***
  - typically, non-blocking assignments are executed last in the time step, i.e., after all blocking assignments in the same time step are executed

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
      x=0; y=1; z=1;                    // scalar assignments
      count = 0;
      reg_a = 16'b0; reg_b = reg_a;

      // the above statements are executed sequentially, but at time=0
      // the following statements are executed in parallel after
      reg_a[2]      <= #15 1'b1;        // bit select assignment at time=15
      reg_b[15:13] <= #10 {x,y,z};      // at time=10
      count         <= count +1;        // at time =0
end
```

Two Steps in non-blocking assignments

- modeling concurrent data transfers that take place after a common event

```
always @(posedge clock)
begin
   reg1 <= #1 in1; // read in1 at posedge clk, write to reg1 after 1 unit delay
   reg2 <= @(negedge clock) in2^in3; // read in2, in3 at posedge clk,
   // and perform XOR. Then write the result to reg2 at negedge clk
   reg3 <= #1 reg1;  // read reg1 at poedge clk, and then write to reg3 after
   // 1 unit delay, i.e., reg3 gets the old value of reg1
end
```

1. ***read*** operation is performed on each right-hand-side (RHS) variable at positive edge of the clock
2. ***write*** operations to the left-hand-side variables (LHS) are scheduled to be executed according to delay control
3. the ***write*** operations are executed at the scheduled time

emulate non-blocking using blocking

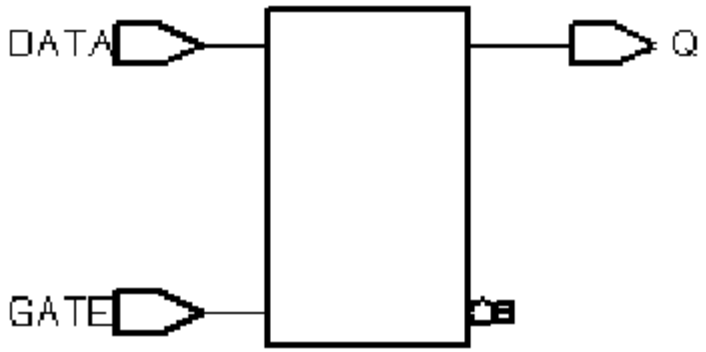
```
always @(posedge clk)
begin
// read operations from RHS
     temp_a = a;
     temp_b = b;
// write operations to LHS
     b = temp_a;
     a = temp_b;
end
```

```
// FF a and FF b swap
// at every clock edge
always @(posedge clk)
begin
a <= b;
b <= a;
end
```

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
```

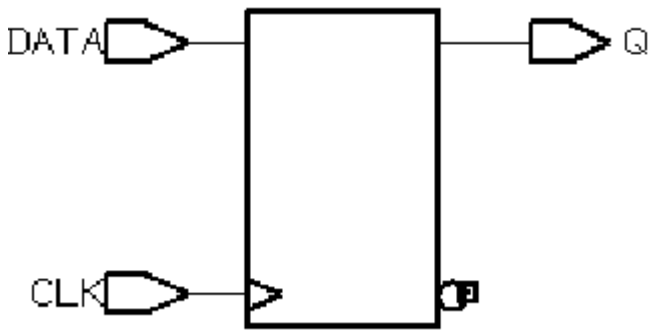



```
module d_CL (
input gate, data,
output reg q );

// 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

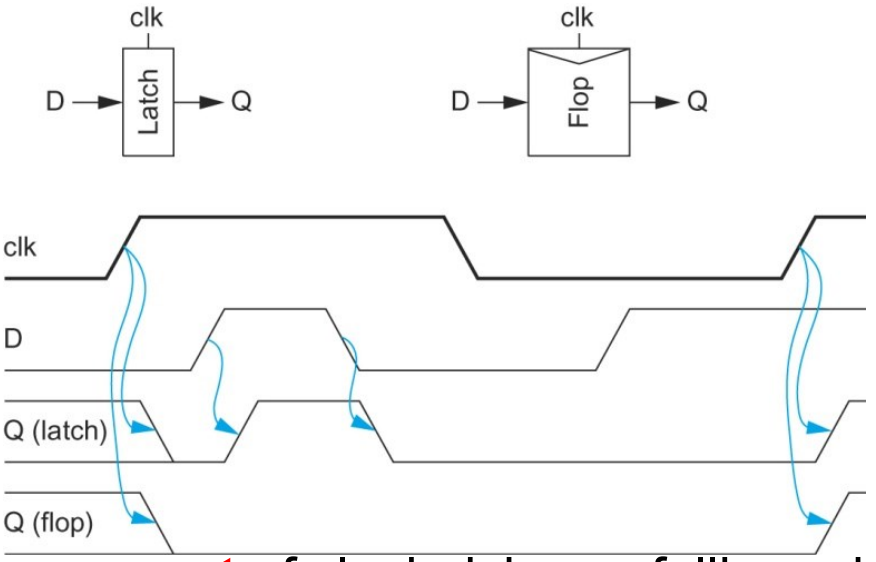
```
module dff_pos (DATA, CLK, Q);
  input DATA, CLK;
  output Q;
  reg Q;

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



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;
```

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
```

Non-blocking avoid race condition

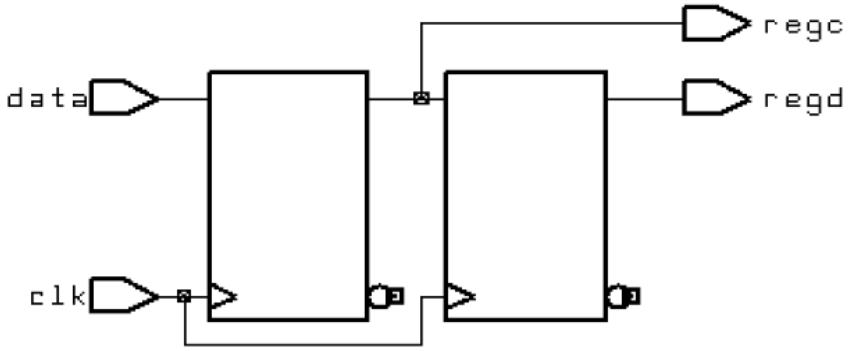
```
// two concurrent always blocks with blocking statements
// the following statements are NOT OK
always @(posedge clk) a = b;
always @(posedge clk) b = a;
// either a=b are executed before b=a, or vice versa
// but the execution order matters
// both registers will get same value (previous value of a or b)
```

```
// two concurrent always blocks with nonblocking statements
// the following statements are OK
always @(posedge clk) a <= b;
always @(posedge clk) b <= a;
// at posedge clk, the values of all RHS variables are read,
// and stored in temporary variables, waiting to be evaluated
// reg a and reg b swap value
```

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

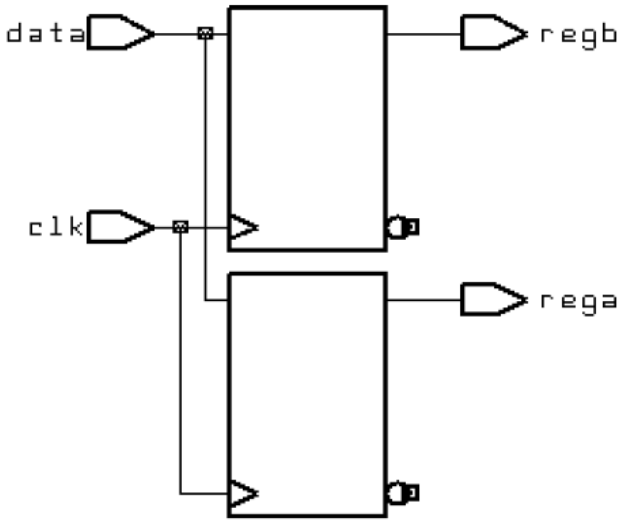
- non-blocking assignment (**<=**)

```
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
```



Some rules of coding for <= and =

- In a procedural block (say **always**), never mix blocking (=) and non-blocking (<=) assignments
- For complicated *combinational* logic, use blocking assignments **=**
- Use non-blocking assignments **<=** for clocked *sequential* logic (with results stored in DFF)
- Never assign any data object in a module from more than one **always** block
  - Avoid race condition in *multiple assignment*
  - e.g.

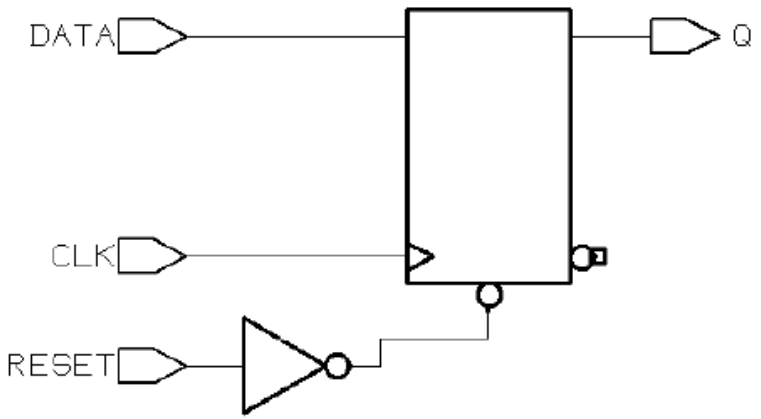
    always @ (posedge clk) q <= a+b;

    always @ (posedge clk) q <= a-b;
- DO NOT mix level and edge events in the sensitivity list of a procedural block
  - e.g. **always @ (posedge** clk, **negedge** reset_n) is good, but **always @ (posedge** clk, reset_n) is NOT good.

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

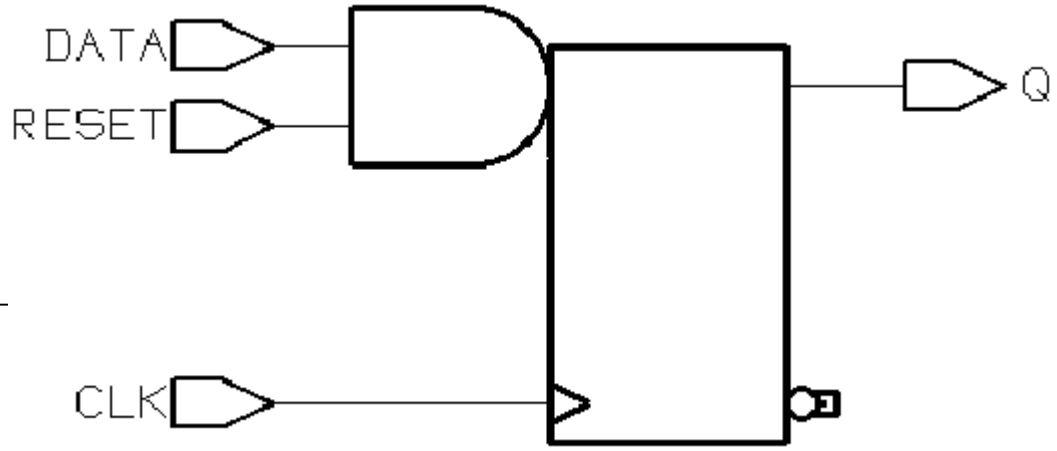
```
module dff_async_reset (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

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



DFF with synchronous reset (active-low)

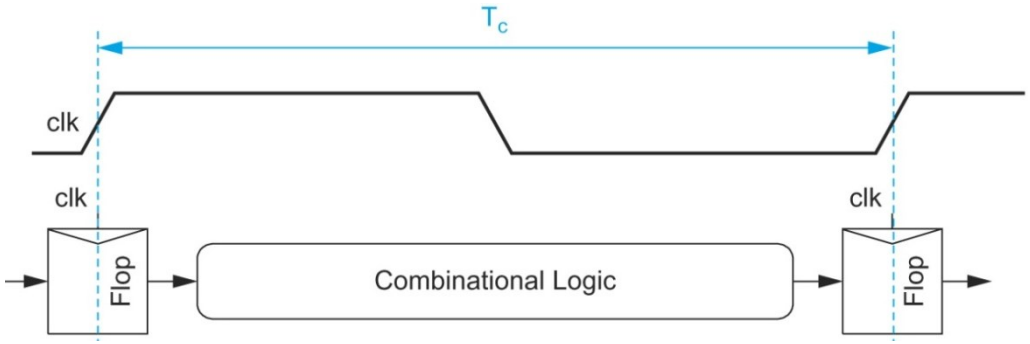
- reset FF only at clock edge
  - synchronous FF is regular FF with logic gate for input

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

//synopsys sync_set_reset "RESET"
always @(posedge CLK)
  if (~RESET)
    Q <= 1'b0;
  else
    Q <= DATA;
endmodule
```

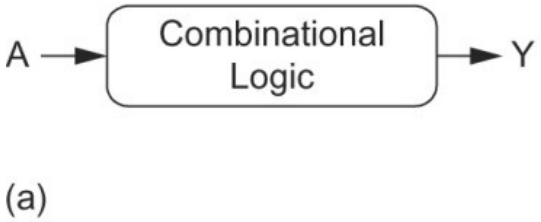


Timing in Sequential Logic

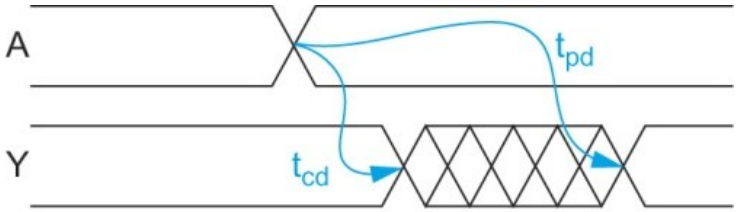
- sequential logc = combinational block + FF's

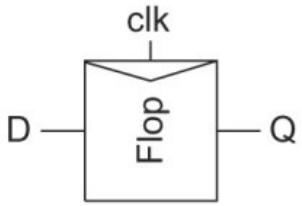


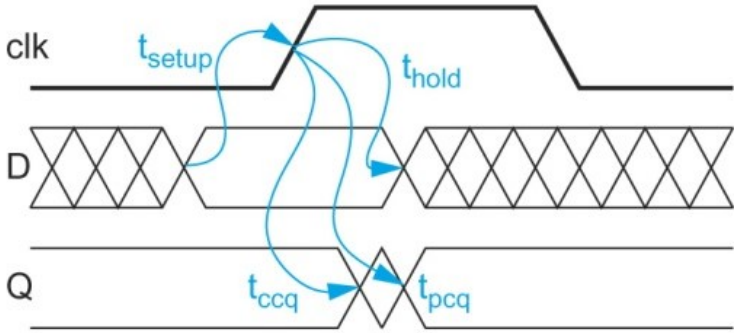
- ***setup time*** and ***hold time*** of FFs



(a)







Critical path delay

- Synopsys static timing analysis tool reports the critical path delay by finding the maximum delay of the following 4 types of paths
  - Primary inputs to primary outputs
    - ✓ e.g., the delay in fully combinational logic
  - Primary input to the data inputs of FFs
    - ✓ e.g., the delay of the 1st pipelined stage
  - Outputs of FFs to data inputs of next FFs
    - ✓ e.g., the delay of the 2nd pipeline stage
  - Outputs of FFs to primary outputs
    - ✓ e.g., the delay AFTER the last pipeline stage where the primary outputs are NOT registered

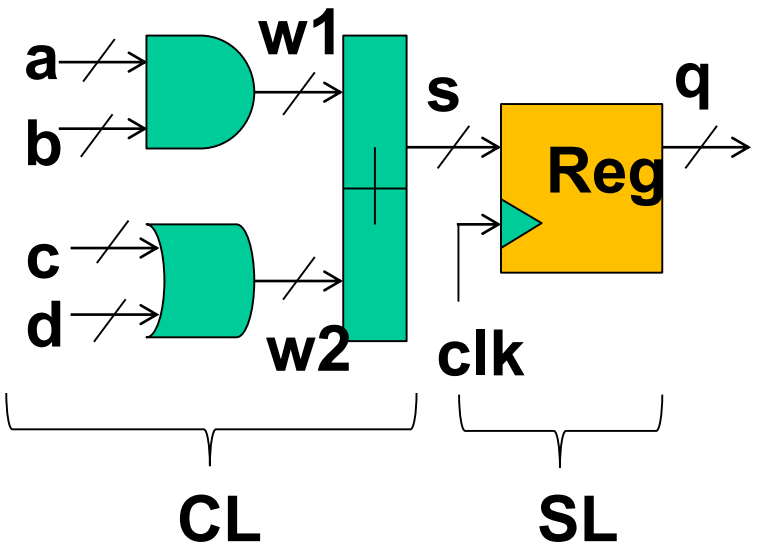
Delay of each pipelined stage

- If all pipeline stages are modeled in a single module
  - Need to identify the critical path of each pipeline stage from the synthesized messy gate-level netlists, which might be
    - ✓Primary input (PI) to data input of some FFs, or
    - ✓Outputs of some FFs to data inputs of other FFs, or
- Model each pipeline stage with a module
  - e.g., M1(I1, O1), M2(I2, O2), …
- and then instantiate all the pipeline-stage modules
- could easily identify the delays of the following paths
  - 1st pipeline stage delay T1: from I1_ext to O1_ext
  - 2nd pipeline stage delay T2: from I2_ext to O2_ext
  - …

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
```

regular *delay* control (#T c=a+b;)

```
// evaluation from RHS and assignment to LHS
// at the scheduled time
parameter latency = 20;
reg x, y, z, p, q;
begin
x = 0 ; // no delay
#10 y=1; // delay execution of y=1 by 10 time units
#(latency) z=0;
#(4:5:6) q=0; // min, typical and max delays
end
```

Intra-assignment delay (c = #T a+b;)

// evaluation from RHS immediately
// and then assignment to LHS after the scheduled time

```
reg x, y, z;
initial begin
x=0; z=0;   // initialization of x and z at time=0
y= #5 x+z; // take values of x and z at time=0, evaluate x+z,
                // and then wait for 5 time units before assigning x+z valuie to y
end

/* equivalent method with temporary variables and regular delay control */
initial begin
x=0; z=0;
temp_xz = x+z;  // x+z is executed at time=0
#5 y=temp_xz;   // y is assigned the value of x+z at time=5
end
```

event control @ (regular or intra-assighment)

@(clk) q=d;
// q=d is executed when clk changes value

@(**posedge** clk) q<=d;
// q=d is executed whenever clk does a
// positive transition (i.e., positive edge)

```
always
@ (posedge clk) q<=d;
```

```
always @ (posedge clk)
    q<=d;
```

q <= @(**posedge** clk) d;
// d is evaluated immediately,
// and assigned to q at the next positive edge of clk

```
always @ (negedge clk)
    q <= @(posedge clk) d;
```

named event control

```
// define an event called received_data
event received_data;
always @(posedge clk)
begin
// an event is triggered by the symbol ->
    if (last_data_packet) -> received_data;
end

// await triggering of event received_data
always @(received_data)
   data_buf={packet[0], packet[1], packet[2]};
```

event OR control (sensitivity list)

```
// level-sensitive latch with asynchronous reset
always @ (rst or clk or d) // use or operator
begin
   if (rst) q=1'b0;              // if rst=1, set q to 0
   else if (clk) q=d;            // else if clk=1, latch input
end                              // if rst=clk=0, latch keeps old value
```

```
always @ (rst, clk, d)      // use comma (,) instead of or (IEEE 1364-2001)
begin
  if (rst) q=1'b0;
  else if (clk) q=d;
end
```

```
// a positive edge triggered D flip-flop with asynchronous falling reset
// DO NOT mix edge signals and regular signals in sensitivity list
always @ (posedge clk, negedge rst_n)  // active-low asynch. reset
if ( !rst_n) q <= 0;
else     q <= d;
```

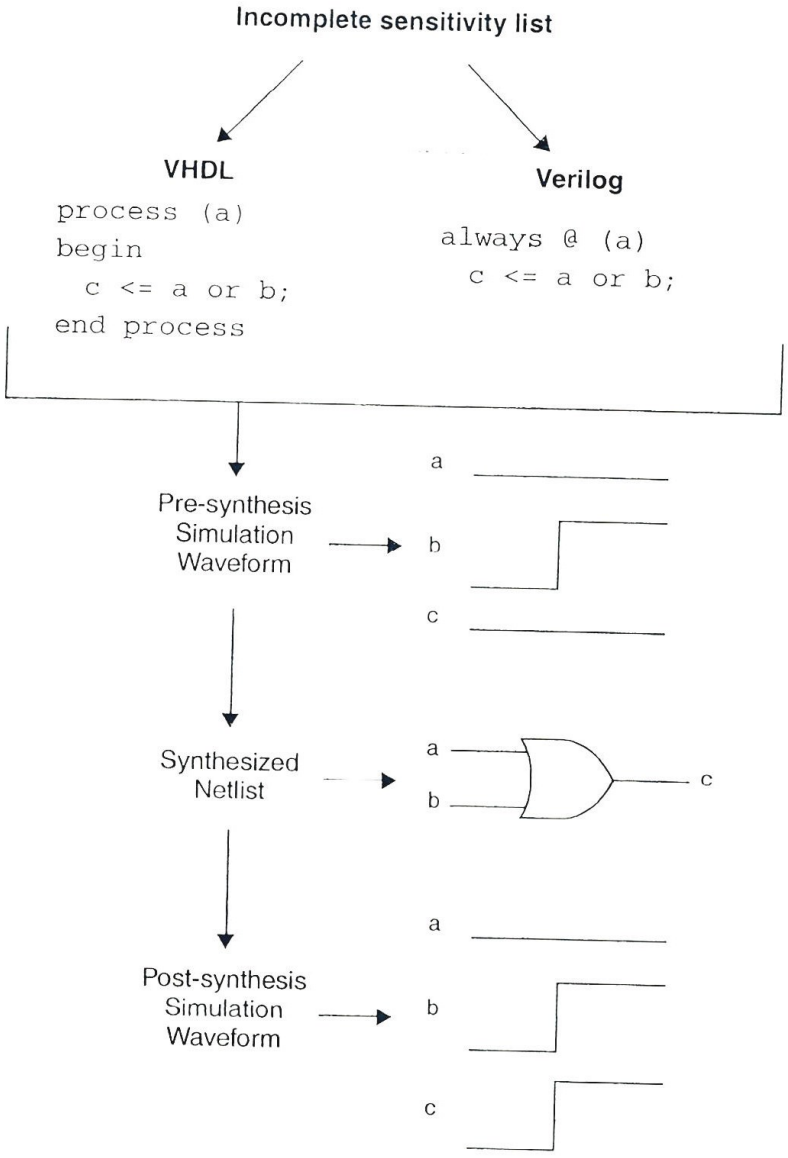
Use of @*, or @(*) for CL

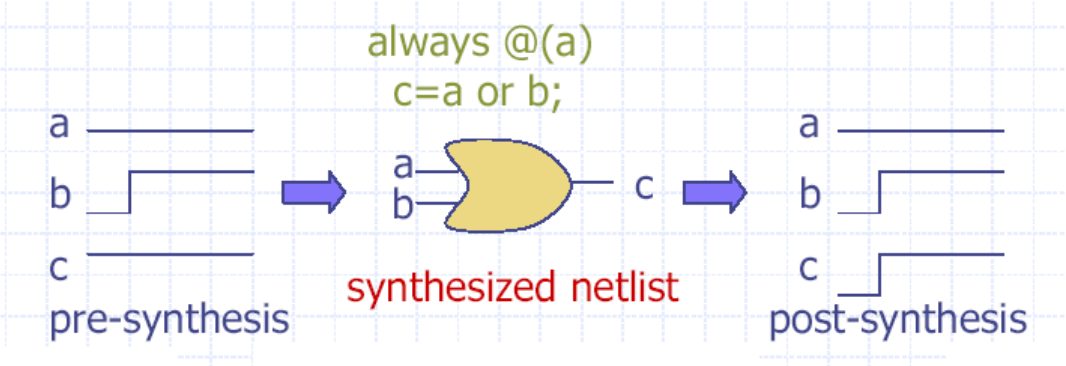
```
always @(a or b or c or d or e or f or g or h or p or m)
begin
out1 = a ? b+c: d+e;      // out1, out2 should be reg
out2 = f ? g+h: p+m;
end
```

```
// 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





Level-Sensitive Timing control: wait

always

wait (*count_enable*) #20 count = count+1;

// wait for *count_enable* = 1'b1

// count will be incremented (with delay=20)

// only when *count_enable* is at logic 1

// cp. **@** (event) …

// wait for the *change* of a signal value, or

// for the triggering of an event

behavioral statements (used inside initial or always blocks)

- procedural assignments
  - blocking (**=**) vs. non-blocking (**<=**)
  - operators (such as +, -, *, &, |, ^, ?:, …)
- **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**)
- timing control (**#**)

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");
```

Verilog vs. VHDL (conditional)

```
// Verilog

if (...)
   begin ... ;  end
else if (…)
   begin …; end
else if (…)
   begin …; end
else
   begin …; end
```

```
-- VHDL

if (…) then
…
elsif (…) then
…
elsif (…) then
…
else
…
end if;
```

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
```

behavioral statements (used inside initial or always blocks)

- procedural assignments
  - blocking (**=**) vs. non-blocking (**<=**)
  - operators (such as +, -, *, &, |, ^, ?:, …)
- 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**)
- timing control (**#**)

Multiway Branching (case Statements)

```
wire [31:0] x, z;
reg [31:0] y;
wire [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");
```

Verilog vs. VHDL (multi-way branching)

```
// Verilog

case ( )
      …      : begin … end ;
      …      : begin … end ;
   default :  begin … end ;
endcase;
```

```
-- VHDL

case … is
   when …          => … ;
   when …          => … ;
   when others => … ;
end case;
```

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
```

case statement with x and z

```
// 1-to-4 demultiplexer (in, s[1:0], out1, out2, out3, out4)

reg out0, out1, out2, out3;

always @ (s1 or s0 or in)
case ({s1, s0})
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'b11 : begin out0 = 1'bz;  out1 = 1'bz;  out2=1'bz;  out3=in;    end
2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
        begin out0=1'bx;   out1=1'bx;   out2=1'bx;   out3=1b'x; end
2'bz0, 2'bz1, 2b'zz, 2'b0z, 2'b1z :
        begin out0=1'bz;   out1=1'bz;   out2=1'bz;   out3=1b'z; end
default: $display ("unspecified control signals");
endcase
```

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
```

parallel case, but not full case

- infer a latch for variable b to keep the value in cases that are not specified

```
// parallel case, but not full case
// infer a multiplexer with latched output b

input [1:0] a;
always @ (a, w,  z) begin
   case (a)
      2'b11: b=w;
      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

```
wire [3:0] encoding;
integer next_state;  // integer is reg type

// 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
```

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)**

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
```

repeat Loop

- repeat construct must contain a constant, a variable or a signal value
  - If the number is a variable or signal value, it is evaluated only when the loop starts, and not during the loop execution

```
integer count;

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

```
module data_buffer(data_start, data, clk);
parameter cycles=8;
input data_start;
input [15:0] data;
input clk;
reg [15:0] buffer[0:7];
integer i;

always @(posedge clk) begin
        if (data_start) begin
                i=0;
                // store data at next 8 clock edges
                repeat (cycles)
                begin
                        // wait till next posedge to latch data
                        @(posedge clk) buffer[i]=data;
                        i=i+1;
                end
        end
end
endmodule
```

forever loop

- execute forever until the **$finish** task
  - non-synthesizable

```
// clock generation using
// forever loop instead of
// always block
reg clk;

initial
begin
clk=1'b0;
forever #10 clk=~clk;
end
```

```
// synchronize two register values
// at every positive edge of clk
reg clk;
reg x, y;

initial
forever @(posedge clk)  x = y;
```

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

```
always @ (*) begin
   c[0] = cin;
   for (i=0; i<=31; i=i+1)
      {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];
```

Leading One Detector (LOD (using casex)

```
module LZD_case (
                input [15:0]din,
                output reg [3:0]zero_cnt);
always@(*) begin
  casex(din)
  16'b1xxx_xxxx_xxxx_xxxx : zero_cnt = 4'd0;
  16'b01xx_xxxx_xxxx_xxxx : zero_cnt = 4'd1;
  16'b001x_xxxx_xxxx_xxxx : zero_cnt = 4'd2;
  16'b0001_xxxx_xxxx_xxxx : zero_cnt = 4'd3;
  16'b0000_1xxx_xxxx_xxxx : zero_cnt = 4'd4;
  16'b0000_01xx_xxxx_xxxx : zero_cnt = 4'd5;
  16'b0000_001x_xxxx_xxxx : zero_cnt = 4'd6;
  16'b0000_0001_xxxx_xxxx : zero_cnt = 4'd7;
  16'b0000_0000_1xxx_xxxx : zero_cnt = 4'd8;
  16'b0000_0000_01xx_xxxx : zero_cnt = 4'd9;
  16'b0000_0000_001x_xxxx : zero_cnt = 4'd10;
  16'b0000_0000_0001_xxxx : zero_cnt = 4'd11;
  16'b0000_0000_0000_1xxx : zero_cnt = 4'd12;
  16'b0000_0000_0000_01xx : zero_cnt = 4'd13;
  16'b0000_0000_0000_001x : zero_cnt = 4'd14;
  16'b0000_0000_0000_0001 : zero_cnt = 4'd15;
  default : zero_cnt = 4'd0;
  endcase
end
endmodule
```

	Global cell area		Local cell area			
Hierarchical cell	Absolute Total	Percent Total	Combi-national	Noncombi-national	Black-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	

```
module LZD_case ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n15, n16, n17,
        n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30, n32;

  INVX2 U3 ( .A(n25), .Y(n3) );
  NAND3XL U4 ( .A(n5), .B(n4), .C(n24), .Y(n25) );
  NAND2X2 U5 ( .A(n17), .B(n10), .Y(n16) );
  INVX2 U6 ( .A(n17), .Y(n2) );
  INVX2 U7 ( .A(n29), .Y(n6) );
  OAI21X1 U8 ( .A0(n10), .A1(n2), .B0(n11), .Y(zero_cnt[3]) );
  NOR3BX1 U9 ( .AN(n18), .B(din[3]), .C(din[4]), .Y(n10) );
  NOR4BX2 U10 ( .AN(din[0]), .B(n16), .C(din[1]), .D(din[2]), .Y(n15) );
  NOR3X1 U11 ( .A(n25), .B(din[8]), .C(n6), .Y(n17) );
  OAI31X1 U12 ( .A0(n9), .A1(din[2]), .A2(n16), .B0(n23), .Y(n22) );
  INVX2 U13 ( .A(din[1]), .Y(n9) );
  NOR3X1 U14 ( .A(din[11]), .B(din[9]), .C(din[10]), .Y(n29) );
  NOR3X1 U15 ( .A(din[6]), .B(din[7]), .C(din[5]), .Y(n18) );
  NOR2X2 U16 ( .A(din[13]), .B(din[12]), .Y(n24) );
  NOR2X2 U17 ( .A(n32), .B(n15), .Y(n11) );
  AOI2BB1X2 U18 ( .A0N(din[1]), .A1N(din[2]), .B0(n16), .Y(n32) );
  OAI2B11X2 U19 ( .A1N(din[2]), .A0(n16), .B0(n23), .C0(n26), .Y(zero_cnt[0])
       );
  AOI21X1 U20 ( .A0(n27), .A1(n4), .B0(n28), .Y(n26) );
  OAI2B1X1 U21 ( .A1N(din[12]), .A0(din[13]), .B0(n5), .Y(n27) );
  OAI33X2 U22 ( .A0(n7), .A1(din[11]), .A2(n25), .B0(n8), .B1(din[7]), .B2(n2),
       .Y(n28) );
  NAND3XL U23 ( .A(n12), .B(n13), .C(n11), .Y(zero_cnt[2]) );
  OAI21X1 U24 ( .A0(din[8]), .A1(n6), .B0(n3), .Y(n12) );
  NAND4BX2 U25 ( .AN(din[4]), .B(din[3]), .C(n17), .D(n18), .Y(n13) );
  AOI31X1 U26 ( .A0(din[8]), .A1(n3), .A2(n29), .B0(n1), .Y(n23) );
  INVX2 U27 ( .A(n30), .Y(n1) );
  AOI31X1 U28 ( .A0(n17), .A1(n18), .A2(din[4]), .B0(n15), .Y(n30) );
  OR4X2 U29 ( .A(n19), .B(n20), .C(n21), .D(n22), .Y(zero_cnt[1]) );
  NOR3X1 U30 ( .A(n24), .B(din[15]), .C(din[14]), .Y(n21) );
  NOR4BX2 U31 ( .AN(din[9]), .B(din[11]), .C(din[10]), .D(n25), .Y(n20) );
  NOR4BX2 U32 ( .AN(din[5]), .B(din[7]), .C(din[6]), .D(n2), .Y(n19) );
  INVX2 U33 ( .A(din[15]), .Y(n4) );
  INVX2 U34 ( .A(din[14]), .Y(n5) );
  INVX2 U35 ( .A(din[10]), .Y(n7) );
  INVX2 U36 ( .A(din[6]), .Y(n8) );
endmodule
```

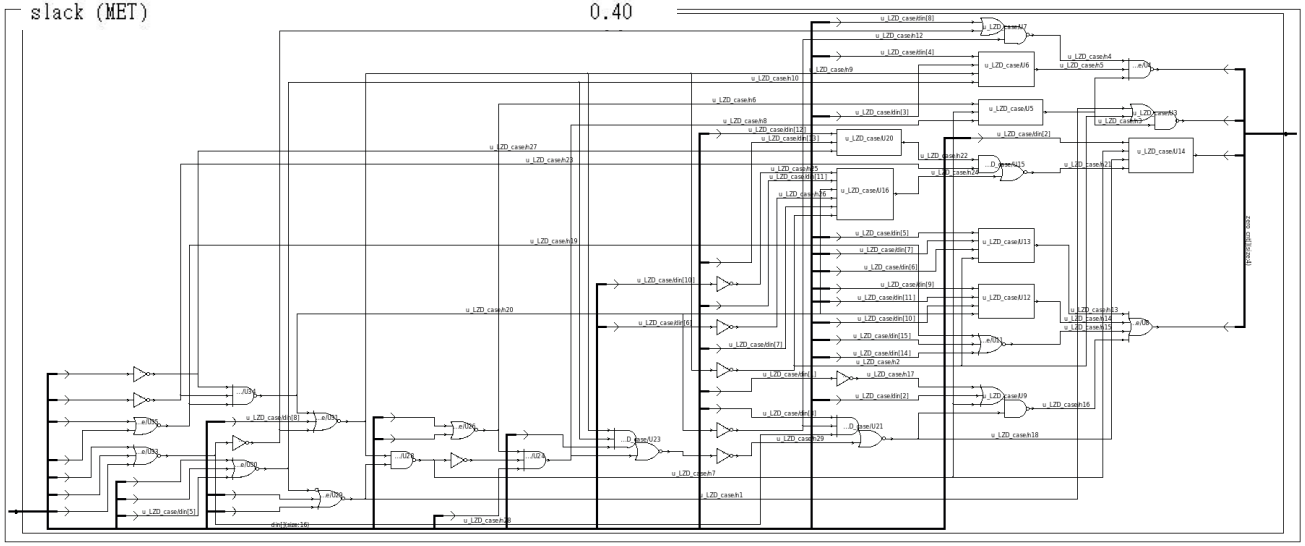
LOD (using casex) after synthesis

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combinational	Local cell area: Noncombinational	Local cell area: Black-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	

Point	Incr	Path
input external delay	0.00	0.00 f
din[12] (in)	0.00	0.00 f
U56/Y (NOR2X2)	0.03	0.03 r
U39/Y (NOR3BX1)	0.11	0.14 r
U34/Y (INVX2)	0.03	0.17 f
U40/Y (NOR3BX1)	0.09	0.26 r
U33/Y (INVX2)	0.04	0.30 f
U41/Y (NOR4X2)	0.10	0.40 r
U49/Y (NAND4BX2)	0.06	0.46 f
U38/Y (AND3X2)	0.07	0.52 f
U61/Y (OAI31X1)	0.02	0.55 r
U58/Y (OR4X2)	0.06	0.60 r
zero_cnt[1] (out)	0.00	0.60 r
data arrival time		0.60
max_delay	1.00	1.00
output external delay	0.00	1.00
data required time		1.00
data required time		1.00
data arrival time		-0.60
slack (MET)		0.40

Delay=0.60ns

Area=126 um x um (um^2)
~=(126/2.82) NAND2 gates
~= 45 gate equivalent



```
module LZD_case ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n15, n16, n17,
         n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30, n32;

  INVX2 U3 ( .A(n25), .Y(n3) );
  NAND3XL U4 ( .A(n5), .B(n4), .C(n24), .Y(n25) );
  NAND2X2 U5 ( .A(n17), .B(n10), .Y(n16) );
  INVX2 U6 ( .A(n17), .Y(n2) );
  INVX2 U7 ( .A(n29), .Y(n6) );
  OAI21X1 U8 ( .A0(n10), .A1(n2), .B0(n11), .Y(zero_cnt[3]) );
  NOR3BX1 U9 ( .AN(n18), .B(din[3]), .C(din[4]), .Y(n10) );
  NOR4BX2 U10 ( .AN(din[0]), .B(n16), .C(din[1]), .D(din[2]), .Y(n15) );
  NOR3X1 U11 ( .A(n25), .B(din[8]), .C(n6), .Y(n17) );
  OAI31X1 U12 ( .A0(n9), .A1(din[2]), .A2(n16), .B0(n23), .Y(n22) );
  INVX2 U13 ( .A(din[1]), .Y(n9) );
  NOR3X1 U14 ( .A(din[11]), .B(din[9]), .C(din[10]), .Y(n29) );
  NOR3X1 U15 ( .A(din[6]), .B(din[7]), .C(din[5]), .Y(n18) );
  NOR2X2 U16 ( .A(din[13]), .B(din[12]), .Y(n24) );
  NOR2X2 U17 ( .A(n32), .B(n15), .Y(n11) );
  AOI2BB1X2 U18 ( .A0N(din[1]), .A1N(din[2]), .B0(n16), .Y(n32) );
  OAI2B11X2 U19 ( .A1N(din[2]), .A0(n16), .B0(n23), .C0(n26), .Y(zero_cn
       );
  AOI21X1 U20 ( .A0(n27), .A1(n4), .B0(n28), .Y(n26) );
  OAI2B1X1 U21 ( .A1N(din[12]), .A0(din[13]), .B0(n5), .Y(n27) );
  OAI33X2 U22 ( .A0(n7), .A1(din[11]), .A2(n25), .B0(n8), .B1(din[7]), .B2(
       .Y(n28) );
  NAND3XL U23 ( .A(n12), .B(n13), .C(n11), .Y(zero_cnt[2]) );
  OAI21X1 U24 ( .A0(din[8]), .A1(n6), .B0(n3), .Y(n12) );
  NAND4BX2 U25 ( .AN(din[4]), .B(din[3]), .C(n17), .D(n18), .Y(n13) );
  AOI31X1 U26 ( .A0(din[8]), .A1(n3), .A2(n29), .B0(n1), .Y(n23) );
  INVX2 U27 ( .A(n30), .Y(n1) );
  AOI31X1 U28 ( .A0(n17), .A1(n18), .A2(din[4]), .B0(n15), .Y(n30) );
  OR4X2 U29 ( .A(n19), .B(n20), .C(n21), .D(n22), .Y(zero_cnt[1]) );
  NOR3X1 U30 ( .A(n24), .B(din[15]), .C(din[14]), .Y(n21) );
  NOR4BX2 U31 ( .AN(din[9]), .B(din[11]), .C(din[10]), .D(n25), .Y(n20) );
  NOR4BX2 U32 ( .AN(din[5]), .B(din[7]), .C(din[6]), .D(n2), .Y(n19) );
  INVX2 U33 ( .A(din[15]), .Y(n4) );
  INVX2 U34 ( .A(din[14]), .Y(n5) );
  INVX2 U35 ( .A(din[10]), .Y(n7) );
  INVX2 U36 ( .A(din[6]), .Y(n8) );
endmodule
```

16-bit LOD using casex (~45 gates)

```
module LZD_case (nput [15:0]din,  output reg [3:0]zero_cnt);
always@(*) begin
  casex(din)
  16'b1xxx_xxxx_xxxx_xxxx : zero_cnt = 4'd0;
  16'b01xx_xxxx_xxxx_xxxx : zero_cnt = 4'd1;
  16'b001x_xxxx_xxxx_xxxx : zero_cnt = 4'd2;
  16'b0001_xxxx_xxxx_xxxx : zero_cnt = 4'd3;
  16'b0000_1xxx_xxxx_xxxx : zero_cnt = 4'd4;
  16'b0000_01xx_xxxx_xxxx : zero_cnt = 4'd5;
  16'b0000_001x_xxxx_xxxx : zero_cnt = 4'd6;
  16'b0000_0001_xxxx_xxxx : zero_cnt = 4'd7;
  16'b0000_0000_1xxx_xxxx : zero_cnt = 4'd8;
  16'b0000_0000_01xx_xxxx : zero_cnt = 4'd9;
  16'b0000_0000_001x_xxxx : zero_cnt = 4'd10;
  16'b0000_0000_0001_xxxx : zero_cnt = 4'd11;
  16'b0000_0000_0000_1xxx : zero_cnt = 4'd12;
  16'b0000_0000_0000_01xx : zero_cnt = 4'd13;
  16'b0000_0000_0000_001x : zero_cnt = 4'd14;
  16'b0000_0000_0000_0001 : zero_cnt = 4'd15;
  default : zero_cnt = 4'd0;
  endcase
end
endmodule
```

Delay=0.60ns

Area=126 um x um (um^2)
~=(126/2.82) NAND2 gates
~= 45 gate equivalent

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-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

LOD
(using for loop)

```
`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
```

```
module LZD_for ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16
         n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27;

  AOI211X2 U2 ( .A0(n10), .A1(n11), .B0(n5), .C0(n12), .Y(zero_cnt[3]) );
  INVX2 U3 ( .A(n13), .Y(n5) );
  NAND4X2 U4 ( .A(n4), .B(n3), .C(n2), .D(n1), .Y(n12) );
  AOI21X1 U5 ( .A0(n13), .A1(n14), .B0(n12), .Y(zero_cnt[2]) );
  NAND2BX2 U6 ( .AN(n10), .B(n11), .Y(n14) );
  NOR4X2 U7 ( .A(din[10]), .B(din[11]), .C(din[8]), .D(din[9]), .Y(n13) );
  NOR4X2 U8 ( .A(din[4]), .B(din[5]), .C(din[6]), .D(din[7]), .Y(n11) );
  NOR4X2 U9 ( .A(din[0]), .B(din[1]), .C(din[2]), .D(din[3]), .Y(n10) );
  AOI31X1 U10 ( .A0(n7), .A1(n6), .A2(n19), .B0(n20), .Y(n17) );
  INVX2 U11 ( .A(din[4]), .Y(n7) );
  OR2X2 U12 ( .A(din[7]), .B(din[6]), .Y(n20) );
  OAI211XL U13 ( .A0(din[1]), .A1(din[0]), .B0(n9), .C0(n8), .Y(n19) );
  AOI2B1X1 U14 ( .A1N(din[6]), .A0(n25), .B0(din[7]), .Y(n24) );
  OAI21X1 U15 ( .A0(n26), .A1(din[4]), .B0(n6), .Y(n25) );
  AOI21X1 U16 ( .A0(n9), .A1(n27), .B0(din[3]), .Y(n26) );
  NAND2BX2 U17 ( .AN(din[1]), .B(din[0]), .Y(n27) );
  AOI31X1 U18 ( .A0(n4), .A1(n3), .A2(n15), .B0(n16), .Y(zero_cnt[1]) );
  NAND2X2 U19 ( .A(n2), .B(n1), .Y(n16) );
  OAI31X1 U20 ( .A0(n17), .A1(din[9]), .A2(din[8]), .B0(n18), .Y(n15) );
  NOR2X2 U21 ( .A(din[11]), .B(din[10]), .Y(n18) );
  AOI21X1 U22 ( .A0(n2), .A1(n21), .B0(din[15]), .Y(zero_cnt[0]) );
  OAI21X1 U23 ( .A0(n22), .A1(din[12]), .B0(n3), .Y(n21) );
  AOI2B1X1 U24 ( .A1N(din[10]), .A0(n23), .B0(din[11]), .Y(n22) );
  OAI21BX1 U25 ( .A0(n24), .A1(din[8]), .B0N(din[9]), .Y(n23) );
  INVX2 U26 ( .A(din[14]), .Y(n2) );
  INVX2 U27 ( .A(din[13]), .Y(n3) );
  INVX2 U28 ( .A(din[15]), .Y(n1) );
  INVX2 U29 ( .A(din[12]), .Y(n4) );
  INVX2 U30 ( .A(din[2]), .Y(n9) );
  INVX2 U31 ( .A(din[5]), .Y(n6) );
  INVX2 U32 ( .A(din[3]), .Y(n8) );
endmodule
```

LOD (using for loop) after synthesis

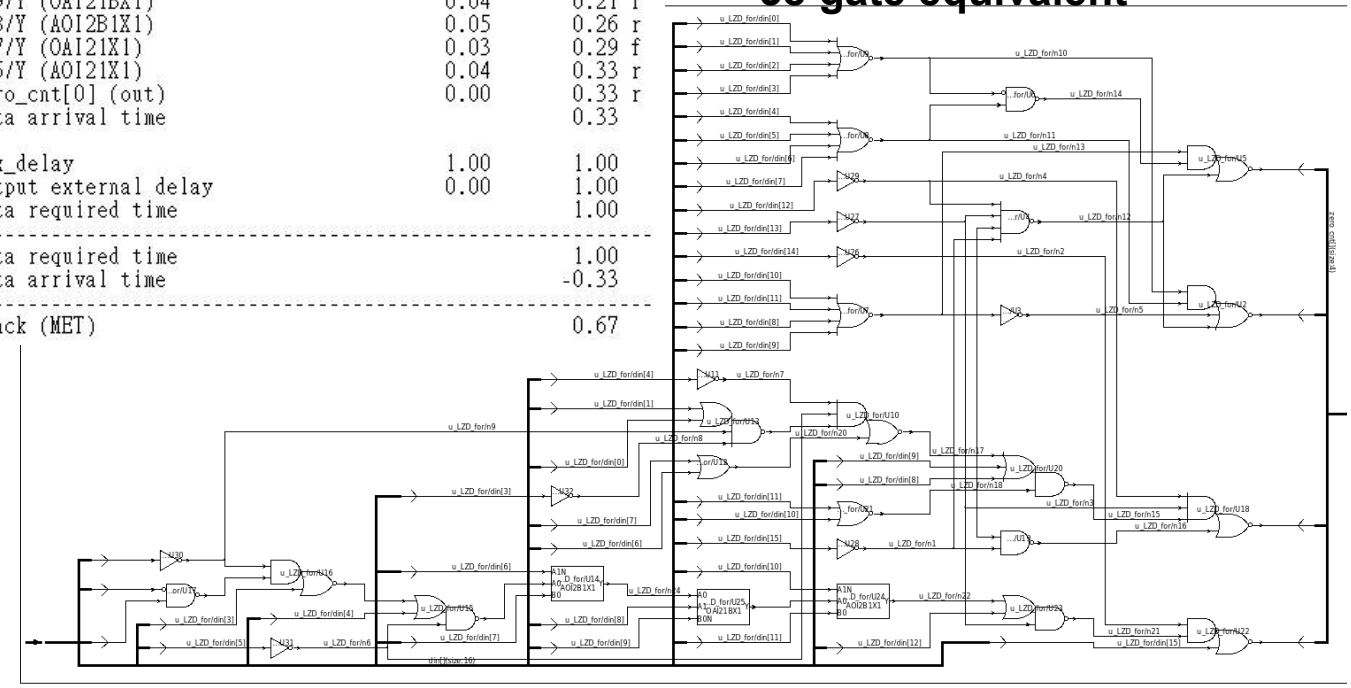
- synthesized gate-level netlist (TSMC 90nm)

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-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	

Delay=0.33ns

Point	Incr	Path
input external delay	0.00	0.00 f
din[1] (in)	0.00	0.00 f
U54/Y (NAND2BX2)	0.05	0.05 f
U53/Y (AOI21X1)	0.04	0.09 r
U52/Y (OAI21X1)	0.03	0.12 f
U51/Y (AOI21X1)	0.05	0.17 r
U49/Y (OAI21BX1)	0.04	0.21 f
U48/Y (AOI2B1X1)	0.05	0.26 r
U47/Y (OAI21X1)	0.03	0.29 f
U46/Y (AOI21X1)	0.04	0.33 r
zero_cnt[0] (out)	0.00	0.33 r
data arrival time		0.33
max_delay	1.00	1.00
output external delay	0.00	1.00
data required time		1.00
data required time		1.00
data arrival time		-0.33
slack (MET)		0.67

Area=106 um x um (um^2)
~=(106/2.82) NAND2 gates
~=38 gate equivalent



```
module LZD_for ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n1
         n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27;

  AOI211X2 U2 ( .A0(n10), .A1(n11), .B0(n5), .C0(n12), .Y(zero_cnt[3]) );
  INVX2 U3 ( .A(n13), .Y(n5) );
  NAND4X2 U4 ( .A(n4), .B(n3), .C(n2), .D(n1), .Y(n12) );
  AOI21X1 U5 ( .A0(n13), .A1(n14), .B0(n12), .Y(zero_cnt[2]) );
  NAND2BX2 U6 ( .AN(n10), .B(n11), .Y(n14) );
  NOR4X2 U7 ( .A(din[10]), .B(din[11]), .C(din[8]), .D(din[9]), .Y(n13) )
  NOR4X2 U8 ( .A(din[4]), .B(din[5]), .C(din[6]), .D(din[7]), .Y(n11) );
  NOR4X2 U9 ( .A(din[0]), .B(din[1]), .C(din[2]), .D(din[3]), .Y(n10) );
  AOI31X1 U10 ( .A0(n7), .A1(n6), .A2(n19), .B0(n20), .Y(n17) );
  INVX2 U11 ( .A(din[4]), .Y(n7) );
  OR2X2 U12 ( .A(din[7]), .B(din[6]), .Y(n20) );
  OAI211XL U13 ( .A0(din[1]), .A1(din[0]), .B0(n9), .C0(n8), .Y(n19) );
  AOI2B1X1 U14 ( .A1N(din[6]), .A0(n25), .B0(din[7]), .Y(n24) );
  OAI21X1 U15 ( .A0(n26), .A1(din[4]), .B0(n6), .Y(n25) );
  AOI21X1 U16 ( .A0(n9), .A1(n27), .B0(din[3]), .Y(n26) );
  NAND2BX2 U17 ( .AN(din[1]), .B(din[0]), .Y(n27) );
  AOI31X1 U18 ( .A0(n4), .A1(n3), .A2(n15), .B0(n16), .Y(zero_cnt[1]) )
  NAND2X2 U19 ( .A(n2), .B(n1), .Y(n16) );
  OAI31X1 U20 ( .A0(n17), .A1(din[9]), .A2(din[8]), .B0(n18), .Y(n15) );
  NOR2X2 U21 ( .A(din[11]), .B(din[10]), .Y(n18) );
  AOI21X1 U22 ( .A0(n2), .A1(n21), .B0(din[15]), .Y(zero_cnt[0]) );
  OAI21X1 U23 ( .A0(n22), .A1(din[12]), .B0(n3), .Y(n21) );
  AOI2B1X1 U24 ( .A1N(din[10]), .A0(n23), .B0(din[11]), .Y(n22) );
  OAI21BX1 U25 ( .A0(n24), .A1(din[8]), .B0N(din[9]), .Y(n23) );
  INVX2 U26 ( .A(din[14]), .Y(n2) );
  INVX2 U27 ( .A(din[13]), .Y(n3) );
  INVX2 U28 ( .A(din[15]), .Y(n1) );
  INVX2 U29 ( .A(din[12]), .Y(n4) );
  INVX2 U30 ( .A(din[2]), .Y(n9) );
  INVX2 U31 ( .A(din[5]), .Y(n6) );
  INVX2 U32 ( .A(din[3]), .Y(n8) );
endmodule
```

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

```
`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
```

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-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	

Delay=0.33ns

Area=106 um x um (um^2)
~=(106/2.82) NAND2 gates
~=38 gate equivalent

LOD (using while loop)

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

module LZD_while (
            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
  i = 15;
  continue = `TRUE;
  while ((i>=0) && continue) begin
    zero_cnt = din[i] ? 4'd15 - i : 4'd0;
    continue = din[i] ? `FALSE : `TRUE;
    i=i-1;  // problematic ?
  end
end
```

```
module LZD_while ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
         n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27;

  AOI211X2 U2 ( .A0(n10), .A1(n11), .B0(n5), .C0(n12), .Y(zero_cnt[3]) );
  INVX2 U3 ( .A(n13), .Y(n5) );
  NAND4X2 U4 ( .A(n4), .B(n3), .C(n2), .D(n1), .Y(n12) );
  AOI21X1 U5 ( .A0(n13), .A1(n14), .B0(n12), .Y(zero_cnt[2]) );
  NAND2BX2 U6 ( .AN(n10), .B(n11), .Y(n14) );
  NOR4X2 U7 ( .A(din[10]), .B(din[11]), .C(din[8]), .D(din[9]), .Y(n13) );
  NOR4X2 U8 ( .A(din[4]), .B(din[5]), .C(din[6]), .D(din[7]), .Y(n11) );
  NOR4X2 U9 ( .A(din[0]), .B(din[1]), .C(din[2]), .D(din[3]), .Y(n10) );
  AOI31X1 U10 ( .A0(n7), .A1(n6), .A2(n19), .B0(n20), .Y(n17) );
  INVX2 U11 ( .A(din[4]), .Y(n7) );
  OR2X2 U12 ( .A(din[7]), .B(din[6]), .Y(n20) );
  OAI211XL U13 ( .A0(din[1]), .A1(din[0]), .B0(n9), .C0(n8), .Y(n19) );
  AOI2B1X1 U14 ( .A1N(din[6]), .A0(n25), .B0(din[7]), .Y(n24) );
  OAI21X1 U15 ( .A0(n26), .A1(din[4]), .B0(n6), .Y(n25) );
  AOI21X1 U16 ( .A0(n9), .A1(n27), .B0(din[3]), .Y(n26) );
  NAND2BX2 U17 ( .AN(din[1]), .B(din[0]), .Y(n27) );
  AOI31X1 U18 ( .A0(n4), .A1(n3), .A2(n15), .B0(n16), .Y(zero_cnt[1]) );
  NAND2X2 U19 ( .A(n2), .B(n1), .Y(n16) );
  OAI31X1 U20 ( .A0(n17), .A1(din[9]), .A2(din[8]), .B0(n18), .Y(n15) );
  NOR2X2 U21 ( .A(din[11]), .B(din[10]), .Y(n18) );
  AOI21X1 U22 ( .A0(n2), .A1(n21), .B0(din[15]), .Y(zero_cnt[0]) );
  OAI21X1 U23 ( .A0(n22), .A1(din[12]), .B0(n3), .Y(n21) );
  AOI2B1X1 U24 ( .A1N(din[10]), .A0(n23), .B0(din[11]), .Y(n22) );
  OAI21BX1 U25 ( .A0(n24), .A1(din[8]), .B0N(din[9]), .Y(n23) );
  INVX2 U26 ( .A(din[14]), .Y(n2) );
  INVX2 U27 ( .A(din[13]), .Y(n3) );
  INVX2 U28 ( .A(din[15]), .Y(n1) );
  INVX2 U29 ( .A(din[12]), .Y(n4) );
  INVX2 U30 ( .A(din[2]), .Y(n9) );
  INVX2 U31 ( .A(din[5]), .Y(n6) );
  INVX2 U32 ( .A(din[3]), .Y(n8) );
endmodule
```

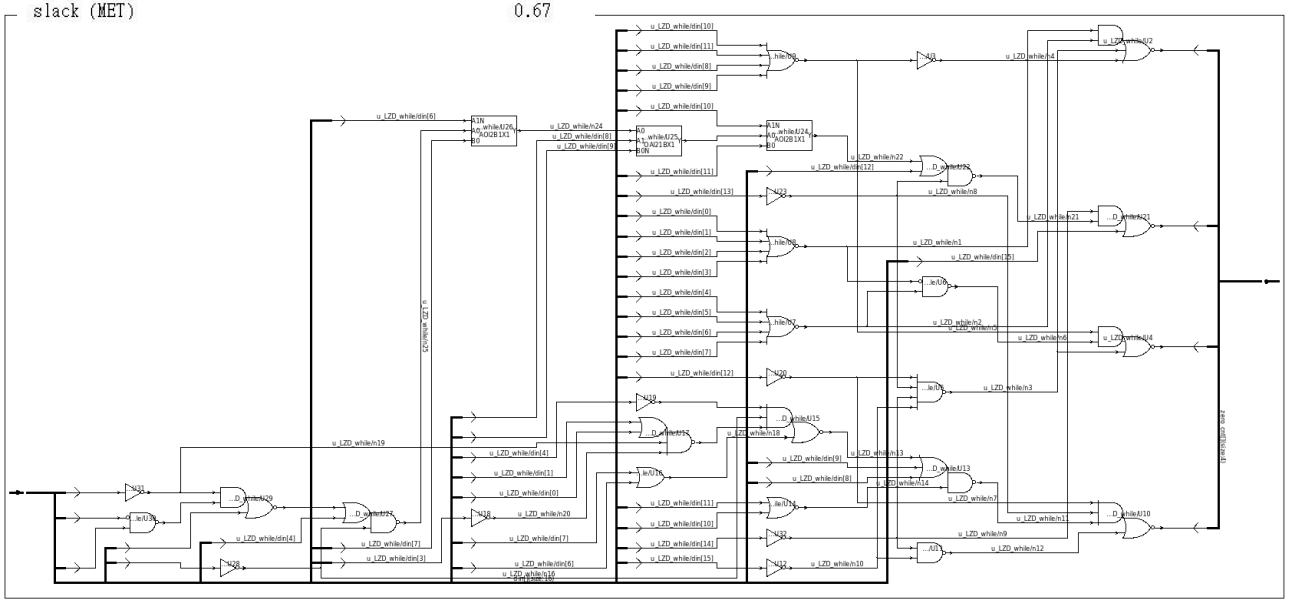
LOD (using while loop) after synthesis

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-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	

Point	Incr	Path
input external delay	0.00	0.00 f
din[1] (in)	0.00	0.00 f
U54/Y (NAND2BX2)	0.05	0.05 f
U53/Y (AOI21X1)	0.05	0.09 r
U52/Y (OAI21X1)	0.03	0.12 f
U51/Y (AOI21X1)	0.05	0.17 r
U49/Y (OAI21BX1)	0.04	0.21 f
U48/Y (AOI2B1X1)	0.05	0.26 r
U47/Y (OAI21X1)	0.03	0.29 f
U46/Y (AOI21X1)	0.04	0.33 r
zero_cnt[0] (out)	0.00	0.33 r
data arrival time		0.33
max_delay	1.00	1.00
output external delay	0.00	1.00
data required time		1.00
data required time		1.00
data arrival time		-0.33
slack (MET)		0.67

Delay=0.33ns

Area=106 um x um (um^2)
~=(106/2.82) NAND2 gates
~=38 gate equivalent



```
module LZD_while ( din, zero_cnt );
  input [15:0] din;
  output [3:0] zero_cnt;
  wire   n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
         n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27;

  AOI211X2 U2 ( .A0(n10), .A1(n11), .B0(n5), .C0(n12), .Y(zero_cnt[3]) );
  INVX2 U3 ( .A(n13), .Y(n5) );
  NAND4X2 U4 ( .A(n4), .B(n3), .C(n2), .D(n1), .Y(n12) );
  AOI21X1 U5 ( .A0(n13), .A1(n14), .B0(n12), .Y(zero_cnt[2]) );
  NAND2BX2 U6 ( .AN(n10), .B(n11), .Y(n14) );
  NOR4X2 U7 ( .A(din[10]), .B(din[11]), .C(din[8]), .D(din[9]), .Y(n13) );
  NOR4X2 U8 ( .A(din[4]), .B(din[5]), .C(din[6]), .D(din[7]), .Y(n11) );
  NOR4X2 U9 ( .A(din[0]), .B(din[1]), .C(din[2]), .D(din[3]), .Y(n10) );
  AOI31X1 U10 ( .A0(n7), .A1(n6), .A2(n19), .B0(n20), .Y(n17) );
  INVX2 U11 ( .A(din[4]), .Y(n7) );
  OR2X2 U12 ( .A(din[7]), .B(din[6]), .Y(n20) );
  OAI211XL U13 ( .A0(din[1]), .A1(din[0]), .B0(n9), .C0(n8), .Y(n19) );
  AOI2B1X1 U14 ( .A1N(din[6]), .A0(n25), .B0(din[7]), .Y(n24) );
  OAI21X1 U15 ( .A0(n26), .A1(din[4]), .B0(n6), .Y(n25) );
  AOI21X1 U16 ( .A0(n9), .A1(n27), .B0(din[3]), .Y(n26) );
  NAND2BX2 U17 ( .AN(din[1]), .B(din[0]), .Y(n27) );
  AOI31X1 U18 ( .A0(n4), .A1(n3), .A2(n15), .B0(n16), .Y(zero_cnt[1]) );
  NAND2X2 U19 ( .A(n2), .B(n1), .Y(n16) );
  OAI31X1 U20 ( .A0(n17), .A1(din[9]), .A2(din[8]), .B0(n18), .Y(n15) );
  NOR2X2 U21 ( .A(din[11]), .B(din[10]), .Y(n18) );
  AOI21X1 U22 ( .A0(n2), .A1(n21), .B0(din[15]), .Y(zero_cnt[0]) );
  OAI21X1 U23 ( .A0(n22), .A1(din[12]), .B0(n3), .Y(n21) );
  AOI2B1X1 U24 ( .A1N(din[10]), .A0(n23), .B0(din[11]), .Y(n22) );
  OAI21BX1 U25 ( .A0(n24), .A1(din[8]), .B0N(din[9]), .Y(n23) );
  INVX2 U26 ( .A(din[14]), .Y(n2) );
  INVX2 U27 ( .A(din[13]), .Y(n3) );
  INVX2 U28 ( .A(din[15]), .Y(n1) );
  INVX2 U29 ( .A(din[12]), .Y(n4) );
  INVX2 U30 ( .A(din[2]), .Y(n9) );
  INVX2 U31 ( .A(din[5]), .Y(n6) );
  INVX2 U32 ( .A(din[3]), .Y(n8) );
endmodule
```

16-bit LOD using while loop (~38 gates)

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

module LZD_while ( 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
   i = 15;
   continue = `TRUE;
   while ((i>=0) && continue) begin
      zero_cnt = din[i] ? 4'd15 - i : 4'd0;
      continue = din[i] ? `FALSE : `TRUE;
      i=i-1;  // problematic ?
   end
end

endmodule
```

Hierarchical cell	Global cell area: Absolute Total	Global cell area: Percent Total	Local cell area: Combi-national	Local cell area: Noncombi-national	Local cell area: Black-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	

Delay=0.33ns

Area=106 um x um (um^2)
~=(106/2.82) NAND2 gates
~=38 gate equivalent

Quiz

- modify 16-bit LOD so that zero_count = 16 (instead of 0) for input = 16'b0000_0000_0000_0000

```
`define TRUE 1'b1
`define FALSE 1'b0
module LZD_for (
    input [15:0]din, output reg [4:0]zero_cnt);
input [15:0] din,
output reg [4:0] zero_cnt;
reg continue;
integer i;
 always@(*) begin
   continue = `TRUE;
   for (i=15; continue && (i>=0); i=i-1) begin
      zero_cnt = din[i] ? 5'd15 - i : 5'd16;
      continue = din[i] ? `FALSE  : `TRUE;
   end
end
endmodule
```

Example: Shift Registers

x[0] <= d

x[1] <= x[0]

x[2] <= x[1]

…

x[N-2] <= x[N-3]

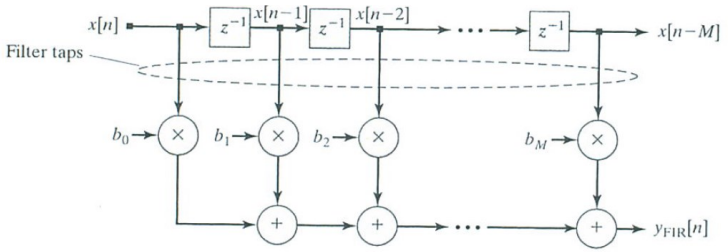
out <= x[N-2]

```
// 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
```

Example: FIR Code

- combinational logic of y=y+1 has the path from output to input, creating a feedback loop !




```
// wrong codes,
// Why?
…
reg [word_size_in-1:0] x[0:M];
reg [word_size_out-1:0] y;

y=0;
always @ (x)
   for (i=0; i<=M-1; i++)
      y = y + x[i]*b[i];
…
```

```
…
parameter bw=8, bw_out=16;
reg [bw-1:0] x[0:M-1]. b[0:M-1];
reg [bw_out-1:0] temp[0:M];
reg [bw_out-1:0] y;

always @ (posedge clk)
   x [0:M-1] <= {in, x[0:M-2]};

temp[0]=0;
always @ (x)
   for (i=0; i<=M-1; i++)
      temp[i+1] = temp[i] + x[i]*b[i];
assign  y = temp[M];
…
```

Verilog code for Saturation Adder

- clip the computation results to a fixed range

```
wire signed  [7:0] a, b, tmp, maxk min;
reg signed [7:0] sum;

assign max = 8'd127;   // max = 0111_1111
assign min = 8'd-128;   // min = 1000_0000
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
```

Example: Pipelined DP4

- see previous lectures for pure CL DP4
  - first convert I/O ports into arrays; then pipeline it

```
module DP4 (input clk, input[16*4-1:0]in_a, in_b, output reg [31:0]out);
wire [15:0] a[3:0];
wire [15:0] b[3:0];
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] pipe1_tmp[0:3], pipe2_tmp[1:4];
always @(posedge clk) begin
        for (i=0; i<=3; i=i+1)  pipe1_tmp[i] <= a[i]*b[i];
end
always @ (posedge clk) begin
        pipe2_tmp[1] = pipe1_tmp[0];
        for (i=1; i<=3; i=i+1) pipe2_tmp[i+1] <= pipe2_tmp[i] + pipe1_tmp[i];
        out<= pipe2_tmp[4];
end
endmodule
```

Example: Pipelined DPn

- parametrized long bit-vectors as I/O ports

```
module  #parameter n=4 DPn (input clk, input[16*n-1:0]in_a, in_b, output reg [31:0]out);
reg [15:0] a[n-1:0];
reg [15:0] b[n-1:0];
integer i;
always@ (*) begin
        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] pipe1_tmp[0:n-1], pipe2_tmp[1:n];
always @(posedge clk) begin
        for (i=0; i<=n-1; i=i+1) pipe1_tmp[i] <= a[i]*b[i];
end
always @ (*) begin
        pipe2_tmp[1] = pipe1_tmp[0];
        for (i=1; i<=n-1; i=i+1) pipe2_tmp[i+1] = pipe2_tmp[i] + pipe1_tmp[i];
end
always @ (posedge clk) begin
        out <= pipe2_tmp[n];
end
endmodule
```

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 Blocks (begin ... end)

- the statements are processed ***sequentially*** in the order they are specified
  - except for *non-blocking assignments* (**<=**) with intra-assignment timing control
- delay or event control is relative to the simulation time when the *previous statement* completed execution

Examples (Sequential Block)

```
// sequential block
// without delay
reg x, y;
reg [1:0] z, w;
initial
begin
    x=1'b0;
    y=1'b1;
    z={x,y};  // z=01
    w={y,x}; // w=10
end
```

```
// sequential block with delay
reg x, y;
reg [1:0] z, w;
initial
begin
        x=1'b0;   // complete at time=0
    #5  y =1'b1;  // complete at time=5
    #10 z={x,y};  // complete at time=15
    #20 w={y,x};  // complete at time=35
end
```

Parallel Blocks (fork ... join)

- statements are executed ***concurrently***
- ordering of statements is controlled by the delay or event control
- delay or event control is relative to the time the block was entered

Examples (Parallel Blocks)

```
reg x, y;      reg [1:0] z, w;
initial
fork
          x=1'b0;        // complete at time=0
     #5   y=1'b1;        // complete at time=5
     #10 z={x,y};        // complete at time=10
     #20 w={y,x};        // complete at time=20
join
```

```
// sequential block with delay
reg x, y;
reg [1:0] z, w;
initial
begin
          x=1'b0;   // complete at time=0
     #5   y =1'b1;  // complete at time=5
     #10 z={x,y};   // complete at time=15
     #20 w={y,x};   // complete at time=35
end
```

```
// race condition, values of z and w are non-deterministic
reg x, y;   reg [1:0] z, w;
initial
fork
     x=1'b0;
     y=1'b1;
     z={x,y};  // z is unknown
     w={y,x};  // w is unknown
join
```

Named Block

```
module top;

initial
begin: block1    // sequential block named block1
  integer i;       // integer i is static and local to block1,
                   // can be accessed by hierarchical name, top.block1.i
  ....
end

initial
fork: block2    // parallel block named block2
  reg  i;         // register i is static and local to block 2
                  // can be accessed by hierarchical name, top.block2.i
  ...
join

endmodule
```

Disabling Named Block

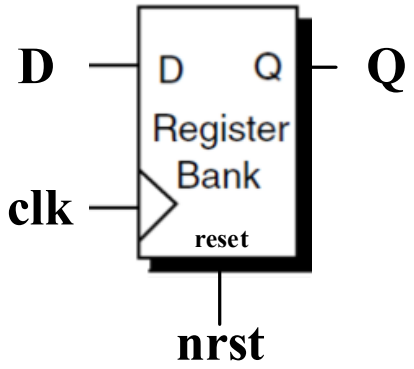
```
reg [15:0] flag;
integer i;
initial
begin
flag = 16'b 0010_0000_0000_0000;
i = 0;

begin: block1
while (i<16) begin
     if (flag[i])  begin
          $display("encountered a TRUE bit at element number %d", i);
          disable blcok1;
     end
     i=i+1;
end
end

end
```

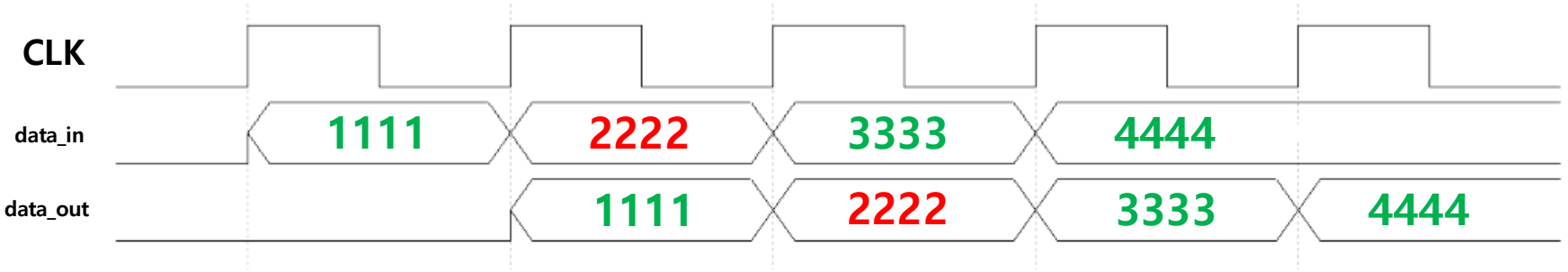
Clock Gating

Registered Output

- Regular register
  - waste dynamic power when accessing unused data

```
always@(posedge clk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else
        Q   <=  D   ;
end
```



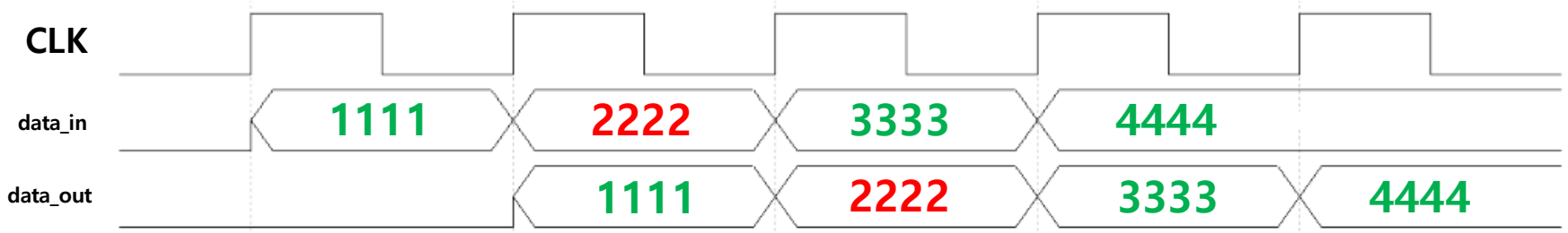


Green = Unnecessary data

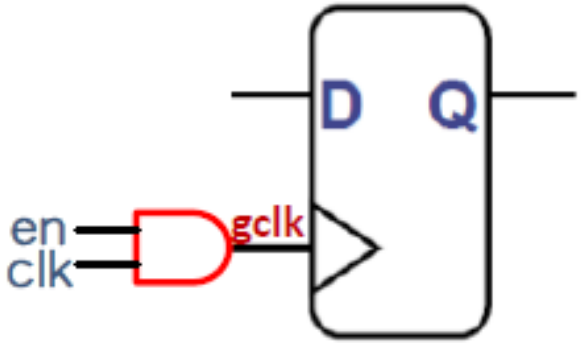
Red = Valid data

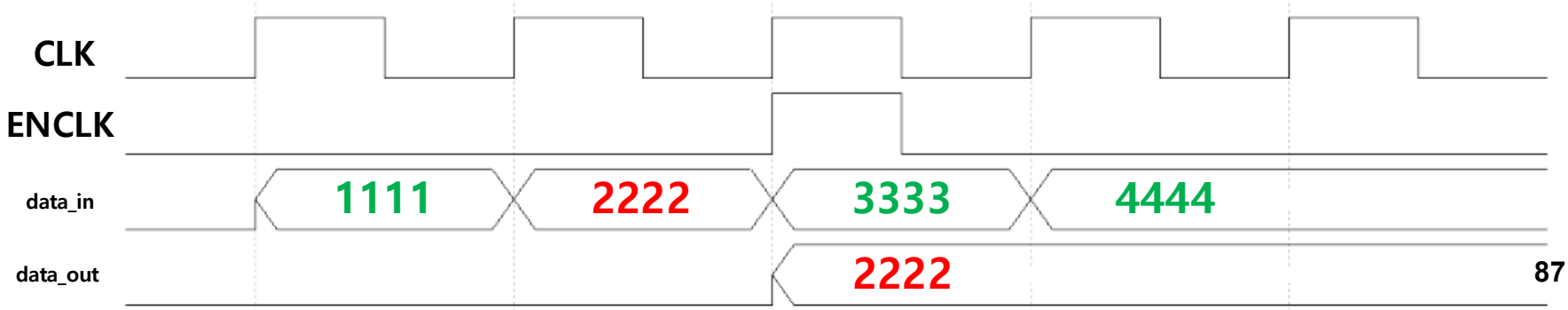
Clock gating

- Normal waveforms



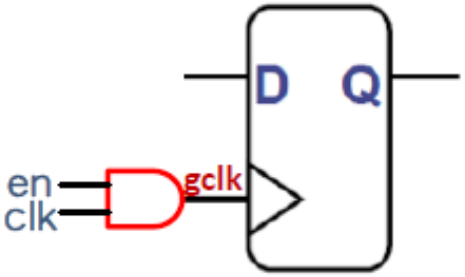
- Clock-gated waveforms
  - Controlled by ENCLK





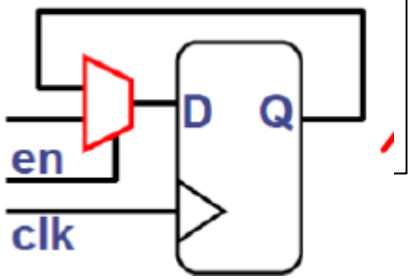
clock gating

```
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
```

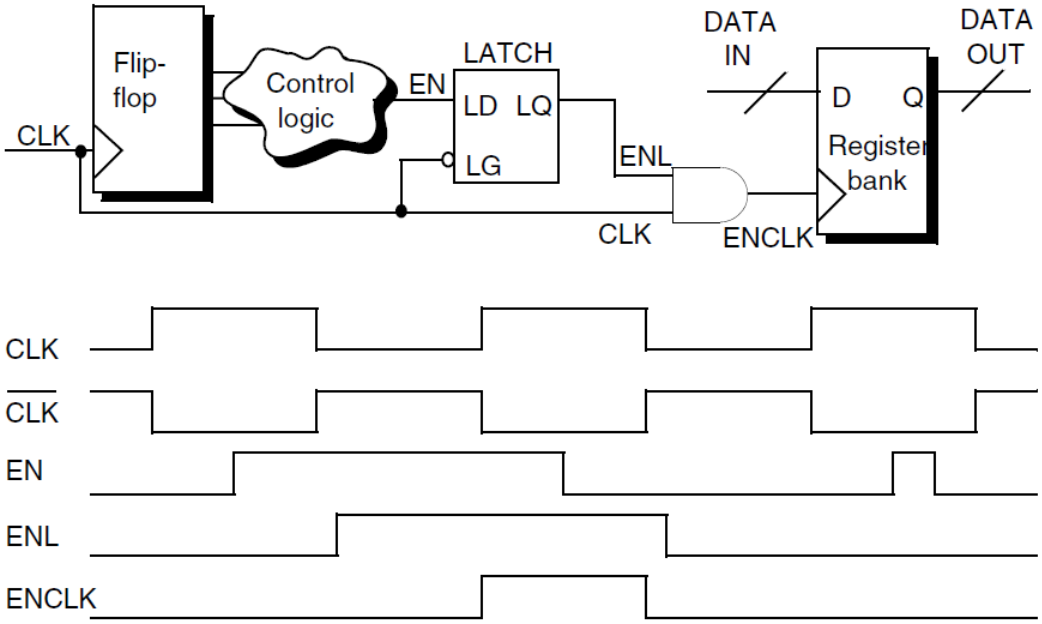


The clk port might still have switching power!
not efficiently reduce dynamic power

Latch-based clock gating (Safe Design)

- avoid glitches in clock signals
  - glitches incur unwanted signal edges
- clock-gating signal is en is latched by a latch controlled by ~clk before feeding into the AND gate
  - signal to the input of AND is stable when clk goes high

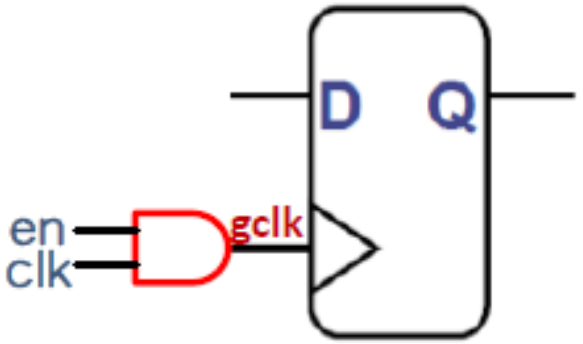
```
module safe_clock_gated_dff (Q, D, clk);
input D, clk;  output Q;
reg tmp, Q;

always @ (CLK or EN) // latch
  if (!CLK)  tmp = EN;
assign gclk = tmp1 & CLK;  // AND gate
always @(posedge gclk)  // CG FF
  Q <= D;
endmodule
```



Clock gating methods

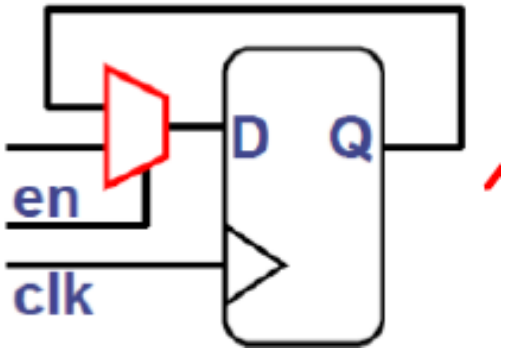
- Method 1 (unsafe gclk with possible glitches)

```
assign  gclk = clk && enable    ;

always@(posedge gclk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else
        Q   <=  D   ;
end
```



- Method2 (high switching activity of clk)

```
always@(posedge clk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else if(enable)
        Q   <=  D   ;
end
```



Synthesize lock gating with Synopsys

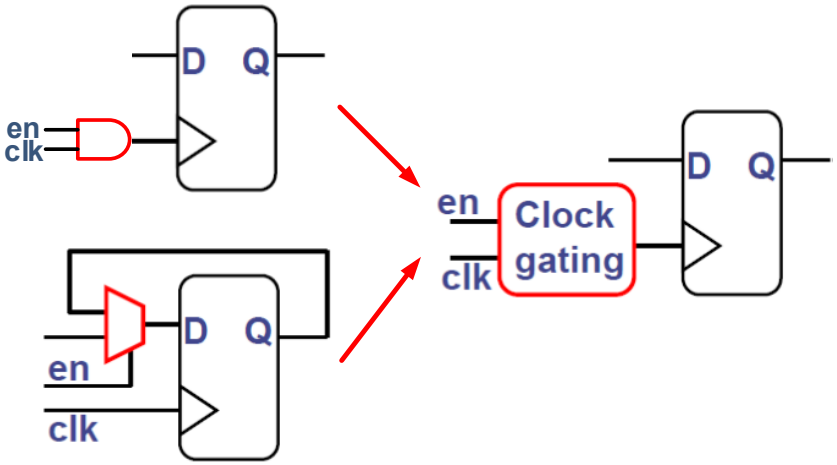
- use Synopsys synthesis script ***set_clock_gating_style*** (or other commands) which automatically find the gated clock signal and generate the safe clock-gating design

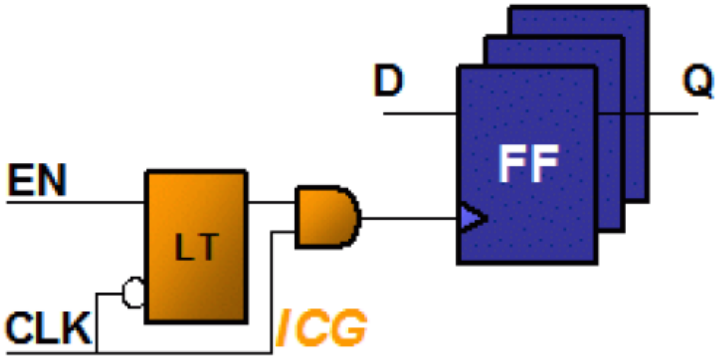
```
assign  gclk = clk && enable    ;

always@(posedge gclk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else
        Q   <=  D   ;
end
```

```
#set clock gating
set_clock_gating_style -sequential_cell latch -minimum_bitwidth 1 -max_fanout 1
propagate_constraints -gate_clock
current_design [get_designs Module_Name]
replace_clock_gates
```

```
always@(posedge clk or negedge nrst)
begin
    if(!nrst)
        Q   <=  8'd0;
    else if(enable)
        Q   <=  D   ;
end
```

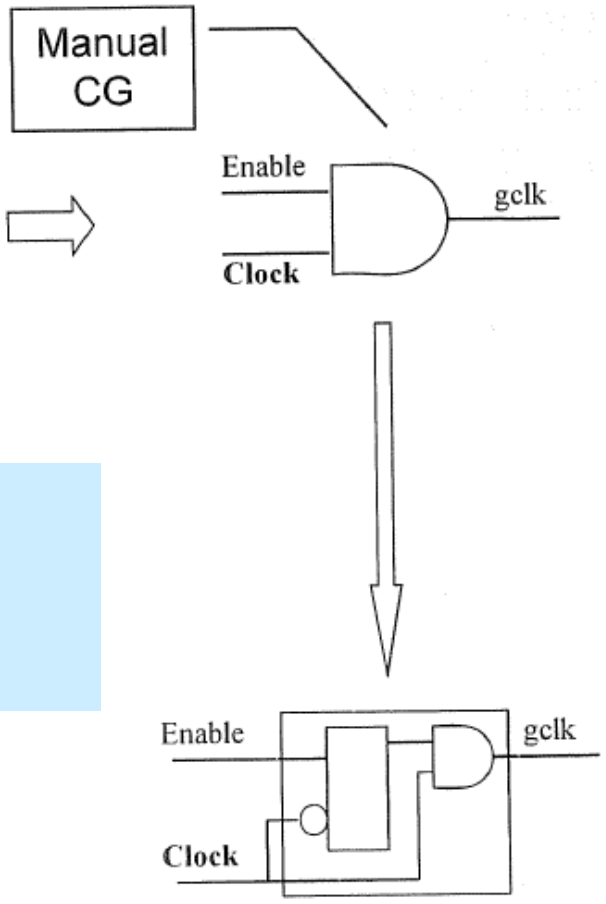
```
#set clock gating
set_clock_gating_style -sequential_cell latch -minimum_bitwidth 1 -max_fanout 1
propagate_constraints -gate_clock
current_design [get_designs Module_Name]
replace_clock_gates
```





version 1 (AND gate at clock input) (Manual Clock Gating)

```
module dff(Q, D, clk, set, rst);
input D, Clock, set, rst;
output Q;
reg gclk;
wire gclk, Enable;

// clock input is from the output of AND
assign gclk = Clock & Enable;
always @(posedge gclk)
if (rst)
   Q <= 1'b0;
else if (set)
   Q <= 1'b1;
else
   Q <= D;
endmodule
```



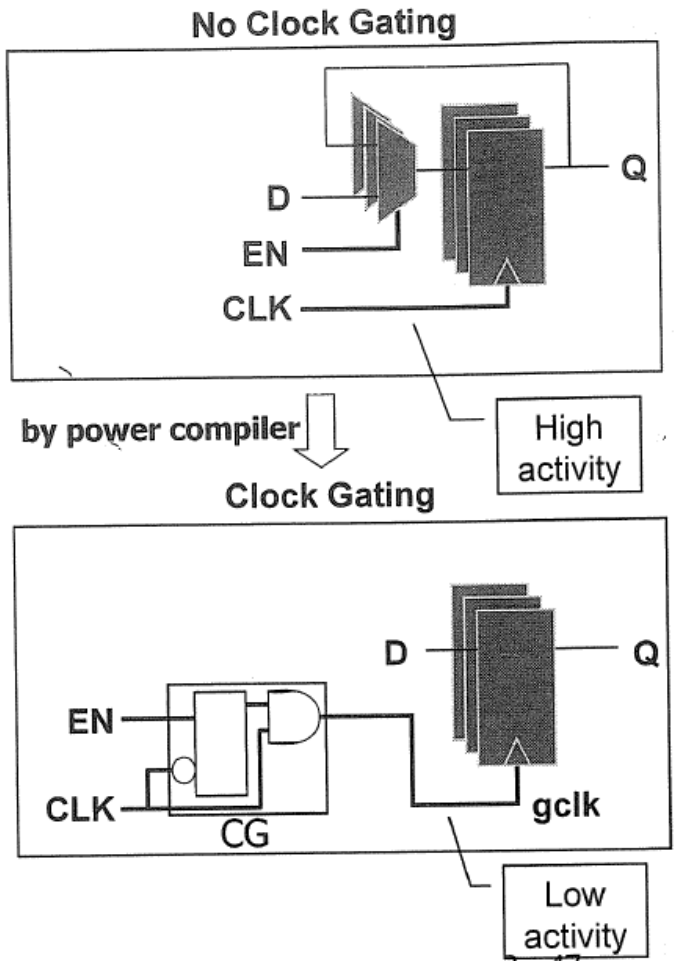
replace_clock_gate

compile

```
#set clock gating
set_clock_gating_style -sequential_cell latch -minimum_bitwidth 1 -max_fanout 16
propagate_constraints -gate_clock
current_design [get_designs Module_Name]
replace_clock_gates
```

version 2 (MUX at data input) (Auto Clock Gating)

```
module dff(Q, D, clk, set, rst);
input D, CLK, set, rst;
output Q;
reg Q;
wire EN;

// data input from MUX controlled by en
always @(posedge CLK)
if (EN) begin
   if (rst)
      Q <= 1'b0;co
   else if (set)
      Q <= 1'b1;
   else
      Q <= D;
end
endmodule
```

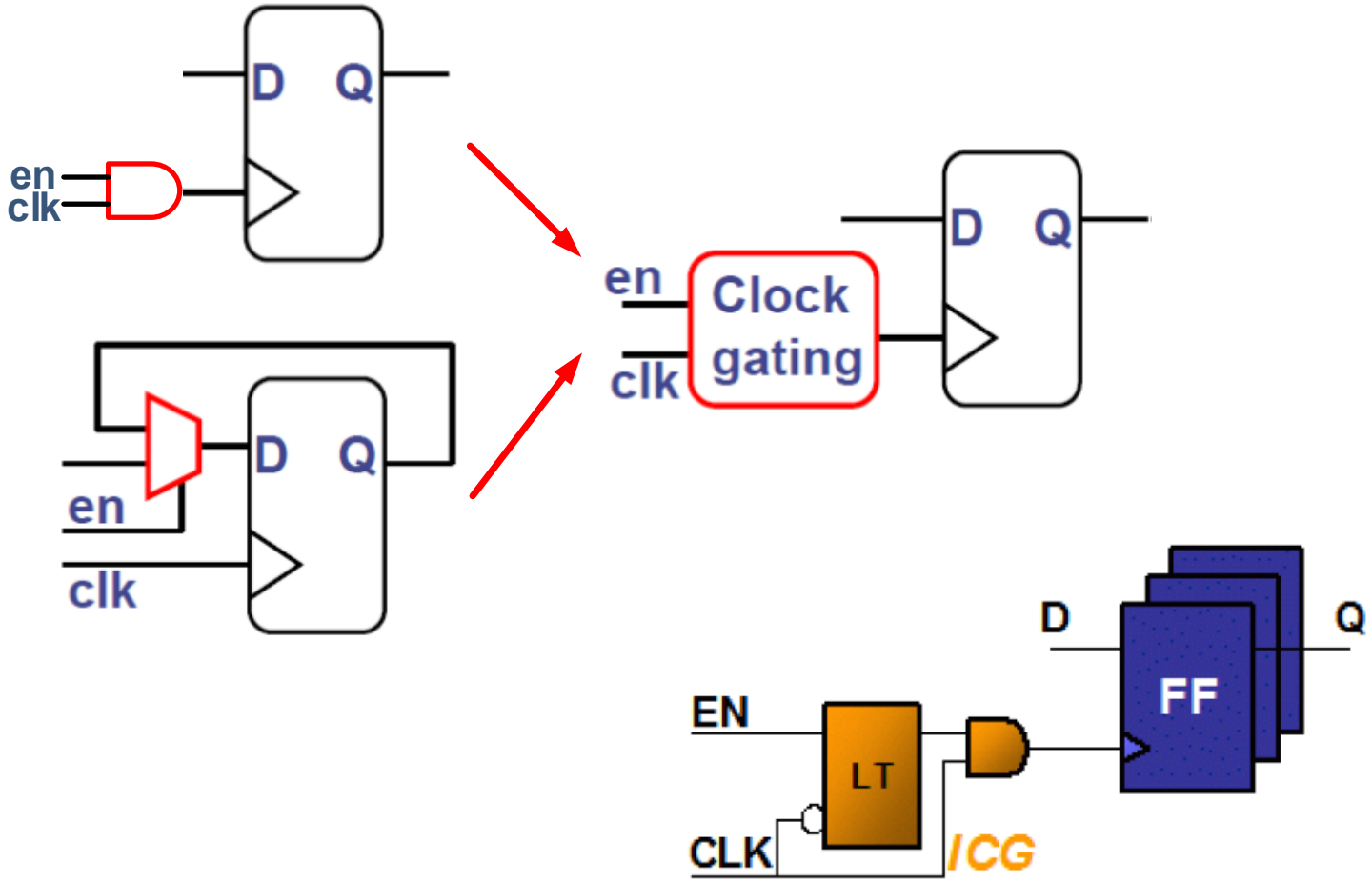


compile -gate_clock

```
#set clock gating
set_clock_gating_style -sequential_cell latch -minimum_bitwidth 1 -max_fanout 16
current_design [get_designs Module_Name]
insert_clock_gating
```

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
```