
Verilog HDL Training Course

Paul Laskowski
paulverilog@gmail.com

Overview

- HDL Background (5-13)
 - Introduction to HDL
 - Design Flow for HDL
 - Why HDL?
 - RTL mode
 - Typical Simulation Environment
- Language Basics (14-19)
 - Basic language Structure
 - Edge detection Example
 - Comments & Naming
 - Line Termination and Grouping
- Timing and Variables (20-36)
 - Logic and variable values
 - Timing variables and values
 - FORK/JOIN
 - REG Variable Type
 - INITIAL blocks
 - ALWAYS Block
 - Asynchronous RAM example 1k x 8
 - Non-blocking Procedural Assignment
 - WIRE Variable Type
 - 4bit up counter Example
- Ports and Hierarchy (37-44)
 - Module Port Declarations
 - Hierarchy
 - `DEFINE statement
 - `IFDEF, `ELSE, `ENDIF statement
 - PARAMETER statement
 - INCLUDE statement
- Logic Operations and Variables (45-54)
 - Expressions, Operators, Operands
 - WIRE & REG Operands
 - INTEGER and REAL
 - Result Operators
 - Result Operator Examples
 - Test Operators
 - Test Operator Examples
 - Order of Precedence
 - Arithmetic Examples
 - Strings
- Test, State Machines, and Tri-state (55-70)
 - Bit Error Rate generator Example
 - Conditional and Decision
 - State Machines
 - Bi-Directional/Tristate
 - Bi-Directional Example
- Looping and Control (71-75)
 - FOREVER statements
 - REPEAT
 - WHILE statement
 - FOR statement
 - WAIT statement
- Subroutines (76-77)
 - FUNCTION Statement
 - TASK statement

Overview

- Test Benches (78-96)
 - Overall Test Bench Design
 - Count4 test bench Example
 - FORCE statement
 - Typical organization
 - Clock generation
 - Cycle counter
 - Vector generation
 - Tips and techniques
- System Calls (97-110)
 - \$time
 - \$display()
 - String formats
 - \$write()
 - \$monitor()
 - \$strobe()
 - \$stop
 - \$finish
 - \$fopen
 - \$fclose
 - \$fdisplay/\$fwrite/\$fmonitor/\$fstrobe
 - \$readmemb()/readmemh()
 - \$random()
- Verilog References (111)
- Practical examples of Verilog HDL (112-124)
 - muxes
 - registers
 - counters
 - decoders
 - uP/uC interface
- Top Down/Bottom Up Design Flow (125)
 - Top Down Flow
 - Bottom Up Flow
- System Level Integration (126-130)
- Design for Reuse (131)
- Practical Coding Techniques (132-145)
 - Code layout and length
 - Signals
 - Design Hierarchy and partitioning
 - Multiple HDL statements for signal
 - Revision Control
 - Design Directory Organization
- Practical Design Techniques (146-160)
 - Synchronous design
 - Resets and clocks
 - Clock enables versus gated clocks
 - Tristate versus muxed busing
 - Inputs and Outputs
 - Interfacing asynchronous signals
 - Asynchronous sets and resets

Overview

- Logic Synthesis (161-207)
 - Introduction
 - Typical Synthesis flow
 - Synopsys, Ambit, Exemplar
 - Goals and Constraints
 - Scripts
 - Timing
 - Electrical
 - Compile Directives
 - Reports
 - Read and Write
 - Test
 - Example

Introduction to HDL

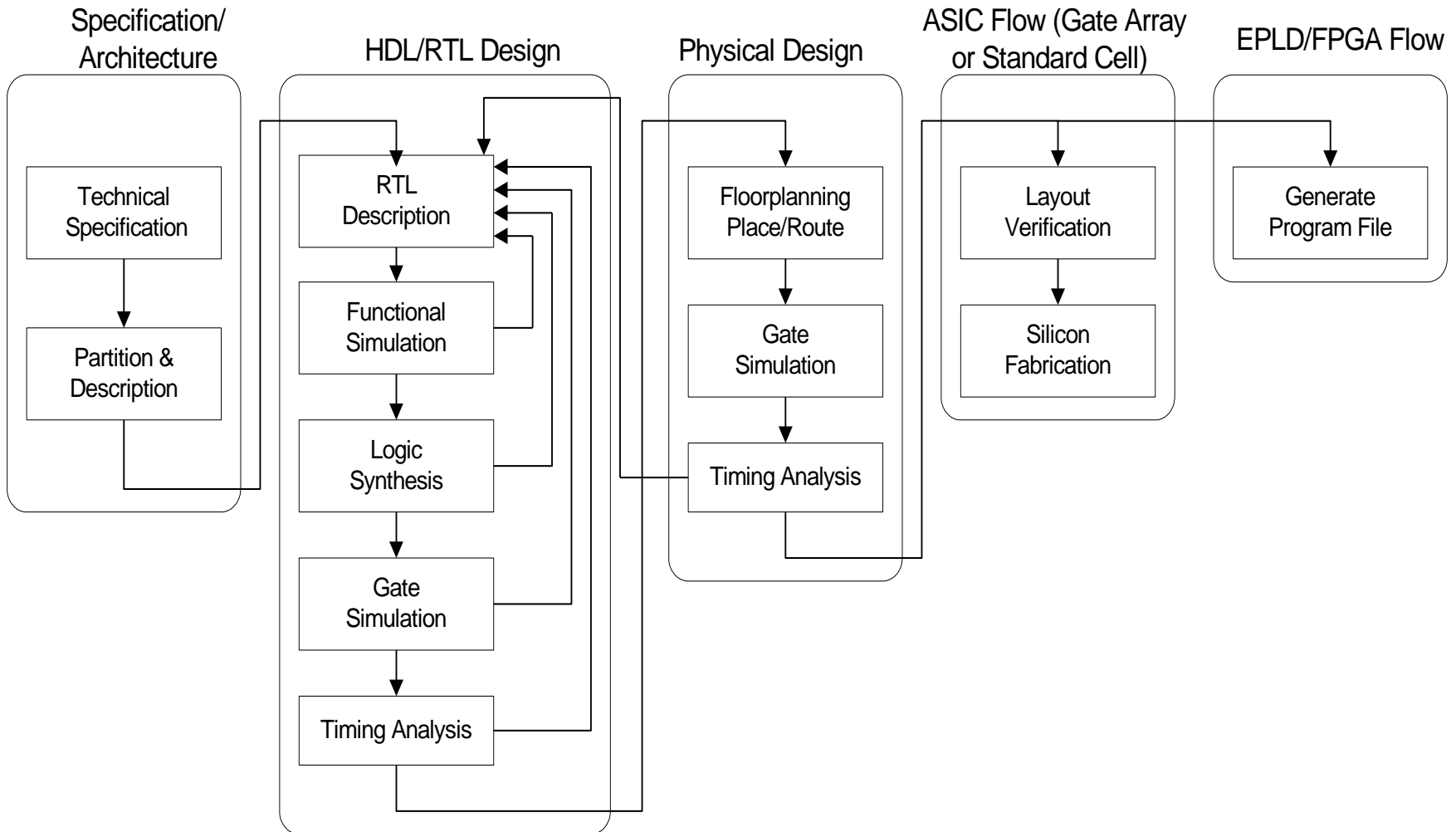
- *Hardware Description Languages* (HDL) are used to describe digital logic and algorithms in a textual language similar to programming high-level computer languages (Pascal, C, Ada, PL1, ...).
- Use a common “computer language” to efficiently describe and simulate digital hardware and provide a means to synthesize/compile into gates.
- Introduced in the 1970’s from many sources, mostly used for simulation and modeling purposes until 1980’s when logic synthesis tools allows a means to covert HDL’s into gates for a target technology (EPLD, FPGA, ASIC,...)

Introduction to HDL

- Equate “schematic capture” to programming in computer assemble language and HDL’s to programming in computer high level languages (C, Pascal, PL1, FORTRAN,...)
 - Schematic Capture is like Assembly Language in that the gates/instructions are tied to target library/computer.
 - High level design allows abstract level of design that allows designer to be more involved at system level than gate level.
 - HDL languages allows many choices of end product (FPGA/Gate Array/Standard Cell) and promotes design reuse.
- Some of the more common HDL’s
 - AHDL (A Hardware Description Language) : Univ. of AZ.
 - ABEL : Data I/O
 - VHDL (VHSIC Hardware Description Language) : DOD/IEEE
 - **Verilog** :designed in 1983 at Gateway Design (Cadence) by Phil Moorby

Design Flow for HDL

Top Down HDL/RTL Design Flow



Why HDL?

- Common textual based language
 - Similar to high level computer languages
 - Hierarchy supported
 - Same language as Test Bench (learn one language for two purposes!)
 - Efficient design reuse (i.e. IP)
 - Mixed mode simulation (Behavior - RTL - Gate - Switch)
 - Parameterized Designs
- Wide range of Abstraction
 - Behavior - Architecture level of modeling
 - **RTL** - Register Transfer Level with defined storage elements
 - Gate - Common logic elements
 - Switch - logic switches

Why HDL?

- No Schematic Capture!
 - Top Down Design, not Bottom Up Design
- Not End Technology Dependent, the designer chooses goals (speed/gate count/reprogrammable)
 - Technology Form : EPLD, FPGA, Gate Array, Standard Cell, Full Custom
 - Technology Type : CMOS, Bipolar, Bi-CMOS, GaAs
- Fast simulation in HDL mode
 - No gate timing or gate logic.
 - Majority of simulation spent verifying logical correctness not timing validation.
 - Simulation at various abstract levels

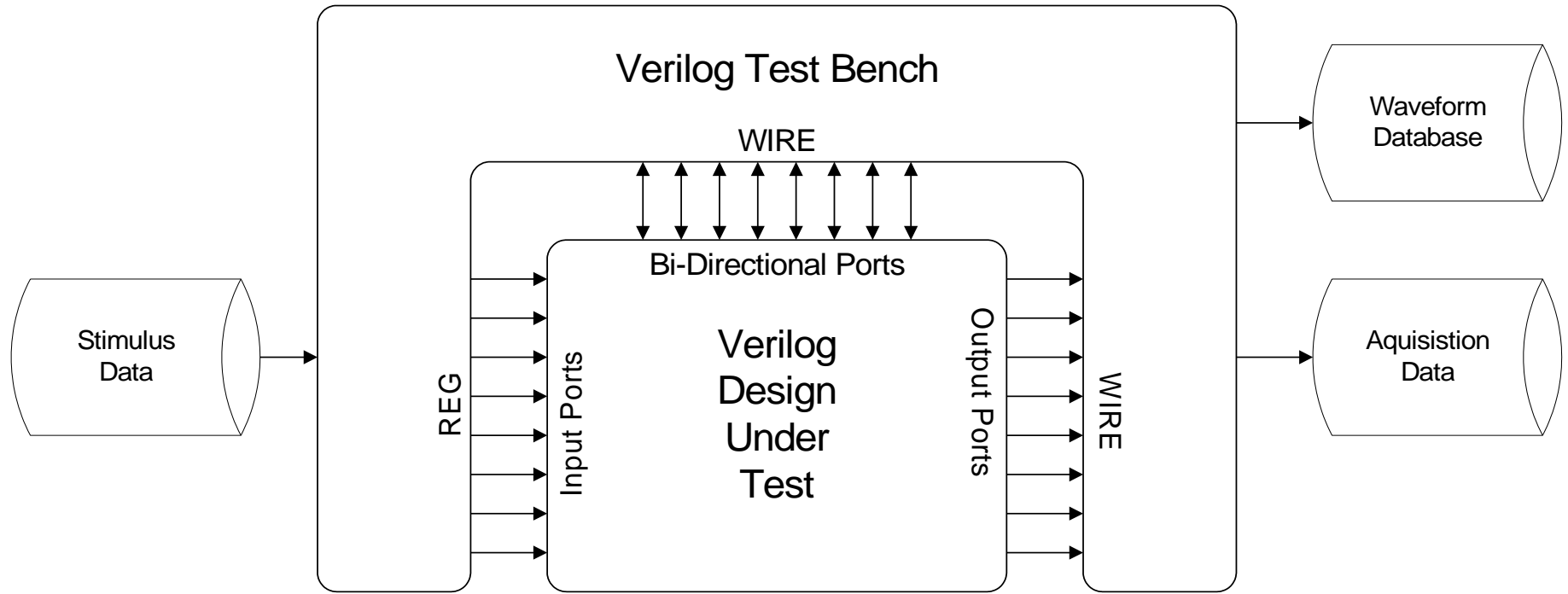
Why HDL?

- **MOST IMPORTANTLY**- the ability to use Logic Synthesis tools to translate HDL code into logic gates for chosen target library
 - Mostly automated process
 - Designer sets goals for timing (speed) and area (gate count)
- Read and Write data from files for effective simulations
 - Ability to use real world stimulus (ucode, test patterns)
- Why Verilog?
 - Very similar to “C” S/W language
 - Efficient language for compact syntax and not verbose
 - Most popular HDL in the world (note - great debate between VHDL and Verilog for this honor)
 - Adopted by IEEE as standard 1364 in year 1995. New updated version 1364-2000 in year 2000 currently in committee

RTL Mode

- RTL mode is the most common level of HDL coding for Verilog. All of the logic storage states (registers, latches, memory) are predefined. The majority of the synthesis tools support this level of abstraction. Behavior modeling is higher level of abstraction but not well supported by synthesis tools.
 - Design written in synthesizable Verilog HDL.
 - A Verilog testbench is written to simulate the design at HDL level for functional verification.
 - HDL design changes are made until design is correct and verified.
 - The design HDL code is synthesized to target technology gates.
 - Gate level design is simulated using Verilog testbench and results are compared to HDL design to re-verify functionality.

Typical Simulation Environment



Typical Simulation Environment

- The Verilog Test Bench contains all of the timing and control needed to stimulate the Verilog design.
- The Verilog Design Under Test (DUT) HDL contains no timing information and is limited to much more restricted use of the Verilog language commands/instructions.
- The Verilog Test Bench is the highest level of hierarchy in the Verilog simulation and is not synthesized.
- The Verilog simulator reads in all of the Verilog files (designs and test benches) and compiles, loads, links, and executes the resulting image file similar to a typical S/W design

Basic language Structure

```
module <name> (<port list>);  
<parameters/defines>  
<port definitions>  
<internal variables>  
<internal definitions>  
<internal functions>  
endmodule
```

Edge Detection Example

```
module edge_detect (clk, rstn, in_sig, rise_det);  
    input      clk,          // rising edge sync clock  
              rstn,         // active low async reset  
              in_sig;       // input signal to be rise edge detect  
    output     rise_det;     // rising edge detect  
  
    reg  dly1, // register delay 1  
        dly2; // register delay 2  
    wire rise_det; // rising edge detect  
  
    assign rise_det = dly1 & ~dly2; //rising edge on reg_dly  
  
    always @(posedge clk or negedge rstn)  
        if (!rstn) // if reset active  
        begin // async reset registers  
            dly1 <= 1'b0;  
            dly2 <= 1'b0;  
        end  
        else  
        begin  
            dly1 <= in_sig ; // register in_sig  
            dly2 <= dly1;    // second register of in_sig  
        end  
    endmodule
```

Port Definitions

Internal Variables

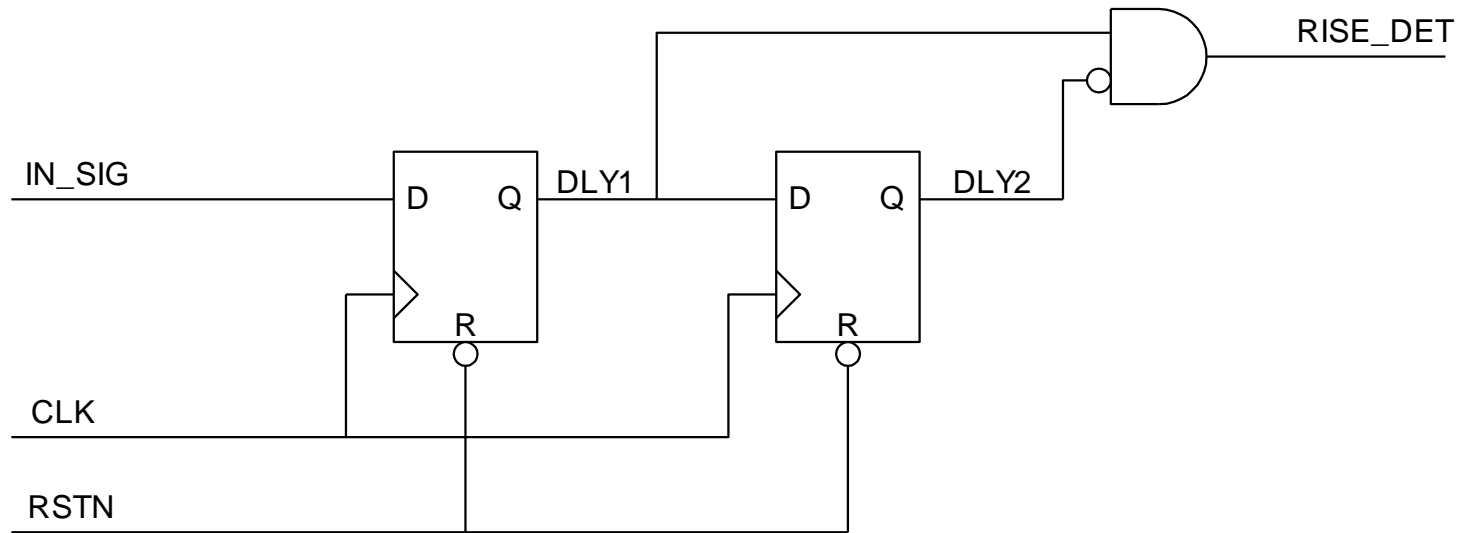
Combinational Logic

State Logic

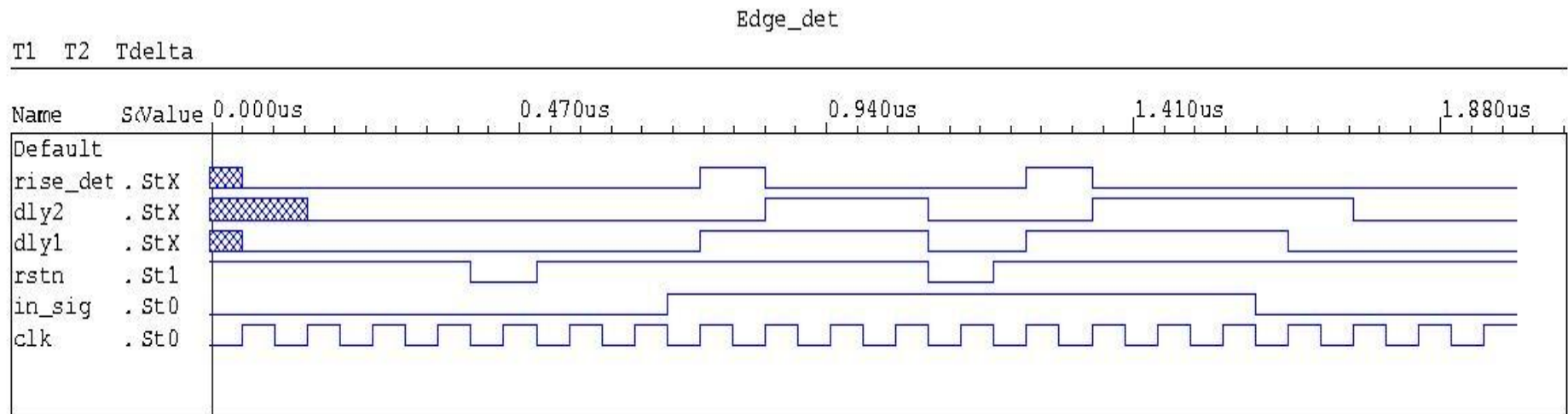
Asynchronous Logic

Synchronous Logic

Edge Detection Example



Edge Detection Example



Comments & Naming

- Comments
 - “//” - single line comment until end of line
// This is a comment until the end of the line
 - “/* */” - multi-line comments beginning with /* and ending with */
/* This is a multi-line
comment */
- Variable naming
 - First character must begin with “a-z” “A-Z” “_”
 - The remaining characters can use “a-z” “A-Z” “_” “0-9” “\$”
 - There is a list of reserved keywords in Verilog Guidelines
 - There is no practical limit to Verilog naming, but most FPGA/ASIC tools put some limit of about 16 characters
 - Upper and Lower case names (BER & ber) may get mapped to same name, so highly suggested not to use both in coding!!!

Line Termination and Grouping

- A “;” is used for line termination. Continuation or wrapping over multiple lines is allowed except for file writing (discussed later in presentation).

| | | | | |
|-------|--------|----------------|-------|--------|
| input | SIG_A, | | input | SIG_A; |
| | SIG_B, | ← equivalent → | input | SIG_B; |
| | SIG_C; | | input | SIG_C; |

- A group of statements can be put or grouped together with a “begin” and “end” statement

```
if ( SIG_A == 1'b1)
  begin
    parity_error <= 1'b1;
    ber_error <= 1'b1;
  end
```

Logic and variable values

- The logic values are 1 0 X Z
 - a Z can be overpowered by 1, 0, or X
 - A combination of 1, 0, or X results in X
- Use a “_” to separate values for better readability
- The basic logic representation is <size>’<base format><value> with leading “0” unless leftmost value is X or Z which are automatically left extended. Decimal and 32 bit size is used if no size and base format are declared
- 1'b1 = 1 bit wide using binary value of 1
 - 32'hz = 32 bit wide using hex value of high impedance on all bits
 - 8'o75 = 8 bit wide using octal values of 075
 - 11'd373 = 11 bit wide with decimal value of 373
 - 1024 = 32 bit wide using decimal value of 1024
 - 20'hX0AX = 20 bit wide using hex value of XX0AX
 - 12'b0010_1011_1001 = 12 bit value using binary value of 001010111001

Timing variables and values

- The “#” symbol is used to designate time units in Verilog. It is equivalent to *wait for this amount of time before performing this statement*.
- A “`timescale <step>/<resolution>” statement is used to indicate what the units are equivalent to in actual time.

```
`timescale 1ns/10ps    // 1ns time step with 10ps simulation accuracy
```

```
reg a, b ;
```

```
initial
```

```
begin
```

```
a <= 0; // clear a
```

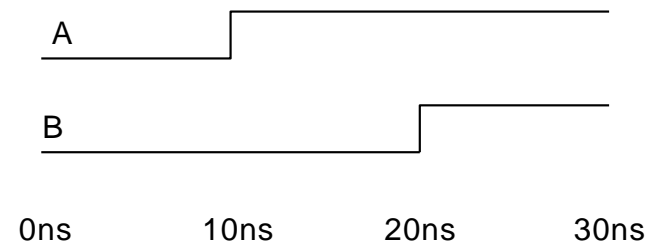
```
b <= 0; // clear b
```

```
#10 a <= 1; // wait 10ns and set a
```

```
#10 b <= 1; // wait 10ns and set b
```

```
#10; // wait 10ns
```

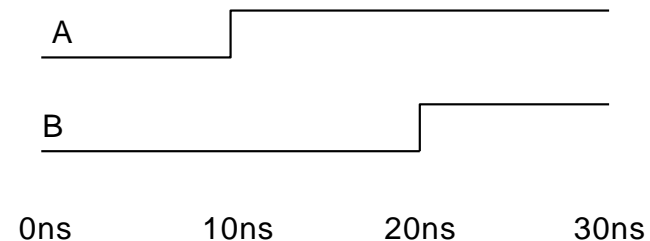
```
end
```



Timing variables and values

or use the right hand side of the assignment for setting the delay from the previous time reference

```
`timescale 1ns/10ps    // 1ns time step with 10ps simulation accuracy
reg a, b ;
initial
begin
  a <= 0; // clear a
  b <= 0; // clear b
  a <= #10 1; // set a at 10ns
  b <= #20 1; // set b at 20ns
  #30; // run for 30ns
end
```



Note: All of these assignments occur at time 0, any delay associated with a signal change will occur at “0 + delay”

FORK/JOIN

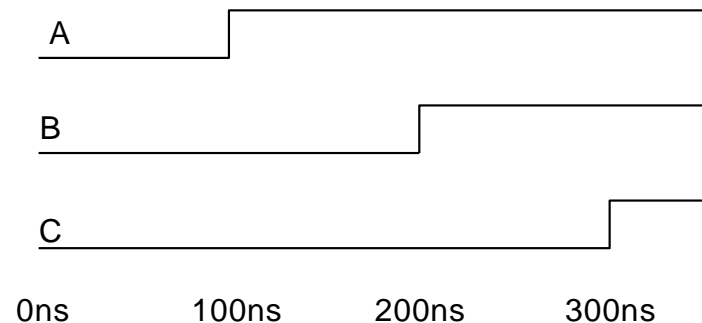
- To have statements execute once in parallel versus sequentially in time use the fork/join statement. Used inside initial blocks. *Not synthesizable*. Useful for parallel operations especially with tasks.

```
`timescale  
parameter CP=100; // 10MHz clock
```

```
initial  
begin  
#(CP) a <= 1;  
#(CP) b <= 1;  
#(CP) c <= 1;  
end
```

“=” equal execution

```
initial  
begin  
fork  
#(CP) a <= 1;  
#(2*CP) b <= 1;  
#(3*CP) c <= 1;  
join  
end
```



REG Variable Type

- REG - Represents register or data storage element. Can hold values depending on conditions. It can represent registers, latches, memory (RAM or ROM), and asynchronous combinatorial logic. REG can be scalar (single bit), vector (n-bit wide), or 2 dimensional array (RAM/ROM). Cannot be used for input port declaration. Can be used in two types of assignment blocks “initial” and “always”
 - initial - This is a procedural assignment which executes only *once*, it is only used in test benches for assigning values at predetermined times. Cannot be synthesized.
 - always - This is a procedural assignment which executes repeatedly, it is used for synthesizable Verilog design code and test benches. A sensitivity trigger list controls when the block is executed.

REG Variable Type

- REG examples

```
reg d_flipflop; // single bit register
```

```
reg [7:0] oct_reg; // 8bit register
```

```
reg [7:0] ram_8x256 [255:0]; // 8 bit wide by 256 deep RAM
```

```
reg [3:0] state_mach; // 4 bit state machine
```

```
reg [7:0] mux_out; // 8 bit multiplexer
```

```
reg [0:3] rev_check; // 4 bit register, NOTE that reg[0] is MSB and reg[3] is LSB
```

...



Left index number is most significant bit (MSB)

INITIAL blocks

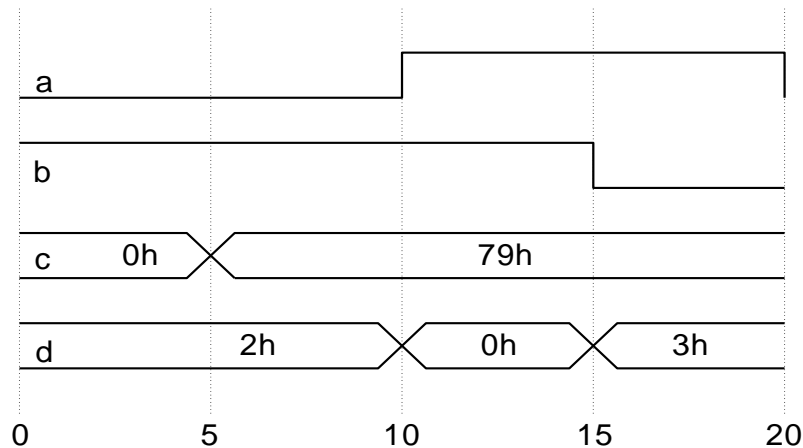
- All initial blocks start execution in parallel at “zero” time (0ns). This always a parallel programming method to have multiple signals in a test bench to be independently controlled. An initial block is only executed once in a Verilog simulation. All of the time references occur at the same time. *Only used in test benches.*

initial

```
begin
a <= 0;
b <= 1;
#10 a <= 1;
#5 b <= 0;
#5 a <= 0;
end
```

initial

```
begin
c <= 8'h0;
d <= 2'b10;
#5 c <= 8'h79;
#5 d <= 2'b0;
#5 d <= 2'b11;
end
```



ALWAYS Block

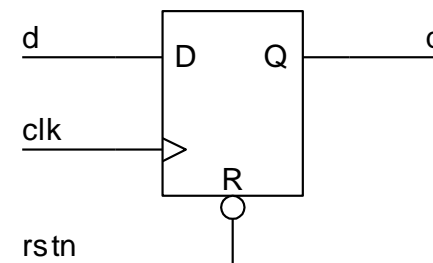
A always block contains a trigger list of signals (variables) which will cause the block to be executed whenever a signal changes it's value. A always block can be used for synchronous and asynchronous logic depending on the sensitivity list. Rising and falling edge detection is included in sensitivity list as well as either edge. *Multiple always block operate in parallel, not in sequence or order.*

```
      rising edge of "clk"      falling edge of "rstn"
      ↘                        ↘
always @(posedge clk or negedge rstn)
    if (!rstn) // active low reset
        q <= 1'b0;
    else // register d
        q <= d;
```

When "rstn" is low or false

"rstn" has priority over "clk"

D Flip-flop Example with asynchronous reset



ALWAYS Block

D Latch Example with asynchronous reset

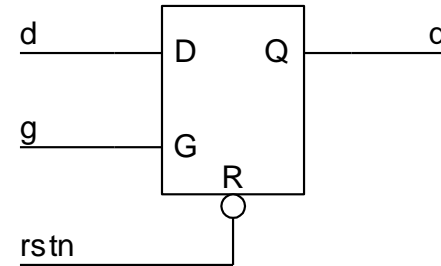
```

    whenever "g"
    always @(d or g or negedge rstn)
    if (!rstn) // active low reset
        q <= 1'b0;
    else if (g) // latch d
        q <= d;

```

Annotations for the D Latch code:

- "g" is annotated with "whenever 'g'" and "falling edge of 'rstn'".
- "!rstn" is annotated with "active low reset".
- "q <= 1'b0;" is annotated with "rstn has priority over clk".
- "g" is annotated with "transparent latch enable".
- "q <= d;" is annotated with "transparent latch enable".
- The entire block is annotated with "no need for 'else q <= q;' statement to hold value, will cause cross-coupled gates in Synopsys".



Combinational logic

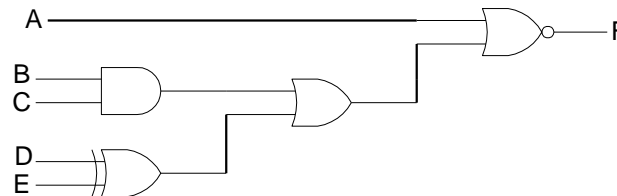
```

    always @(A or B or C or D or E)
    F = ~(A | ((B & C) | (D ^ E)));

```

Annotations for the Combinational logic code:

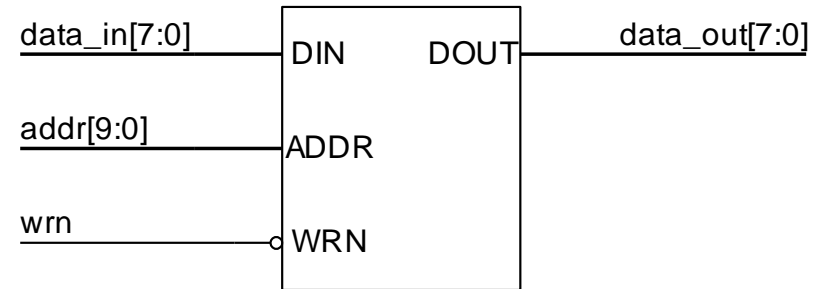
- "A" is annotated with "invert".
- "|" is annotated with "or".
- "&" is annotated with "and".
- "^" is annotated with "xor".
- The entire block is annotated with "All input signals are in the list OR ELSE A LATCH IS MADE!".



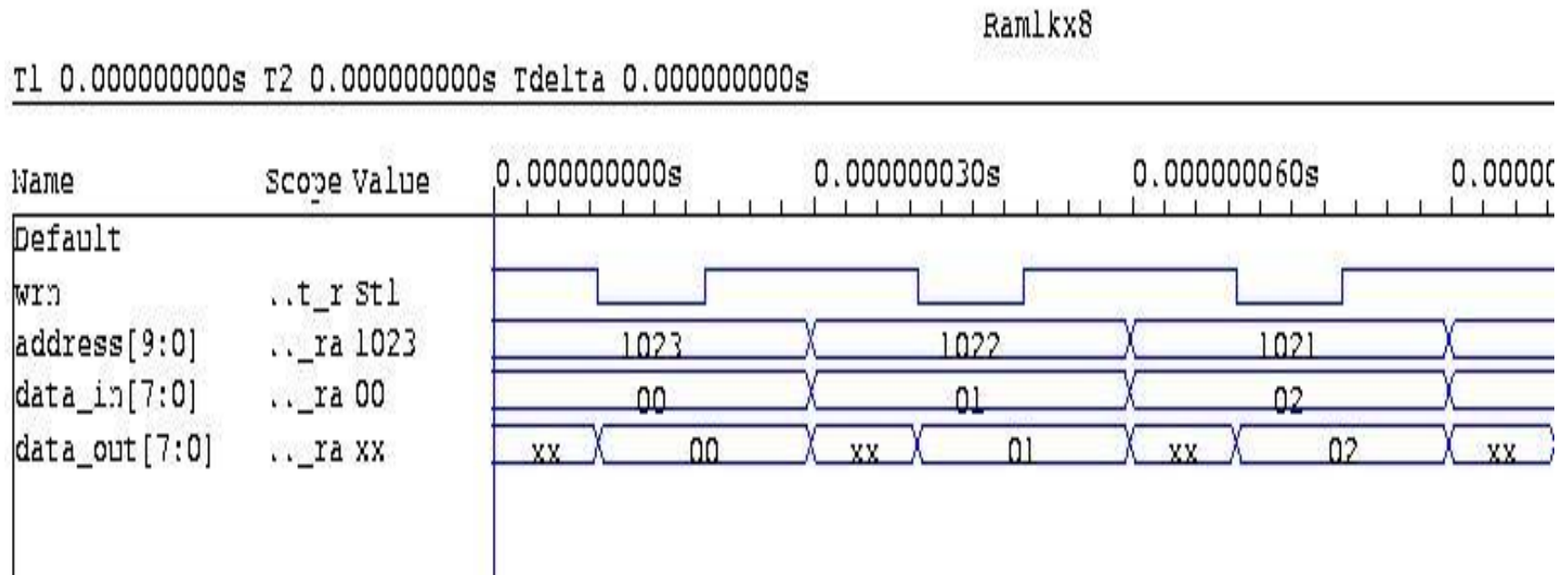
Asynchronous RAM example 1k x 8

Asynchronous RAM example 1k x 8

```
module ram1kx8(addr,data_in,wrn,data_out);
input [7:0] data_in; // 8 bit data in bus
input [9:0] addr; // 10 bit address bus
input wrn; // active low write enable
output [7:0] data_out; // 8 bit data output bus
reg [7:0] ram [1023:0] ; // two dimensional array for RAM
wire [7:0] data_out;
assign data_out <= ram[addr]; // ram data out
always @(addr or wrn or data_in) //
    if (!wrn) // active low write enable
        ram[addr] <= data_in; // ram data in
endmodule
```



Asynchronous RAM example 1k x 8



Non-blocking Procedural Assignment

Extremely Important : The non-blocking assignment “<=” is used to prevent race conditions in simulation in REG variables used for logic states (registers, latches).

Synthesizable and should be used for all assignment within synchronous ALWAYS blocks in designs. Should also be used in test benches for INITIAL and ALWAYS blocks to avoid race conditions. All values in a group of assignments are evaluated before making assignments.

Non-blocking Procedural Assignment

```
module nonblock ;  
parameter CP = 100; // 100ns clock period  
`timescale 1 ns / 1 ns  
reg clk, sig, a, b, c, d;
```

```
initial // initialize and sequence sig 010
```

```
begin  
sig <= 1'b0;  
#(4*CP) sig <= 1'b1;  
#(2*CP) sig <= 1'b0;  
#(2*CP) ;  
$stop;  
$finish;  
end
```

```
initial // clock generator
```

```
begin  
clk <= 1'b0;  
forever  
begin  
#(CP*0.50) clk <= 1'b1;  
#(CP*0.50) clk <= 1'b0;  
end  
end
```

```
always @(posedge clk) // blocking procedural assignments
```

```
begin  
a = sig;  
b = a;  
end
```

← NO!,

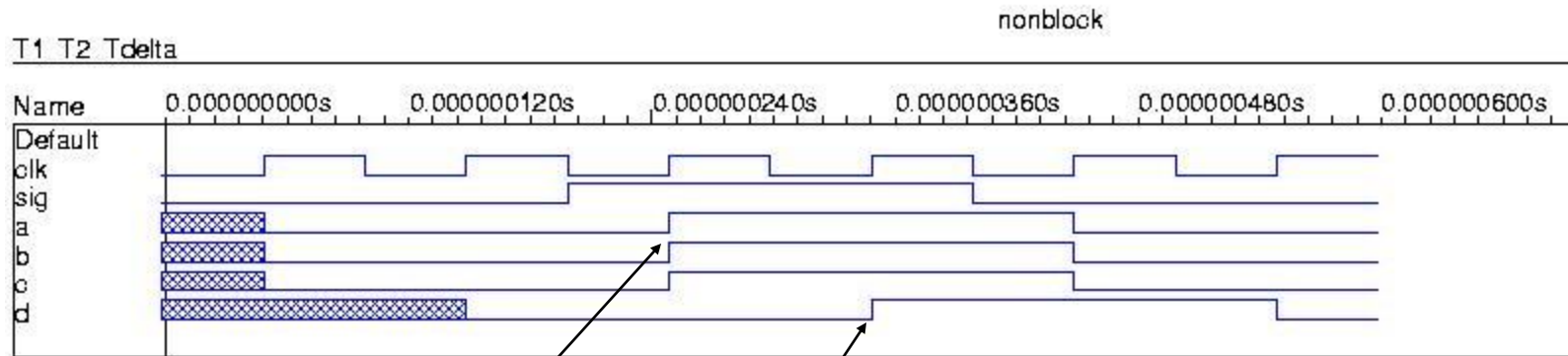
```
always @(posedge clk) // non-blocking procedural assignments
```

```
begin  
c <= sig;  
d <= c;  
end
```

← YES!

```
endmodule
```


Non-blocking Procedural Assignment



Notice how "a" and "b" change at the same time

Notice how "c" and "d" change one clock period apart correctly

WIRE Variable Type

Wire - A wire is a continuous assignment which cannot store any state. Represents a net that can be asynchronous combinatorial logic or a “wire” used to interconnect various Verilog modules through a hierarchy. All input ports into modules are wires. Wires can be scalar (single bit) or vector (bus). The “assign” statement is used to provide the continuous assignment.

```
wire cell_rdy;  
wire net_dly;  
wire [7:0] mark_str;  
wire [8:0] ccitt_cnt;
```

```
assign cell_rdy = if (a == b);  
assign #10 net_dly = cell_rdy;  
assign mark_str = counter_8 + 8'd2;  
assign ccitt[4:0] = block_cnt[4:8];
```

```
// comparing two values if true cell_rdy true  
// net_dly has a delay of 10 time units from cell_rdy
```

```
// bus bit designation only needed if not using full width
```



Note: Bit reversal

4bit up Counter Example

```
module count4 (CLK, RSTN, EN, LOAD, PRELOAD, COUNT, TCN); // 4 bit up counter
`define clear 4'd0           // counter clear state
`define increment 4'd1       // counter increment value
`define terminal 4'd15       // counter terminal count
input CLK, // synch. clock
    RSTN, // asynch. reset, active low
    EN,   // counter enable, active high
    LOAD; // counter preload control
input [3:0] PRELOAD; // counter preload value
output [3:0] COUNT; // counter output
output TCN;         // active low terminal count
wire TCN;
reg [3:0] COUNT;
assign TCN = ~(COUNT == `terminal); // terminal count on 15
always @(posedge CLK or negedge RSTN)
    if (!RSTN) // clear count
        COUNT <= `clear;
    else if (EN) // counter enable
        begin
            if (LOAD) COUNT <= PRELOAD; // preload if enabled
            else COUNT <= COUNT + `increment; // or increment
        end
endmodule
```

constant definitions

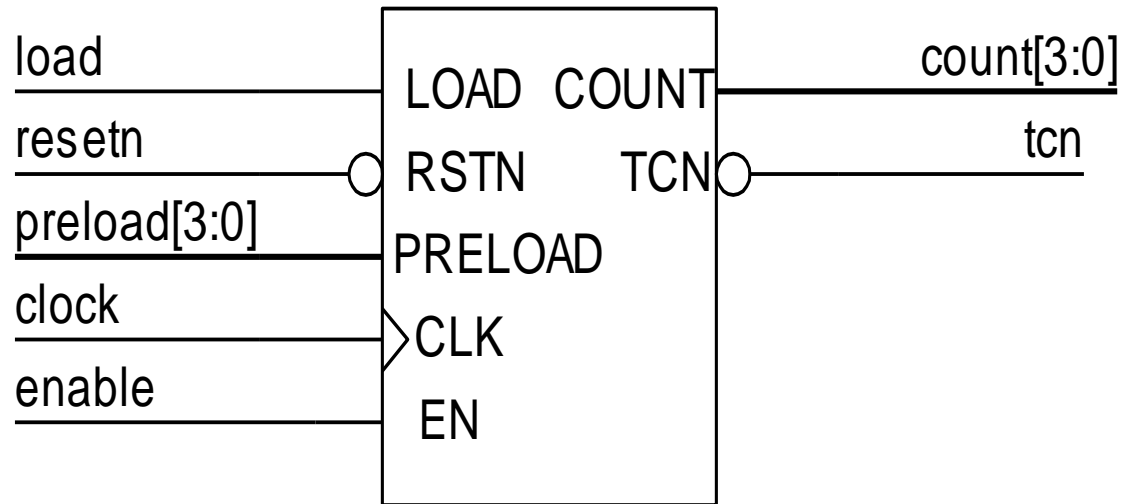
port declarations

reg/wire declarations

combinatorial logic

synchronous logic

4bit up Counter Example



Module Port Declarations

- Ports are listed in the “module” statement and declared in the port definitions.
 - There are three types of ports INPUT, OUTPUT, and INOUT (bi-directional).
 - The order in the port statement does not have to match the definition order.
 - For readability purposes, use UPPER CASE names for ports and lower case for internal regs and wires.
 - Bus port size is declared in the port definition, not in the port list.
 - Use of one port per line improves documentation.
 - The test bench does not have any port definitions.

```
module up_if (CLK, AD, DATA_IN, DATA_OUT, ADDR, CSN, WRN);  
input      CLK, // input clock  
           CSN, // active low chip select  
           WRN; // active low write enable  
input [15:0] DATA_OUT; // output data bus  
output [15:0] ADDR,      // demuxed address bus  
           DATA_IN; // demuxed data bus  
inout [15:0] AD;         // bi-directional muxed address/data bus
```

Module Port Declarations

- There are two techniques for connecting signals to ports on a module when it is instantiated. You can use the identical port list order in the original module or direct port instantiation.

```
module edge_detect (CLK, RSTN, IN_SIG, RISE_DET);  
...  
end module
```

signal name are connected due to the ordering

```
//ordered port listing  
edge_detect det_inter(clock, reset_, interrupt, dsp_inter);
```

“or”

```
//direct port instantiation  
edge_detect det_inter(.CLK(clock), .RSTN(reset_), .RISE_DET(dsp_inter), .IN_SIG(interrupt));
```

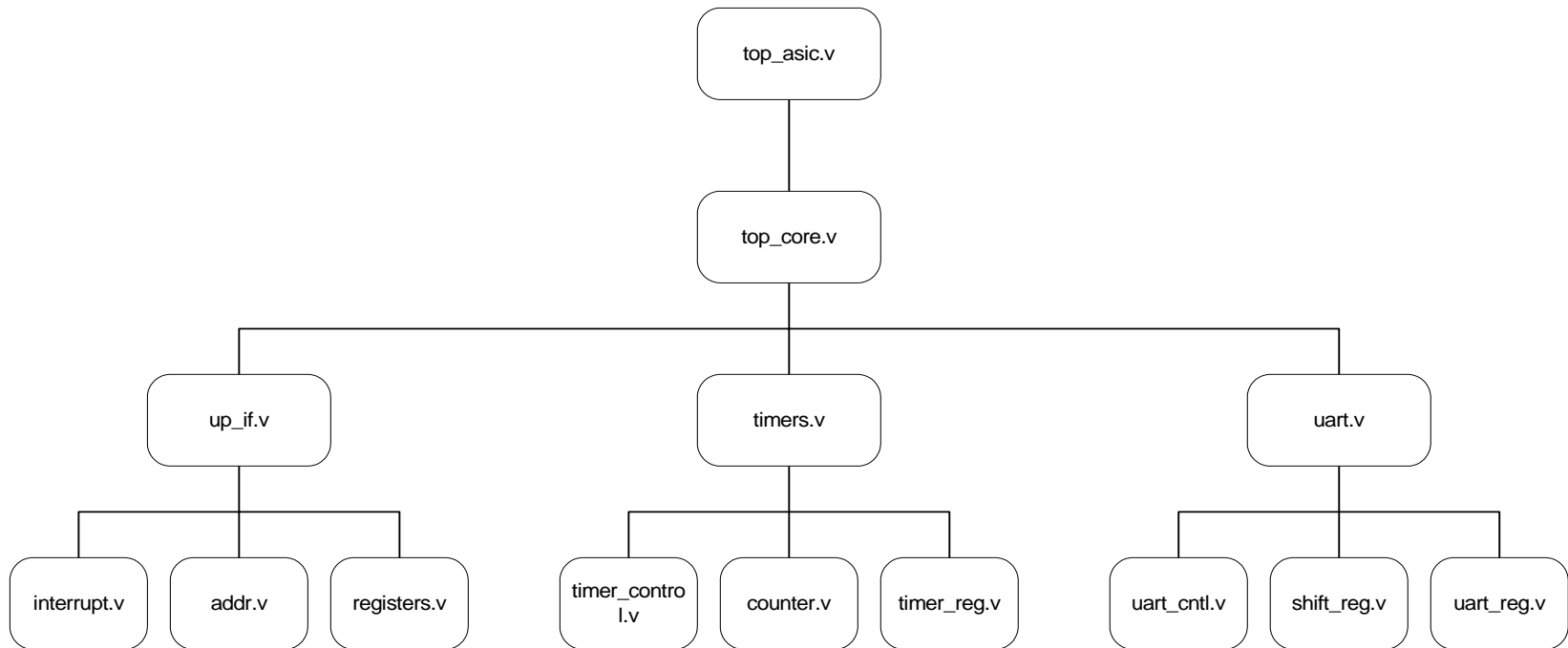
signal name in module

signal name in current level of hierarchy

signal name are connected due to the port naming,
note the different signal ordering

Hierarchy

- Hierarchy is supported by allowing a system level approach of having higher “system-level” modules call lower level modules. This allows cleaner code and promotes a building block approach to digital design. In addition design reuse is enhanced since designs can be partitioned into modules which are common in many areas of the design.



Hierarchy

- Example of how to use hierarchy. There are two techniques for connecting signals through hierarchy, “*ordered list*”, and “*name instantiation*”

```
module ber_test(CLK, RSTN, CH1_DATA, CH2_DATA, CH1_ERR, CH2_ERR, CH1_EN, CH2_EN, CH1_RUN, CH2_RUN);
...
// ordered list naming technique
ber ber_ch1(CLK,RSTN,CH1_EN,CH1_RUN,CH1_ERR,CH1_DATA);
// name instantiation
ber ber_ch2(.CLK(CLK),.RSTN(RSTN),.EN(CH1_EN),.RUN (CH1_RUN),. BITERROR (CH1_ERR),. INDATA (CH1_DATA));
...
endmodule
```

The diagram illustrates the connection of signals between two Verilog modules. The top module, `ber_test`, contains an instantiation of the `ber` module named `ber_ch2`. The bottom module, `ber`, is the component being instantiated. Arrows indicate the mapping of signals:

- `CLK` from `ber_test` to `CLK` in `ber`
- `RSTN` from `ber_test` to `RSTN` in `ber`
- `CH1_EN` from `ber_test` to `EN` in `ber`
- `CH1_RUN` from `ber_test` to `RUN` in `ber`
- `CH1_ERR` from `ber_test` to `BITERROR` in `ber`
- `CH1_DATA` from `ber_test` to `INDATA` in `ber`

```
module ber (CLK,RSTN,EN,RUN,BITERROR,INDATA); // CCITT 15 bit ber generator
...
endmodule
```


`DEFINE statement

- Text substitution can be represented by the “define” statement. Any expression can be used in a define statement. Very similar to define in “C” language. This reduces the length of the code by using common expressions for commonly used conditions. This allows Verilog code to be programmable and flexible without hardcoding values into it, only modifying a header at the top of a module or in a separate header module file. **Important:** A “define” is global across all modules so two defines with the same name will conflict.

```
`define xmit_enable (controlreg[5] & clk_en)      // xmit enable
`define clock_period #100                        // clock period = 10Mhz
...
`clock_period bb_en <= 1'b1;
...
if (`xmit_enable) send_data <= para_data;
```

`IFDEF, `ELSE, `ENDIF Statements

- To create inline conditional code compiles use the `IFDEF statement along with the `ELSE and `ENDIF. If the conditional is defined with a `DEFINE then the code up to `ENDIF or `ELSE is included in the compile. The `ELSE allows the code from `ELSE to `ENDIF to compile if the conditional is not defined. Synthesizable and extremely useful for design and test benches.

```
`define mon_sig // monitor signals

`ifdef mon_sig // if mon_sig defined then monitor signals
initial
$monitor("time = %d, clk = %b, reset = %b", $time, clk, rstn);
`else // else use clock period to track signals
initial
forever
    #CP $display("time = %d, clk = %b, reset = %b", $time, clk, rstn);
`endif
```

PARAMETER statement

- Common values and constants can be represented by the “parameter” statement. Any variable can be use an assignment from a parameter statement. Very useful for defining bus widths and parameterizing code. **Important:** “parameter”s are only local to the module declared in.

```
parameter bus_width = 32;  
parameter ram_depth = 2048;  
parameter puncture_a = 5'b10110;
```

...

```
reg [bus_width-1:0] temp_reg;           // arithmetic operations allowed for parameters  
reg [bus_width-1:0] ram [ram_depth-1:0];  
wire [4:0] punct_pat;
```

```
assign punct_pat = puncture_a; // assign 10110 binary to punct_pat
```

INCLUDE statement

- Allows a file to be included starting at the ``include` line. Can be used for designs or test benches. Very useful for concatenation of design so that all the modules are read into one file for simulation and synthesis during run and compile time for configuration management reasons.

```
`include ber.v // include ber generator
```

```
module tester(CLK,RSTN,IN_DATA,EN,ERROR,WEN,RDN,UP_DATA)
```

```
...
```

```
ber ber1(CLK,RSTN,IN_DATA,EN,ERROR); // bit error rate generator
```

```
...
```

Expressions, Operators, Operands

- The traditional software expression style is used for Verilog
<operand> <operator> <operand> or <operator> <operand>
- Results are stored on the left side of expressions
<result> = <operand> <operator> <operand> ;
- Parenthesis are used for order of precedence
(<operand> <operator> <operand>) <operator> <operand>
- Expressions can have 1, 2 , or 3 operands
- Operands can be wire, reg, integer, real, constants, and time

WIRE & REG Operands

- Used for the bulk of Verilog coding for designs and test benches
- Represents the nets (async.) and states (sync.) of a logic design.
- All values stored are unsigned, the user can use as signed with the desired representation. Verilog OVI 2.0 (Open Verilog International) now supports 2's complement numbers.
- These operands are synthesizable

INTEGER and REAL

- Integer operands are signed fixed point general purpose data types.
 - Very useful for counters and arithmetic operations
 - At least 32 bits, no standard for bit width
 - Can be used in “initial” and “always” blocks
 - Can be synthesized under certain cases (“for” loops)

```
integer cnt_a  
initial  
    cnt_a = 5;
```

- Real operands are used for floating point representation.
 - Excellent for DSP and algorithm development
 - Specified in decimal and scientific notation
 - Cannot be synthesized

```
real rcvr_pwr  
always @(l or Q)  
    rcvr_pwr = l*I + Q*Q;
```

Result Operators

- Arithmetic
 - * Multiply, $A * B$
 - / Divide, A / B
 - + Addition, $A + B$
 - Subtraction, $A - B$
 - % modulo, $A \% B$
 - Two's complement -A
- Bitwise
 - ~ Invert, $\sim A$
 - & And, $A \& B$
 - | Or, $A | B$
 - ^ Exclusive Or, $A \wedge B$
 - $\sim \wedge$ Exclusive Nor, $A \sim \wedge B$
- Assignment
 - = Assignment, $A = B$
- Reduction (useful for vector/bus)
 - & And, $\&A$
 - | Or, $|A$
 - ^ Exclusive Or, $\wedge A$
 - $\sim \wedge$ Exclusive Nor, $\sim \wedge A$
- Shift (shifted in bits are zero)
 - >> Right Shift A, B bits, $A \gg B$
 - << Left Shift A, B bits $A \ll B$
- Concatenation
 - { } Concatenation , {A, B, ...}
- Replication
 - { {} } Replication , {A{B,...}}

Result Operator Examples

A = 4'b0011; B = 4'b0101;

- Arithmetic

A * B // evaluate to 4'b1111

B / A // evaluate to 4'b0001, truncates fraction

B - A // evaluate to 4'b0010

A + B // evaluate to 4'b1000

B % A // evaluate to 4'b0010

-A // evaluate to 4'b1101

- Bitwise

~A // evaluates to 4'b1100

A & B // evaluate to 4'b0001

A | B // evaluate to 4'b0111

A ^ B // evaluate to 4'b0110

A ~^ B // evaluate to 4'b1001

- Reduction

&A // evaluate to 1'b0

|A // evaluate to 1'b1

^A // evaluate to 1'b0, good for odd parity

~^A // evaluate to 1'b1. good for even parity

- Shift

A<<2 // evaluate to 4'b1100

B>>1 // evaluate to 4'b0010

- Concatenation

{A,B} // evaluate to 8'b00110101

- Replication

{3{B}} // evaluate to 12'b010101010101

Test Operators

- Logical

! Inversion, !A

&& And, A && B

|| Or, A || B

?: Select, A ? B : C

- Equality and Identity

== Equal, A == B

!= Not Equal, A != B

>= Greater than or equal, A >= B

<= Less than or equal, A <= B

> Greater than

< Less than

These two are not synthesizable

=== Identical, A === B

!== Not Identical, A !== B

Test Operator Examples

A = 4'b0010; B = 4'b0101; C = 1'b1;

- Logical (reduces to one bit)

!A // evaluate to 1'b0

A && B // evaluate to 1'b1

A || B // evaluate to 1'b1

C ? A : B // evaluate to 4'b0010

D = 4'bx010; E = 4'bx010; F = 1'b1;

- Equality and Identity

A == B // evaluate to 1'b0

A == D // evaluate to 1'bx

A != B // evaluate to 1'b1

A != E // evaluate to 1'bx

A >= B // evaluate to 1'b0

A <= B // evaluate to 1'b1

A > B // evaluate to 1'b0

A < B // evaluate to 1'b1

A === D // evaluate to 1'b0

D === E // evaluate to 1'b1

B !== D // evaluate to 1'b0

Order of Precedence

- The precedence of operators is shown below.
 - Operators on the same line have the same precedence and associate left to right in an expression.
 - Parentheses can be used to change the precedence or clarify the situation. *Use parentheses to improve readability.*

unary operators: ! & ~& | ~| ^ ~^ + - (highest precedence)

* / %

+ -

<< >>

< <= > >=

== != === ~==

& ~& ^ ~^

| ~|

&&

||

?: (lowest precedence)

Arithmetic Examples

- Coding arithmetic operators in Verilog requires attention to bit widths for operands and results.
 - Addition

```
wire [7:0] a, b, c;  
wire carry;  
assign {carry,c} = a + b; // 9 bit result including carry bit
```
 - Multiplication

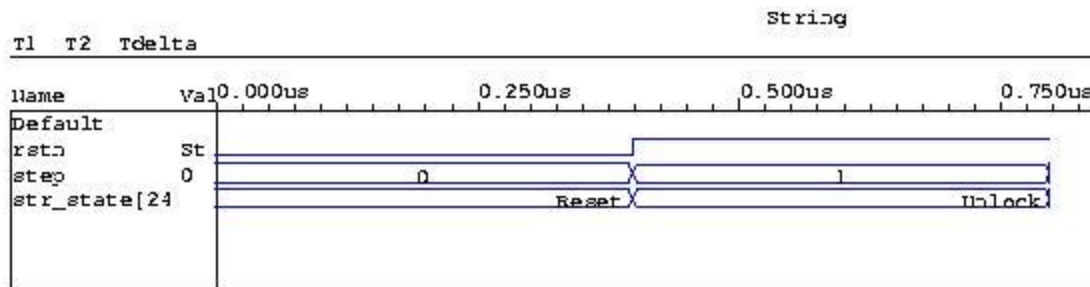
```
wire [7:0] a, b;  
wire [15:0] c;  
assign c = a * b; // 16 bit multiply result
```
- Unsigned binary arithmetic operations with two's complement supported
 - Use “-” in front of any variable to get 2's complement negation

Strings

- Strings can be made using a “array” of regs, this is extremely useful to display ASCII text messages in Verilog simulator waveform.

```
reg [8*30:1] str_state; // 30 character ASCII (8bit) string
```

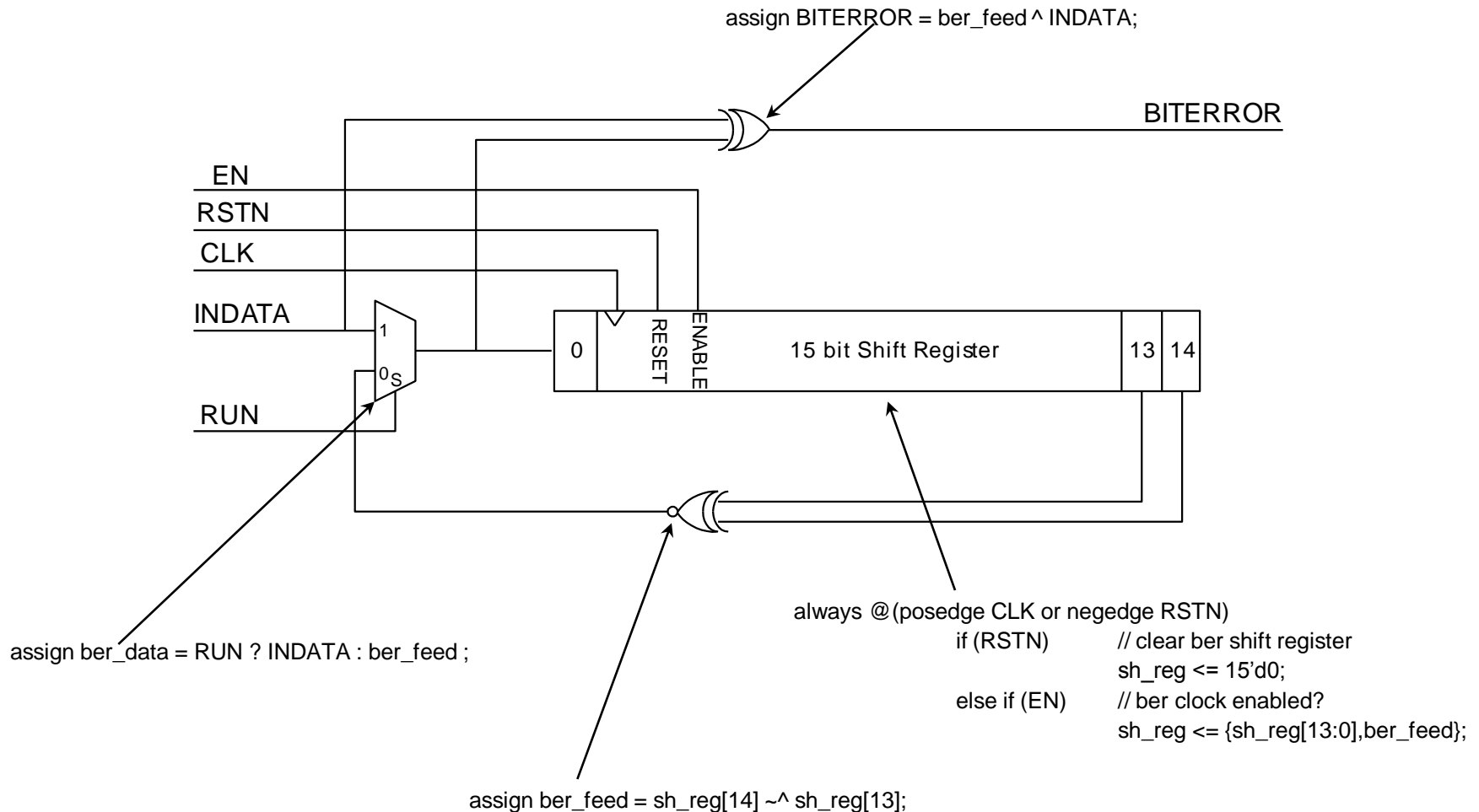
```
initial
begin
  rstn <= 1'b0;
  step <= 0; // initialize regs
  str_state <= "Reset";
  #(CP*4) rstn <= 1'b1; // release reset after 4 clocks
  step <= 1;
  str_state <= "Unlock";
```



Bit Error Rate generator Example

```
module ber (CLK,RSTN,EN,RUN,BITERROR,INDATA); // CCITT 15 bit ber generator
input CLK,                // clock input
      RSTN,               // reset active low
      EN,                 // clock enable
      RUN,                // run = 0, seed = 1
      INDATA;             // input data
output BITERROR;          // bit error
wire ber_feed,            // ber xnor feedback
      ber_data,           // ber shift register input data
      BITERROR;
reg [14:0] sh_reg;        // shift register
assign ber_feed = sh_reg[14] ~^ sh_reg[13]; // CCITT feedback standard
assign ber_data = RUN ? INDATA : ber_feed ; // active chip select and no read
assign BITERROR = ber_feed ^ INDATA; // any bit differences are a ERROR
always @(posedge CLK or negedge RSTN)
    if (!RSTN)            // clear ber shift register
        sh_reg <= 15'd0;
    else if (EN)           // ber clock enabled?
        sh_reg <= {sh_reg[13:0],ber_data}; // left shifted data register with selected data
endmodule
```

Bit Error Rate generator Example



Conditional and Decision

- Just like any high level computer language, there are several approaches for decision making: if else if else
 - *Important note* : The decision inside the parenthesis for a true is any known (no “X” or “Z”) non-zero value. So buses can be used for evaluation.

```
if (a == b)
    c <= d;
```

```
if (en && (lock || search) )
    phase_set <= `lock_state;
else
    phase_set <= `null_state;
```

```
if (!cell_frame || sar_mode)
    parity <= parity + 10'd1;
else if (mark == `cell_start)
    parity <= 10'd0;
else
    parity <= parity ^ cell;
```

Conditional and Decision

... **and : case** (note that all variables in case must be “reg” type, can be used asynchronously or synchronously). Three types of case (case, casex used for don't care conditions, casez for high impedance conditions.) statements. *Only case and casex synthesizable.*

```
case (mux_sel) // regular case decoder
    3'd0: mux_out = a;
    3'd1: mux_out = b;
    3'd2: mux_out = c;
    3'd3: mux_out = d;
    3'd4: mux_out = e;
    default: mux_out = 8'd0; //covers unused states of 3'd5 to 3'd7
endcase;
```

```
casex (addr_dec) // unknown “X” and high impedance “Z” case decoder
    8'h00: data_out = cntl_reg0;
    8'h01: data_out = cntl_reg1;
    8'b1xxxxxxx: data_out = ram[addr_dec]; // covers entire state range from 8'h80 to 8'hff
    default: //covers unused states from 8'h02 to 8'h7f
endcase;
```

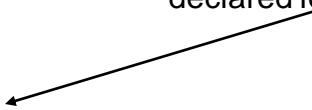
Conditional and Decision

```
casez (state) // casez decoder covers high impedance states, not synthesizable
  2'b0?: next_state = s0; //matches if state = 2'b00, 2'b01, 2'b0z
  2'd10: next_state = s1;
  2'd11: next_state = s2;
endcase;
```

- **Very important** : Using case statements for asynchronous logic requires all outputs to be listed on every case decode statement and default statement (if applicable). *If not logic states will be assumed and unintended latches inferred.*

```
reg [7:0] reg_a, reg_b, accum
reg flag;
case (opcode_decode) // regular case decoder accum and flag
  3'd0: begin accum = reg_a + reg_b; flag = reg_a[7]&reg_b[7]; end
  3'd1: begin accum = reg_a - reg_b; flag = reg_a[7]^reg_b[7]; end
  3'd2: begin accum = reg_a & reg_b; flag = accum&; end
  3'd3: begin accum = reg_a | reg_b; flag = accum|; end
endcase;
```

accum and flag
declared for every decode



Alarm State Machine Example

- Design a Verilog module for the following simple automobile alarm specification.
 - The alarm has the following inputs and outputs
 - clk (input) : 1 Hz clock, rising edge
 - rstn (input) : reset signal, active low
 - key_lock (input) : key lock signal, active high indicates key locking door
 - key_unlock (input) : key unlock signal, active high indicates key unlocking door
 - trip (input) : break-in signal, active high when a intrusion into the vehicle occurs
 - lights (output) : enable signal for vehicle lights, active high
 - horn (output) : enable signal for horn, active high
 - lock (output) : enable for controlling door locks, active high
 - Assume all inputs synchronous to “clk” input.

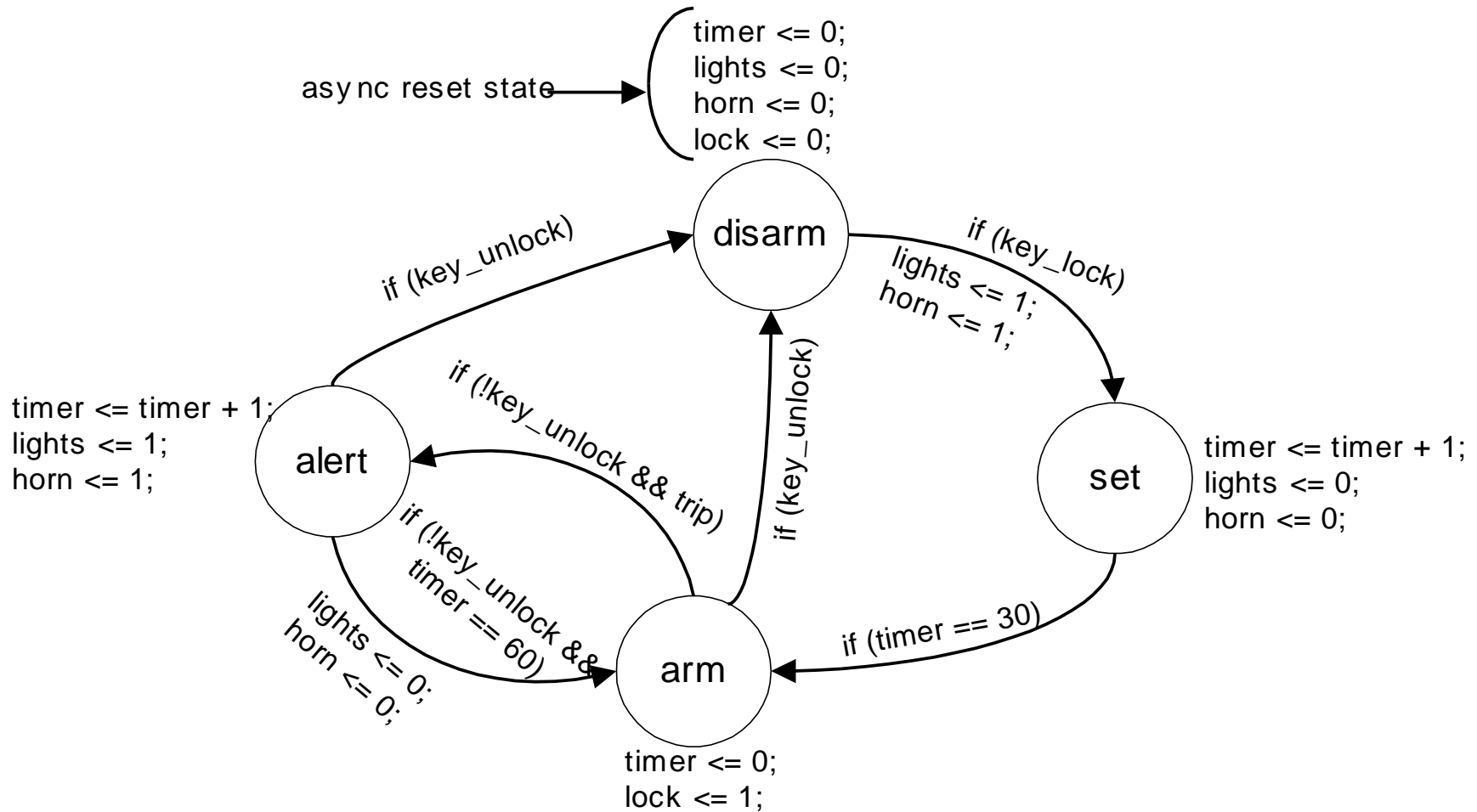
Alarm State Machine Example

- The alarm will operate in the following modes
 - **DISARM** : alarm off mode where the doors are unlocked, and the reset state when “rstn” active. Whenever a “key_unlock” goes active in the ARM or ALERT mode transition to this state. “lights” and “horn” both off.
 - **SET** : alarm set mode when the alarm is activated. When in the DISARM mode transition to SET whenever a “key_lock” goes active. During the transition from DISARM to SET turn on the “lights” and “horn” for one second to indicate the alarm is being enabled. A 30 second delay timer is activated once in this mode to enable the user to still enter the vehicle if needed before the ARM mode is activated
 - **ARM** : alarm active mode, after waiting in SET mode for 30 seconds transition to the ARM state and turn on “lock” (lock the vehicle doors). The alarm will transition back to DISARM if “key_unlock” goes active, if not and a “trip” occurs indicated a intrusion to the vehicle go to the ALERT mode.
 - **ALERT** : alarm alert mode where the “lights” and “horn” are turned on. In addition a 60 second timer is activated. If a “key_unlock” goes active transition back to DISARM mode. If the timer reaches 60 seconds then transition back to the ARM mode and turn the “lights” and “horn” off

Alarm State Machine Example

- From modes specified the following registers will be required to get the desired states
 - mode state machine : 2bit register = 4 states = number of modes
 - second timer : 6 bits = 64 states \geq 60 second maximum counter
 - horn : 1 bit = on/off control for horn
 - lock: 1 bit = on/off control for lock
 - lights: 1 bit = on/off control for lights
- Design Verilog module for synchronous logic (all states and outputs are directly related to rising edge of clock “clk”)
 - Use “case” statement for state machine
 - Current state is “case”, next state is defined within statements in case decode.
 - Control timer, lights, horn, and lock from state machine

Alarm State Machine Example



Alarm State Machine Example

```
module alarm (stat_mach, lights, horn, key_lock, key_unlock, trip, clk, rstn, lock);
```

```
    input  clk,                // rising edge clock
           rstn,               // async reset active low
           key_lock, // key locking door
           key_unlock,        //key_unlocking door
           trip;              // alarm trip
```

```
    output lights, // lights control
           horn,    // horn control
           lock;    // door lock control
    output [1:0] stat_mach; // state machine
```

```
    parameter disarm = 2'd0, // alarm idle/off state
           set = 2'd1, // alarm set state
           arm = 2'd2, // alarm arm'ed state
           alert = 2'd3; // alarm intrusion alert state
```

```
    parameter on = 1'b1,
           off = 1'b0;
```

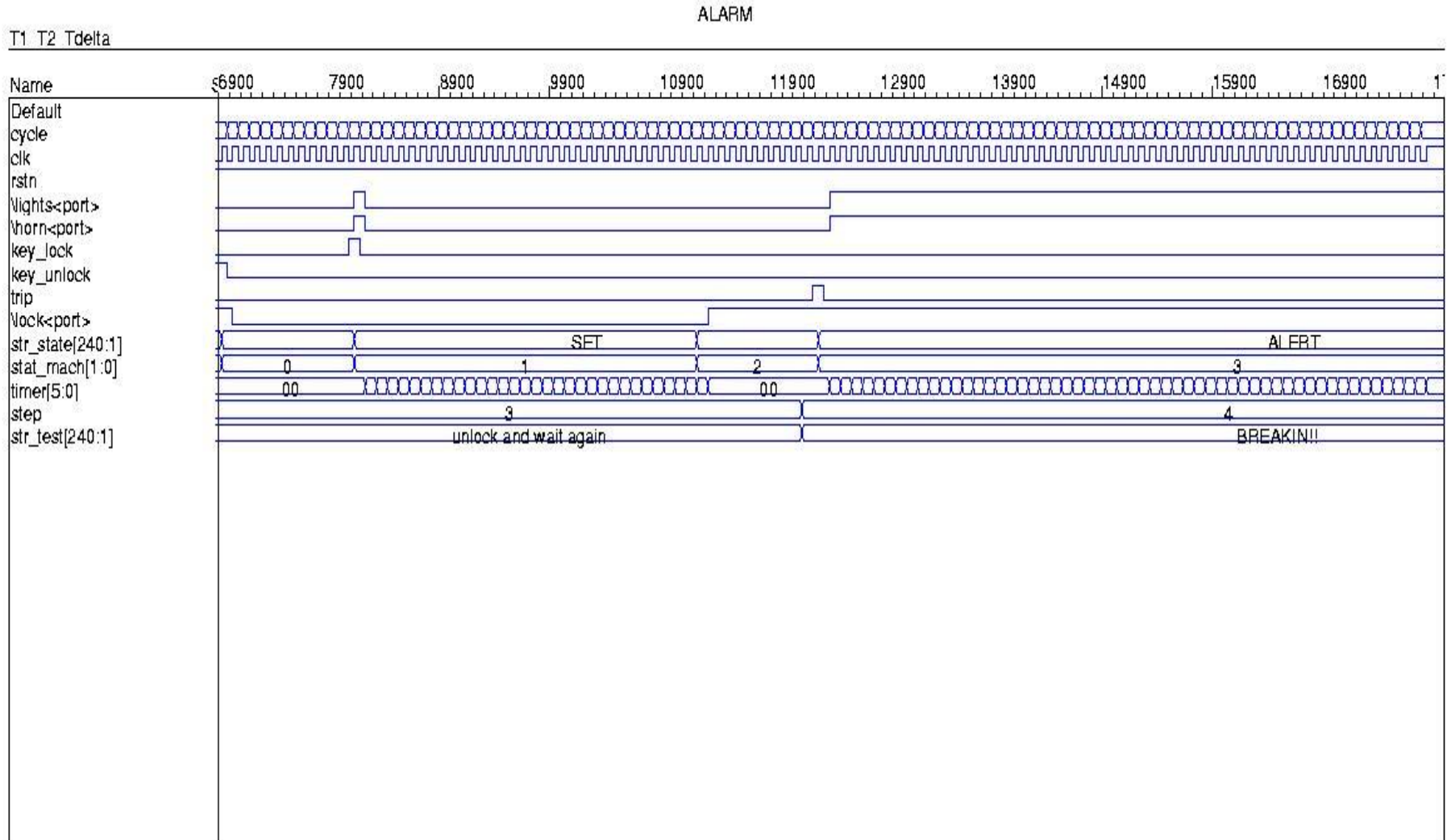
```
    reg [1:0] stat_mach; //two bit state machine
    reg [5:0] timer; // second counter
    reg lights, // lights reg
        horn, // horn reg
        lock; // door lock reg
```


Alarm State Machine Example

```
always @(posedge clk or negedge rstn)
if (!rstn)
begin
// async reset state,
stat_mach <= disarm;
lights <= off;
horn <= off;
timer <= 6'd0 ;
lock <= off;
end
else
// rising edge of clk
case (stat_mach) // case on current state of state machine
disarm: if (key_lock) // driver locks door with key
begin
stat_mach <= set ; // goto set state
lights <= on ; // turn on lights
horn <= on ; // turn on horn
end
else
begin // else stay in disarm state
lights <= off ; // lights, horn, timer, lock off
timer <= 6'd0 ;
horn <= off ;
lock <= off ;
end
set: begin // set alarm
timer <= timer + 6'd1 ; // increment counter
lights <= off ; // lights and horn off
horn <= off ;
if (timer == 6'd30) // if timer reaches 30
stat_mach <= arm ; // goto arm state
end
end
```

```
arm: if (key_unlock) // if door unlocked
stat_mach <= disarm ; // goto disarm state
else if (trip) // else if breakin
stat_mach <= alert ; // goto alert
else // else
begin
lock <= on ; // lock doors and timer cleared
timer <= 6'd0 ;
end
alert: if (key_unlock) // if door unlocked
stat_mach <= disarm ; // goto disarm
else if (timer == 6'd60) // else if timer reaches 60
begin
stat_mach <= arm ; // goto arm state
lights <= off ; // lights and horn off
horn <= off ;
end
else // else in alert state
begin
timer <= timer + 6'd1 ; // increment timer
lights <= on ; // lights and horn on
horn <= on ;
end
endcase
endmodule // alarm.v
```

Alarm State Machine Example



Bi-Directional/Tri-State

- Bi-Directional and Tri-State signals can be easily modeled using wires. A high impedance “assign” will ensure that another source will drive the wire to a known logic state. Bi-Directionals need to use the INOUT port declaration to allow modules to be able to properly send and receive signals.

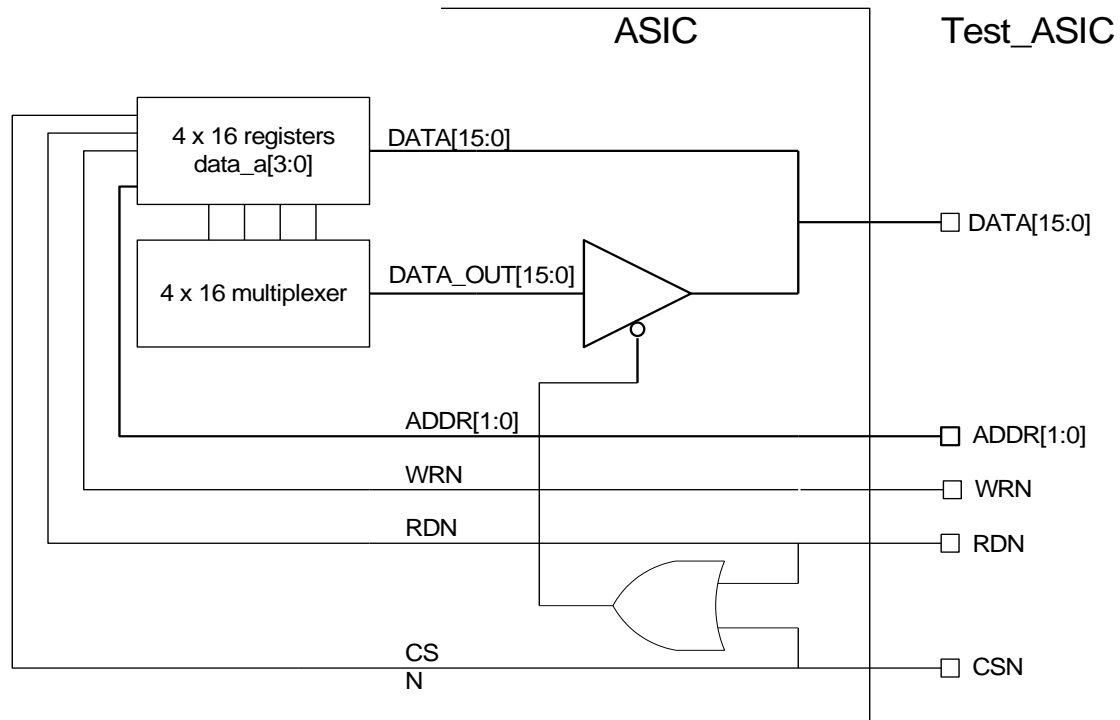
```
inout [15:0] DATA; // 16 bit bi-di data bus
wire [15:0] DATA; // 16 bit bi-di data bus
reg [15:0] DATA_OUT, // 16 bit output data bus
...
assign DATA = (RDN | CSN) ? 16'hz : DATA_OUT; // Bi-Di
```

The diagram illustrates the Verilog code for Bi-Directional and Tri-State signals. It shows three declarations: `inout [15:0] DATA;`, `wire [15:0] DATA;`, and `reg [15:0] DATA_OUT;`. The `inout` and `wire` declarations are annotated with arrows pointing to them from the text "Bi-Directional port declaration" and "Wire for Bi-Di net" respectively. The `assign` statement is annotated with arrows pointing to its components: `RDN` and `CSN` are labeled "Signal", `16'hz` is labeled "Hi-Impedance", and `DATA_OUT` is labeled "Signal". The `DATA_OUT` variable is also annotated with an arrow pointing to it from the text "Tri-state enable, active low (1=Z, 0=driven)".

Bi-Directional port declaration
Wire for Bi-Di net

Signal
Hi-Impedance
Tri-state enable, active low (1=Z, 0=driven)

Bi-Directional Example



Bi-Directional Example

```
Module ASIC (DATA, RDN, CSN, WRN, ADDR);
inout [15:0] DATA; // 16 bit bi-di data bus
input [1:0] ADDR; // 2 bit address bus
input RDN,        // read active low
      CSN,        // chip select active low
      WRN;        // write active low
wire [15:0] DATA; // 16 bit data bus
reg [15:0] DATA_OUT, // 16 bit output data bus

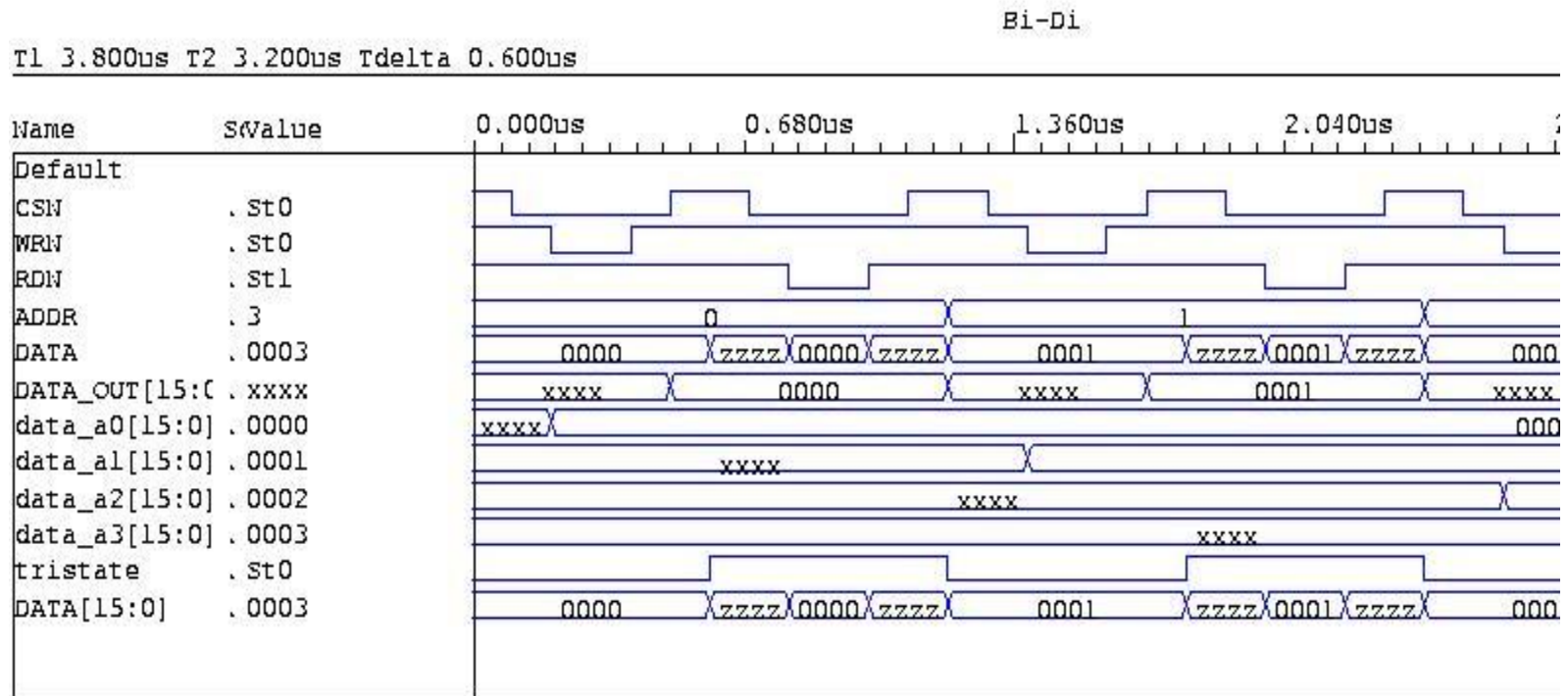
reg [15:0] data_a0, // address 0 data reg
  data_a1, // address 1 data reg
  data_a2, // address 2 data reg
  data_a3; // address 3 data reg

assign DATA = (RDN | CSN) ? 16'hz : DATA_OUT; // Bi-Di
always @(ADDR or data_a0 or data_a1 or data_a2 or data_a3) // output multiplexer
  case(ADDR)
    2'd0: DATA_OUT = data_a0; // address 0
    2'd1: DATA_OUT = data_a1; // address 1
    2'd2: DATA_OUT = data_a2; // address 2
    2'd3: DATA_OUT = data_a3; // address 3
  endcase

always@(RDN or WRN or CSN or ADDR) // 4x16 registers
  if (RDN & !WRN & !CSN) // no read, write and chip select
    case(ADDR)
      2'd0: data_a0 <= DATA; // address 0
      2'd1: data_a1 <= DATA; // address 1
      2'd2: data_a2 <= DATA; // address 2
      2'd3: data_a3 <= DATA; // address 3
    endcase

endmodule
```

Bi-Directional Example



FOREVER statement

- Repeats a statement indefinitely until simulator stops. Only used for test benches, excellent for implementing clocks and repeating sequences. **Not synthesizable**.

```
parameter CP = 100;
```

```
initial
```

```
  #(0.25*CP) forever
```

```
  begin
```

```
    clk <= 1'b1;
```

```
    #(0.50*CP) clk <= 1'b0;
```

```
    #(0.50*CP);
```

```
  end
```

clk

0ns 25ns 50ns 75ns 100ns 125ns 150ns 175ns

REPEAT statement

- Repeats a statement an integer number of times. Only used for test benches. Excellent for implementing loops that execute a determined number of times. **Not synthesizable.**

```
parameter dram_burst = 8;
always @(negedge (CSN | RDN))
begin
    i <= 0;
    repeat (dram_burst)
    begin
        data_buffer[i] <= data_bus;
        #(2*CP) i <= i + 1;
    end
end
end
```


WHILE statement

- Repeats a statement while the test condition is true. Only used for test benches. Excellent for implementing undetermined number of repeating sequences that need a test condition. **Not synthesizable.**

```
parameter dram_burst = 8;
initial
while (!fifo_empty) // fifo not empty
begin
    fifo_rdn <= 1'b0;
    #CP fifo_rdn <= 1'b1;
    data_buffer[i] <= fifo_data;
end
```

FOR statement

- Repeats a statement while the test condition is true and provides an execution and initialization statement. Can be used for synthesizable code but has generally been difficult to use. Great for providing a index integer variable in a loop.

```
parameter ram_size = 1024;
integer i;
reg [7:0] ram_data [ram_size-1:0];
initial
for (i = 0 ; i < ram_size ; i = i + 1)
begin
    ram_data(i) <= i;
    #(CP) wrn <= 1'b0;
    #(CP) wrn <= 1'b1;
    #(CP);
end
```

WAIT Statement

- Event control that “waits” for a test to become true before continuing. **Not synthesizable** only used for test benches. Great for waiting for events to occur in test bench.

```
always
wait (par_error) //parity error!
begin
$display("Parity Error!"); // display error
$stop; // stop simulator
end
```

FUNCTION Statement

- Used to create small “functions” that are used multiple times in module. Perfect for small pieces of code that return one value and do not warrant a new module or level of hierarchy. Great for use in design or test benches. There is no timing or event control.

```
function [7:0] mux4to1x8; // byte wide four to one mux
```

```
input [7:0] A; // input 0
```

```
input [7:0] B; // input 1
```

```
input [7:0] C; // input 2
```

```
input [7:0] D; // input 3
```

```
input [1:0] SEL; // mux selection
```

```
mux4to1x8 = SEL[1] ? (SEL[0] ? D : C) : (SEL[0] ? B : A);
```

```
endfunction
```

Note: function can return either single or multibit values. This example shows a byte wide return. Only one variable can be returned.

Note: The order of the input signals matches the order of the input declarations. So BER1 in the *module* connects to A in the *function*

```
wire [7:0] data_out;
```

```
...
```

```
assign data_out = mux4to1x8(BER1,BER0,PARITY,COUNT,ADDR); // example of use in module
```

```
...
```

TASK Statement

- Subroutine used in test benches for commonly used routines that are repeated multiple times in a test bench. Good to have generic “task” for interfaces such as DSP or microprocessor reads/writes. *Timing control allowed.*

```
task up_write
    input [3:0] address; // write address
    input [7:0] write_data; // write data
    begin
        ADDR <= address; // apply address and data
        DATA <= write_data;
        #(CP) CS_ <= 1'b0; // active chip select
        #(CP) WE_ <= 1'b0; // active write enable
        #(2*CP) WE_ <= 1'b1; // 2 clock period disable both
        #(CP) CS_ <= 1'b1;
        ADDR <= 4'h0; DATA <= 8'hz; // reset address bus and tristate data bus
    end
endtask

...
for (count = 0 ; count <= depth ; count = count + 1) // write address into reg
    begin
        up_write (count, count);
    end
```

Overall Test Bench Design

- A average test bench design will consist of the following structure, an example of which will follow on the following pages. You can deviate to your own preferences if desired.

```
module test()  
  define & parameters  
  reg and wire declarations  
  module(s) under test  
  initial blocks to initialize reg, open files, misc.  
  forever loops for clocks and vector generation  
  initial blocks for actual test  
  tasks  
  functions  
endmodule
```

Count4 test bench Example

```
`timescale 1ns/10ps    // 1ns time step for simulator, 10ps resolution accuracy
module test;           // top level test bench no inputs or outputs
parameter CP=100;      // clock period = 100ns
wire [3:0] count;      // 4bit count value
wire tc;               // terminal count value
reg clk, rstn, en, load; // clock, reset active low, enable, load control
reg [3:0] preload;     // preload reg
integer file1, cycle, step; // test vector file, cycle count, test #
count4 cnt4(clk, rstn, en, load, preload, count, tc); // 4 bit up counter called cnt4
initial
begin
  clk <= 1'b0; // initialize clock
  forever // loop clock
  begin
    #(CP*0.50) clk <= 1'b1; // set clock
    #(CP*0.50) clk <= 1'b0; // clear clock
  end
end
end
```

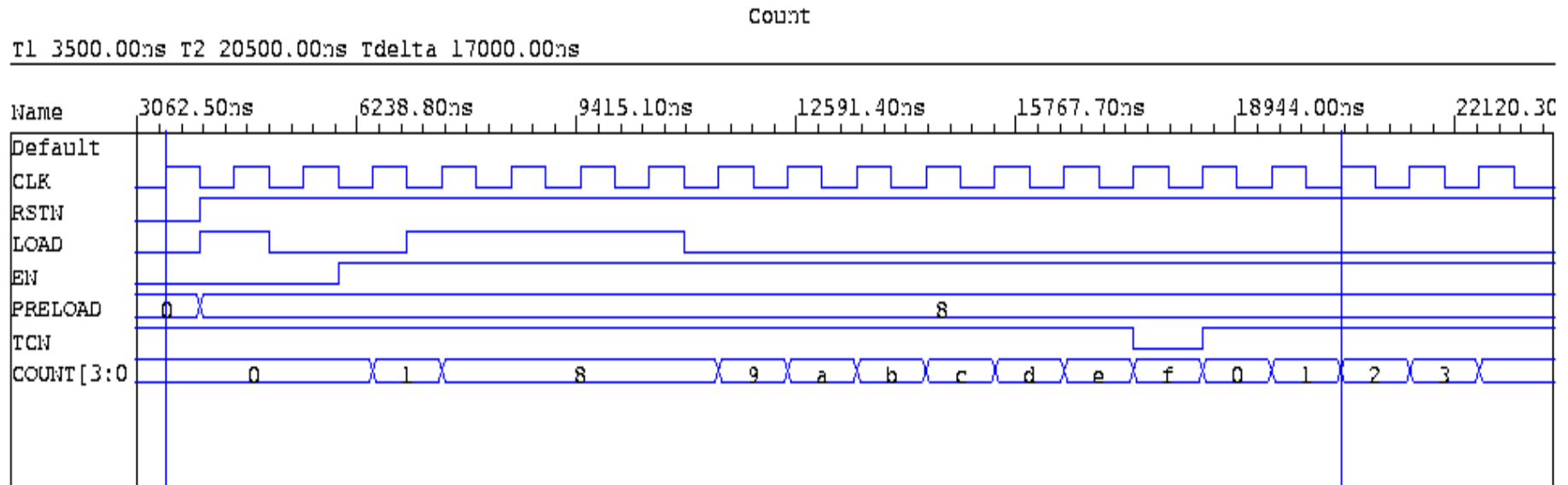
Count4 test bench Example

```
initial
begin
  step <= 0;
  rstn <= 1'b0; // initialize regs
  en <= 1'b0;
  load <= 1'b0;
  preload <= 4'd0;
  #(CP*4) rstn <= 1'b1; // release reset after 4 clocks
  step <= 1;
  preload <= 4'd8; // preload value of 8 and load
  load <= 1'b1;
  #(CP) load <= 1'b0;
  en <= 1'b1; // enable and run 100 clocks
  step <= 2;
  #(CP*100);
  en <= 1'b0; // disable and run 10 clocks
  step <= 3;
  #(CP*10);
  $stop; // halt simulator
  $finish; // exit simulator
end
```


Count4 test bench Example

```
initial // loop for cycle counter and test vector generation
begin
file1 = $fopen("count4.vec"); // open vector file named "count4.vec"
cycle = 0; // initialize cycle
forever
begin
#(CP*0.90) $fwrite(file1, "%d %b %b %b %b %b  %h %b \n", cycle, clk, rstn, en, load, preload,
count, tc); // wait 90% of clock period for signals to settle
#(CP*0.10) cycle = cycle + 1; // increment cycle at following clock period
end
end
endmodule
```

Count4 test bench Example



Count4 test bench Example

"Count4.vec"

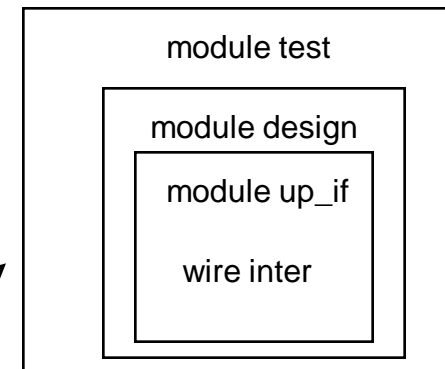
```
0 1 0 0 0 0000 0 1
1 1 0 0 0 0000 0 1
2 1 0 0 0 0000 0 1
3 1 0 0 0 0000 0 1
4 1 1 0 0 0000 0 1
5 1 1 0 0 1000 0 1
6 1 1 0 1 1000 0 1
7 1 1 0 0 1000 0 1
8 1 1 1 0 1000 1 1
9 1 1 1 1 1000 8 1
10 1 1 1 1 1000 8 1
11 1 1 1 1 1000 8 1
12 1 1 1 1 1000 8 1
13 1 1 1 0 1000 9 1
14 1 1 1 0 1000 a 1
15 1 1 1 0 1000 b 1
```

FORCE Statement

- Used to override wires and regs in design. Great for debug purposes and interactive simulation. **Not synthesizable**

```
always @(inter) // interrupt signal changing
if (inter === 1'bx)
begin
force test.design.up_if.inter = 1'b0; // force to 1'b0
$stop; // stop simulator for checking cause
release test.design.up_if.inter ;
end
```

periods used for hierarchy delimiator



Typical organization

- Typical format of testbench involves
 - Provide timescale unit
 - Select time units and resolution
 - Create the testbench module
 - Providing variables (reg, wire, integer, float,...)
 - All ports of Design Under Test (DUT)
 - reg's for inputs
 - wires for outputs
 - wires and regs for inouts
 - loop counters and index's
 - vector counters
 - file ids for files opened for writing
 - memory arrays for storing results and readmem
 - Parameter and Defines (or Include file)
 - Clock period
 - bit widths
 - Register/Memory maps
 - Create ifdef defines for test control

Typical organization

- Instantiating the DUT(s)
 - Connections of ports
- Control Tri-state and Bi-directional signals
 - Decode tristate controls from output enables, chip selects, and/or read control signals
 - Assign for Bi-Di wires
- Initiate all variables and input/BiDi signals into DUT
 - Initial blocks for setting regs
 - Always blocks for repeating or conditional reg assignment
- Open pattern and stimulus files
 - Use readmemb (binary) and readmemh (hex) for reading in stimulus files
- Open files for test vectors and waveform database
 - Vector file with always block to control time for strobing and acquisition markers
- Reset and initialize all internal states
 - Registers
 - RAM's
 - Latches

Typical organization

- Create task's for commonly used subroutines for easier coding and readability
 - uP/uC interface read/write
 - Communication ports
 - JTAG
- Create test vector pattern sets
 - Use a always block with test cycle counter for “strobed” vectors for logic verification
 - Sample every input and output only once per cycle
 - Excellent for simple compare from HDL to synthesis
 - Use a monitor statement to generate “print on change” vectors for ASIC signoff and signoff simulations
 - Sample every input and output whenever any signal every changes.
 - Records every change in any signal
 - Test vector tools (RuleAnalyzer) convert vectors into tester timing sets and pattern files.
 - Needed to check timing for tester timing sets.

Typical organization

- Run tests for DUT
 - Use initial blocks to create a sequential timeline of test events
 - Use fork/joins to run concurrently different parts of design simultaneously
 - Saves number of test vectors and tester time
 - Partition tests in series with ifdef's to allow certain sections to be skipped when debugging
- Close all files (vector and waveform) before finishing tests.

Clock generation

- Use always block or forever block to generate a continuous clock. Multiple clocks can be made, it's advisable to use harmonics of a common clock frequency to make time format simpler for generating test vectors.

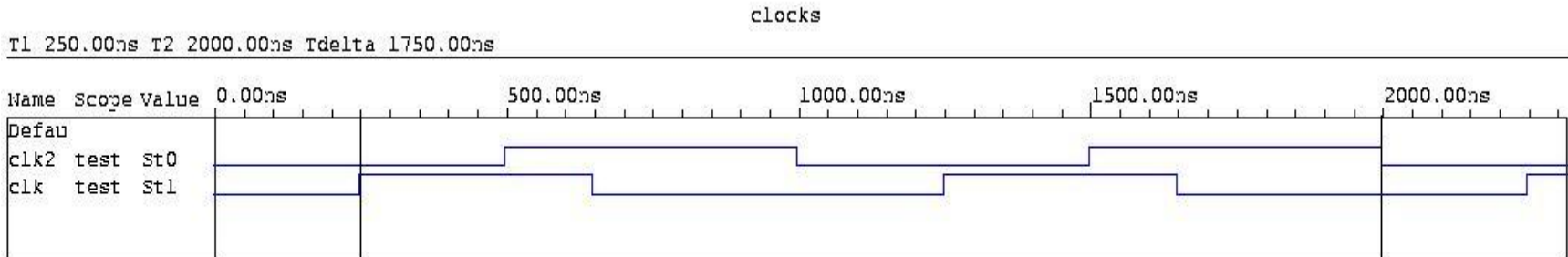
```
parameter CP=100,           // clock period = 100ns
        offset = 25,        // clock offset = 25ns
        high = 0.40;        // high period % time

reg clk, clk2;              // clock

initial // initial clock
begin
    clk <= 1'b0 ; // initialize clock
    #(offset) forever          // delay clock by offset
    begin
        clk <= 1'b1; // set clock
        #(CP*high) clk <= 1'b0; // clear clock after high time
        #(CP*(1.0-high)); // calculate low time and loop
    end
end

initial // clear clock
    clk2 <= 1'b0;
always // simple 50% duty cycle clock always block
    #(CP/2.0) clk2 <= ~clk2;
```

Clock generation



Cycle counter

- A cycle counter is useful for keeping track of number of clock cycles in simulation. Used for debug and vector generation.

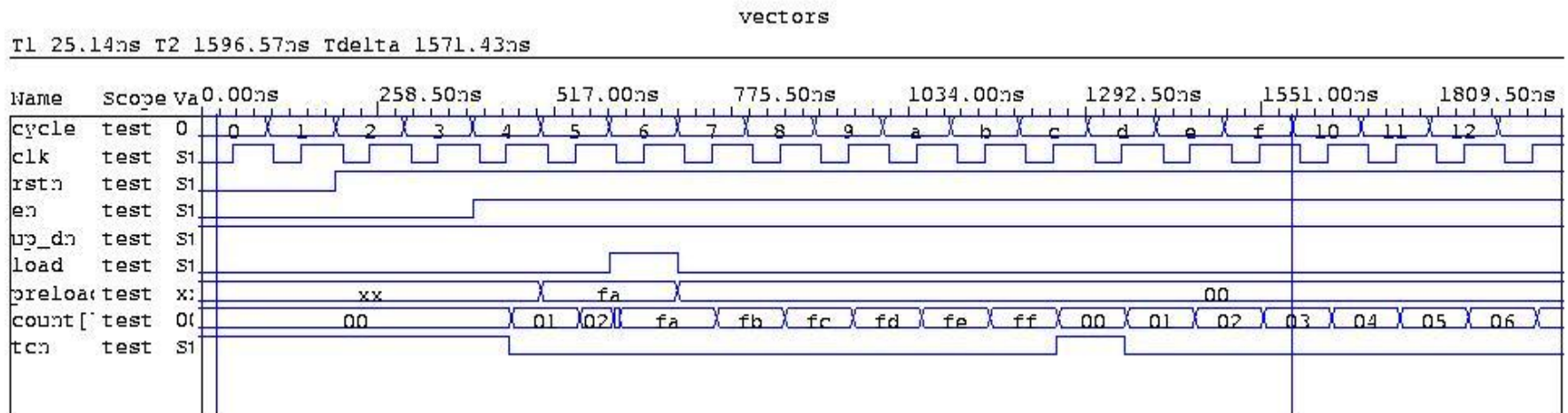
```
Parameter CP = 10; // 10ns clock period  
integer cycle; // test cycle count
```

```
initial  
begin  
cycle = 0; // initialize cycle  
forever  
    #(CP) cycle = cycle + 1; // increment cycle at clock period  
end
```

Vector generation

- Two types of vectors are typically generated for an ASIC design
 - Strobe : Once per clock cycle the input and output signals are sampled. Used for logic verification after synthesis. Can also be used for module level
 - All signals except for clocks can only change once per cycle, commonly called non-return-zero (NRZ)
 - Clocks should be return-to-1 (RO) or return-to-0 (RZ), meaning they can transition twice per test vector cycle.
 - Print on change : Whenever any input or output signal changes, write the entire I/O signal list to the print on change file
 - Use monitor statement to write signals to file whenever any signal in list changes.

Vector generation



Vector generation

- Strobe example

```
initial // loop for cycle counter and strobe vector file generation
begin
file1 = $fopen("strobe.vec"); // open vector file named "strobe.vec"
cycle = 0; // initialize cycle
forever
begin
#(CP*0.90) $fwrite(file1, "%d %b %b %b %b %b %b %b %b\n", cycle, clk, rstn, en, up_dn, load, preload, count, tcn );
// wait 90% of clock period for signals to settle
#(CP*0.10) cycle = cycle + 1; // increment cycle at following clock period
end
end
```

```
0 1 0 0 1 0 xxxxxxxx 00000000 1
1 1 0 0 1 0 xxxxxxxx 00000000 1
2 1 1 0 1 0 xxxxxxxx 00000000 1
3 1 1 0 1 0 xxxxxxxx 00000000 1
4 1 1 1 1 0 xxxxxxxx 00000001 0
5 1 1 1 1 0 11111010 00000010 0
6 1 1 1 1 1 11111010 11111010 0
7 1 1 1 1 0 00000000 11111011 0
8 1 1 1 1 0 00000000 11111100 0
9 1 1 1 1 0 00000000 11111101 0
10 1 1 1 1 0 00000000 11111110 0
11 1 1 1 1 0 00000000 11111111 0
12 1 1 1 1 0 00000000 00000000 1
13 1 1 1 1 0 00000000 00000001 0
14 1 1 1 1 0 00000000 00000010 0
```

Vector generation

- Print on change example

```
initial // print on change file
begin
file2 = $fopen("change.vec"); // open vector file named "change.vec"
$fmonitor(file2, "%d %b %b %b %b %b %b %b %b", $time, clk, rstn, en, up_dn, load, preload, count, tcn
);
// monitor signals
end
```

```
0 0 0 0 1 0 xxxxxxxx 00000000 1
50 1 0 0 1 0 xxxxxxxx 00000000 1
110 0 0 0 1 0 xxxxxxxx 00000000 1
150 1 0 0 1 0 xxxxxxxx 00000000 1
200 1 1 0 1 0 xxxxxxxx 00000000 1
210 0 1 0 1 0 xxxxxxxx 00000000 1
250 1 1 0 1 0 xxxxxxxx 00000000 1
310 0 1 0 1 0 xxxxxxxx 00000000 1
350 1 1 0 1 0 xxxxxxxx 00000000 1
400 1 1 1 1 0 xxxxxxxx 00000000 1
410 0 1 1 1 0 xxxxxxxx 00000000 1
450 1 1 1 1 0 xxxxxxxx 00000000 1
453 1 1 1 1 0 xxxxxxxx 00000000 0
457 1 1 1 1 0 xxxxxxxx 00000001 0
500 1 1 1 1 0 11111010 00000001 0
```

Tips and techniques

- Use a parameter statement declare the clock period and have all timing in test bench use it for reference
 - “parameter CP = 100”; “#(CP*2)”
 - Makes a scaleable test bench such that timing changes are simple for frequency changes.
 - Keep test bench synchronized such that independent initial and always blocks keep events occurring at proper times
- Comment, Comment, Comment!!
 - Easier test bench modifications and maintainability
- Use integer or string markers in the tester to describe which section is running
 - Useful when using waveform viewer
- Use tasks for any repeated steps in test bench
 - uP/uC read/write routines

System Calls

- System calls are used in test benches to provide high level system calls to perform “operating system” like functions. **Not synthesizable.** Extremely useful for importing and exporting data, traces, and messages during simulation. These commands make a Verilog simulator a complete language and operating system solution. Please note that not all Verilog simulators have all of these commands.

\$time

- Used for getting simulator time, the value returned is the real value of the current simulator time. The simulator time unit is whatever the `timescale reference was chosen for the time units. Useful for determining when a particular event occurred during a simulation.

```
$display("time = %d", $time);
```

```
...
```

```
time = 0
```



Variable linking in print statements are similar to "C" code

\$display()

- Used for displaying signals and messages in the simulator window. Very similar to the *printf* command in “C” language. Arguments can be formatted to similar standards that “C” uses. A new line is enabled by default.

```
$display("Hello World this is the value of my counter in hex %h, in decimal %d, in binary %b", count,  
count, count);
```

...

```
Hello World this is the value of my counter in hex 3d, in decimal 61, in binary 111101
```

String formats

Used for display, write, monitor, and strobe

| <u>Format</u> | <u>Display</u> |
|---------------|---|
| %d | Display variable in decimal * |
| %b | Display variable in binary * |
| %o | Display variable in octal * |
| %h | Display variable in hex * |
| %s | Display string * |
| %m | Display hierarchical name |
| %v | Display strength |
| %t | Display in current time format |
| %e | Display real number in scientific format |
| %f | Display real number in decimal format |
| %g | Display real number in decimal or scientific format |

* = most commonly used formats

\$write()

- Used for displaying signals and messages in a simulation. Very similar to the *printf* command in “C” language. Arguments can be formatted to similar standards that “C” uses. The same as \$display *except that there is no new line.*

```
$write("Hello World this is the value of my address in hex %h", addr);
```

```
$write(", in decimal %d", addr);
```

```
$write(", in binary %b \n", addr);
```

```
...
```

```
Hello World this is the value of my address in hex e, in decimal 14, in binary 1110
```

\$monitor()

- Used for monitoring and displaying any activity on a group of signals in a simulation. Any change on any signal in the statement causes the entire list of signals and/or messages to appear on the display window in the Verilog simulator. Messages can also be displayed along with the signals. Can use any number of \$monitor() in a test bench, however the last \$monitor() overrides the previous \$monitor(). There are two additional commands \$monitoron and \$monitoroff that enable and disable the monitoring.

```
$monitor("time = %d, interrupt = %b, parity = %b", $time, interrupt, parity);
```

```
...
```

```
time = 0 , interrupt = x , parity = x
```

```
time = 100 , interrupt = 0 , parity = x
```

```
time = 200 , interrupt = 0 , parity = 0
```

```
time = 21700 , interrupt = 1 , parity = 0
```

\$strobe()

- Used for displaying signals and messages in a simulation. Very similar to the *printf* command in “C” language. Arguments can be formatted to similar standards that “C” uses. A new line is enabled by default. The same as \$display *except that simulator waits until all the simulation events to have executed and settled for the current time step.*

```
$strobe("clock = %b, reset = %b, chip select = %b", clk, rstn, csn);
```

```
...
```

```
clock = 1, reset = 1, chip select = 0
```

\$stop

- Used to halt the Verilog simulation. Can be used multiple times in a test bench to stop the simulation. Very useful for halting the simulation when an anomaly or error condition occurs in circuit to allow manual intervention. Entering a “.” (period) with a return at the simulator will restart the simulator until the next \$stop or \$finish.

```
always @(parity_error)
    if(parity_error)
        $stop;
```


\$finish

- Used to complete the Verilog simulation and exit. Used at the very end of the simulation so that the simulator does not run forever.

```
$stop; // halt simulator  
$finish; // exit  
endmodule // test bench
```

\$fopen

- Used to open a file to write simulation data or messages to. Very similar to file opening in “C”. Generally a file is opened for writing simulation vectors to compare simulations results from HDL to gate level or verification. Another good use is to write “display” or “monitor” messages to a file for observing test results. A integer variable is assigned as the file handle for use in the test bench such that multiple files can be opened simultaneously

```
integer file1, file2; // file variables
```

```
initial // open files
```

```
begin
```

```
file1 = $fopen("test.vec");           // test vector file "test.vec"
```

```
file2 = $fopen("./test_results/qpsk_tst.msg"); // test message file "qpsk_tst.msg" under "test_results"
```

```
end
```

\$fclose

- Used to close a opened file such that no more writes can occur to file. Typically performed at the end of the simulation.

```
integer file1, file2; // file variables
```

```
initial // open files
```

```
begin
```

```
file1 = $fopen("test.vec");           // test vector file "test.vec"
```

```
file2 = $fopen("./test_results/qpsk_tst.msg"); // test message file "qpsk_tst.msg" under "test_results"
```

```
end
```

```
...
```

```
initial
```

```
begin
```

```
...
```

```
$fclose(file1);
```

```
$fclose(file2);
```

```
$finish;
```

```
end
```

\$fdisplay/\$fwrite/\$fmonitor/\$fstrobe

- Similar syntax and format as \$display/\$write/\$monitor/\$strobe except that the output goes to a file and not the simulator window. The first argument in all of the calls is the integer file variable to indicate which file to write to.

- integer file1; // file variable

initial // open files

```
file1 = $fopen("notes.msg"); // simulation notes file "test.vec"
```

...

initial

```
begin
```

```
$fdisplay(file1,"Hello World the time is %d",$time);
```

```
$fwrite(file1,"Hello World the SQF signal is %d \n", sqf);
```

```
$fmonitor(file1,"Hello World the interrupt is %b ", inter);
```

```
$fstrobe(file1,"Bye World the h-register is %h", h_reg);
```

```
end
```

\$readmemb()/readmemh()

Files can be read into a Verilog simulation using the \$readmemb() and \$readmemh() commands. This is extremely useful for reading in data from a stimulus file. The \$readmemb() is used to read binary values and \$readmemh() is used to read hex values. The commands are limited for it can only read binary and hex data in array format but are still very powerful and useful for bring in external data into a simulation.

```
module test();  
...  
reg [7:0] test_data [1023:0]; // declare a 1Kx8 register array for test data  
...  
initial  
    $readmemh("qpsk_iq.vec",test_data,0,1023);  
    $display("test_data[2] = %h", test_data[2]);  
...  
test_data[2] = 82
```

“two dimensional register array 8bits by 1024 address”

“file name” “register array” “start address vector” “stop address vector”

Or use @ to specify address
in data file, incremental
addresses

qpsk_iq.vec
5e
f9
82
a0
b3
...

“or”

qpsk_iq.vec
@0002
82
a0
b3
...

\$random()

A random number generator is included in Verilog that returns a 32 bit value. A optional seed can be used to get identical start values for every simulation run.

```
...  
reg [7:0] ran_data;  
...  
initial  
    $random(25); // use 25 as first seed  
...  
always @(vector)  
    ran_data = $random;  
...
```

References

- Books
 - “Verilog HDL” by Samir Palnikkar
 - “Quick Reference for Verilog HDL” by Rajeev Madhavan
 - "The Verilog Hardware Description Language" by D.E. Thomas and P.R. Moorby
- Recommended Web sites
 - <http://www.ovi.org>
 - <http://www.verilog.net>
 - <http://www.ee.ed.ac.uk/~gerard/Teach/Verilog/>
 - <http://www.angelfire.com/in/rajesh52/verilog.html>
 - <http://www.europa.com/~celiac/verilog-manual.html>
 - <http://www.vol.webnexus.com/>
 - <ftp://ftp.siliconlogic.com/pub/comp.lang.verilog/verilog-faq.html>

Practical example of Verilog HDL

- Multiplexers
- Registers
- Counters
- Decoders
- Micro-Controller Interface

Multiplexers

```
module mux4to1a (A,B,C,D,Y,SEL); // AND-OR wire technique example of four to one mux
parameter bw = 8 ; // bitwidth
input [1:0] SEL ; // input mux select
input [bw-1:0] A, // A,B,C,D mux inputs
    B,
    C,
    D;
output [bw-1:0] Y; // Y output
wire [bw-1:0] Y; // Y wire
assign Y = ({bw{SEL == 2'd0}} & A) // and-or mux HDL
    | ({bw{SEL == 2'd1}} & B)
    | ({bw{SEL == 2'd2}} & C)
    | ({bw{SEL == 2'd3}} & D);
endmodule
```

```
module mux4to1b (A,B,C,D,Y,SEL); // Decision "?" technique example of four to one mux
parameter bw = 8 ; // bitwidth
input [1:0] SEL ; // input mux select
input [bw-1:0] A, // A,B,C,D mux inputs
    B,
    C,
    D;
output [bw-1:0] Y; // Y output
wire [bw-1:0] Y; // Y wire
assign Y = SEL[1] ? (SEL[0] ? D : C) : (SEL[0] ? B : A); // concatenated ? : technique
endmodule
```

Multiplexers

```
module mux4to1c (A,B,C,D,Y,SEL); // "case" reg technique example of four to one mux
parameter bw = 8 ; // bitwidth
input [1:0] SEL ; // input mux select
input [bw-1:0] A, // A,B,C,D mux inputs
           B,
           C,
           D;
output [bw-1:0] Y; // Y output
reg [bw-1:0] Y; // Y reg
always @(A or B or C or D or SEL) // reg technique with a mux
    case (SEL) // case of select
        2'd0 : Y = A ;
        2'd1 : Y = B ;
        2'd2 : Y = C ;
        2'd3 : Y = D ;
    endcase
endmodule
```

Registers

```
// Examples of various registers in Verilog HDL
module registers (CLK, DATA, DATA_A, RSTN, ENABLE, MUX_SEL, FF, FF_R, FF_E, FF_ER,
                 FF_MR, FF_GR, FF_SH, LT_R); // different register styles
parameter bw = 4 ; // bitwidth
input [3:0] DATA,           // input nibble byte
        DATA_A;           // second nibble data
input CLK,                   // clock
        RSTN,               // reset active low
        ENABLE,             // clock enable
        MUX_SEL;            // mux select
output [bw-1:0] FF,          // f/f reg
        FF_R,               // f/f with async. reset
        FF_E,               // f/f with enable
        FF_ER,              // f/f with async. reset and enable
        FF_MR,              // f/f with mux and async. reset
        FF_GR,              // f/f with gated clock and async. reset
        FF_SH,              // shift register
        LT_R;               // latch with async. reset ;
reg [bw-1:0] FF, // f/f reg
        FF_R,               // f/f with async. reset
        FF_E,               // f/f with enable
        FF_ER,              // f/f with async. reset and enable
        FF_MR,              // f/f with mux and async. reset
        FF_GR,              // f/f with gated clock and async. reset
        FF_SH,              // shift register
        LT_R;               // latch with async. reset ;
```

Registers

```
always @(posedge CLK) // f/f register
    FF <= DATA;
```

```
always @(posedge CLK or negedge RSTN) // f/f register with active low async. reset
    if (!RSTN)
        FF_R <= 4'h0;
    else
        FF_R <= DATA;
```

```
always @(posedge CLK) // f/f register with active high enable
    if (ENABLE)
        FF_E <= DATA;
```

```
always @(posedge CLK or negedge RSTN) // f/f register with active low async. reset and active high enable
    if (!RSTN)
        FF_ER <= 4'h0;
    else if (ENABLE)
        FF_ER <= DATA;
```

```
always @(posedge CLK or negedge RSTN) // mux'ed input f/f with active low async. reset
    if (!RSTN)
        FF_MR <= 4'h0;
    else
        FF_MR <= MUX_SEL ? DATA : DATA_A;
```

Registers

```
// gated clock wire/reg for gated clock with latch enable
reg clk_en;      // clock enable latch
wire gate_clk; // gated clock

always @(CLK)
  if (!CLK) //transparent latch when CLK
    clk_en <= ENABLE;
  assign gate_clk = CLK & clk_en;

always @(posedge gate_clk or negedge RSTN) // f/f with gated clock
  if (!RSTN)
    FF_GR <= 4'h0;
  else
    FF_GR <= DATA;

always @(posedge CLK or negedge RSTN) // left shift register with active low async. reset
  if (!RSTN)
    FF_SH <= 4'h0;
  else
    FF_SH <= {FF_SH[bw-2:0],DATA[0]};

always @(DATA or CLK or negedge RSTN) // active low transparent latch with active low async. reset
  if (!RSTN)
    LT_R <= 4'h0;
  else if (!CLK)
    LT_R <= DATA;

endmodule
```

Counter

```
module count_ud4 (CLK, RSTN, EN, UP_DNN, LOAD, PRELOAD, COUNT, TCN); // 4 bit up/down counter
input  CLK,      // synch. clock
      RSTN,     // asynch. reset, active low
      EN,       // counter enable, active high
      UP_DNN,   // up = 1 , down = 0
      LOAD;     // counter preload control, active high
input [3:0] PRELOAD; // counter preload value
output [3:0] COUNT; // counter output
output TCN;      // active low terminal count
wire TCN;
reg [3:0] COUNT;
assign TCN = UP_DNN ? (COUNT == 4'hf) : (COUNT == 4'h0) ; // up = 1111 down = 0000
always @(posedge CLK or negedge RSTN)
  if (!RSTN)      // clear count
    COUNT <= 4'b0;
  else if (EN)    // counter enable
    begin
      if (LOAD)
        COUNT <= PRELOAD;      // preload if enabled
      else if (UP_DNN) // count up
        COUNT <= COUNT + 4'd1; // increment
      else // count down
        COUNT <= COUNT - 4'd1; // decrement
    end
end
endmodule
```

Decoders

```
// Example of various decoders
module decoders (DECODE, DECODE_EN, SELECT, ENABLE); // decode

input [2:0] SELECT;      // data select
input ENABLE;            // decode enable

output [7:0] DECODE;     // decoded output
output [7:0] DECODE_EN; // decoded output with enable

reg [7:0] DECODE,
          DECODE_EN;

always @(SELECT) // 3 to 8 decoder
  case(SELECT)
    3'd0 : DECODE = 8'b00000001; // decode = 0
    3'd1 : DECODE = 8'b00000010; // decode = 1
    3'd2 : DECODE = 8'b00000100; // decode = 2
    3'd3 : DECODE = 8'b00001000; // decode = 3
    3'd4 : DECODE = 8'b00010000; // decode = 4
    3'd5 : DECODE = 8'b00100000; // decode = 5
    3'd6 : DECODE = 8'b01000000; // decode = 6
    3'd7 : DECODE = 8'b10000000; // decode = 7
  endcase
```

Decoders

```
always @(SELECT or ENABLE) // 3 to 8 decoder with enable
if (ENABLE)
  case(SELECT)
    3'd0 : DECODE_EN = 8'b00000001 ; // decode = 0
    3'd1 : DECODE_EN = 8'b00000010 ; // decode = 1
    3'd2 : DECODE_EN = 8'b00000100 ; // decode = 2
    3'd3 : DECODE_EN = 8'b00001000 ; // decode = 3
    3'd4 : DECODE_EN = 8'b00010000 ; // decode = 4
    3'd5 : DECODE_EN = 8'b00100000 ; // decode = 5
    3'd6 : DECODE_EN = 8'b01000000 ; // decode = 6
    3'd7 : DECODE_EN = 8'b10000000 ; // decode = 7
  endcase
else
  DECODE = 8'b00000000; // no active outputs

endmodule
```


Micro-Controller Interface

```
//microcontroller interface with two general purpose parallel ports
// AVR interface 8 bit data and 16 bit address
// Active high enable for write and read
// rising edge of clock for sync write

module micro_if (RSTN, CLK, DATA, RD, WR, ADDR, PORTA, PORTB);

`define A_out_adr 16'hF0 // A output reg
`define A_dir_adr 16'hF1 // A direction reg
`define A_in_adr 16'hF2 // A input reg
`define B_out_adr 16'hF3 // B output reg
`define B_dir_adr 16'hF4 // B direction reg
`define B_in_adr 16'hF5 // B input reg

inout [7:0] DATA; // data bus
input [15:0] ADDR; // address bus
inout [7:0] PORTA; // port bus
inout [7:0] PORTB; // port bus
input RD,      // read active high
       WR,      // write active high
       CLK,     // clock rising edge
       RSTN;    // reset active low

wire [7:0] DATA, // data bus
          PORTA, // port A
          PORTB; // port B

reg [7:0] DATA_OUT; // output data bus
```

Micro-Controller Interface

```
reg [7:0] A_out, // A port output reg
    A_dir, // A port direction reg 1 = drive out
    A_in, // A Port input reg
    B_out, // B port output reg
    B_dir, // B port direction reg 1 = drive out
    B_in; // B Port input reg

wire cs; // databus tristate control

integer index; // index counter for loop

assign cs = (ADDR == `A_out_adr) // data bus tri state control for address range
    | (ADDR == `A_dir_adr)
    | (ADDR == `A_in_adr)
    | (ADDR == `B_out_adr)
    | (ADDR == `B_dir_adr)
    | (ADDR == `B_in_adr);

// tristate microncontroller interface
assign DATA = (RD & cs) ? DATA_OUT : 8'hz; // Bi-Di

always @(ADDR or A_out or A_in or A_dir or B_out or B_in or B_dir) // microcontroller interface output multiplexer
    case(ADDR)
        `A_out_adr: DATA_OUT = A_out;
        `A_dir_adr: DATA_OUT = A_dir;
        `A_in_adr: DATA_OUT = A_in;
        `B_out_adr: DATA_OUT = B_out;
        `B_dir_adr: DATA_OUT = B_dir;
        `B_in_adr: DATA_OUT = B_in;
    endcase
```

Micro-Controller Interface

```
always @(posedge CLK or negedge RSTN) // microcontroller write registers
if (!RSTN) // async reset all regs
begin
A_out <= 8'h00;
A_dir <= 8'h00;
A_in <= 8'h00;
B_out <= 8'h00;
B_dir <= 8'h00;
B_in <= 8'h00;
end
else if (!RD & WR & cs) // sync reg clocking with no read, write and chip select
case(ADDR)
`A_out_adr: A_out <= DATA;
`A_dir_adr: A_dir <= DATA;
`A_in_adr: A_in <= PORTA; // capture port A input with write
`B_out_adr: B_out <= DATA;
`B_dir_adr: B_dir <= DATA;
`B_in_adr: B_in <= PORTB; // capture port B input with write
default: ;
endcase
```

Micro-Controller Interface

```
assign PORTA[7] = A_dir[7] ? A_out[7] : 1'bz ; // if port dir = 1, drive out
assign PORTA[6] = A_dir[6] ? A_out[6] : 1'bz ; // if port dir = 1, drive out
assign PORTA[5] = A_dir[5] ? A_out[5] : 1'bz ; // if port dir = 1, drive out
assign PORTA[4] = A_dir[4] ? A_out[4] : 1'bz ; // if port dir = 1, drive out
assign PORTA[3] = A_dir[3] ? A_out[3] : 1'bz ; // if port dir = 1, drive out
assign PORTA[2] = A_dir[2] ? A_out[2] : 1'bz ; // if port dir = 1, drive out
assign PORTA[1] = A_dir[1] ? A_out[1] : 1'bz ; // if port dir = 1, drive out
assign PORTA[0] = A_dir[0] ? A_out[0] : 1'bz ; // if port dir = 1, drive out
```

```
assign PORTB[7] = B_dir[7] ? B_out[7] : 1'bz ; // if port dir = 1, drive out
assign PORTB[6] = B_dir[6] ? B_out[6] : 1'bz ; // if port dir = 1, drive out
assign PORTB[5] = B_dir[5] ? B_out[5] : 1'bz ; // if port dir = 1, drive out
assign PORTB[4] = B_dir[4] ? B_out[4] : 1'bz ; // if port dir = 1, drive out
assign PORTB[3] = B_dir[3] ? B_out[3] : 1'bz ; // if port dir = 1, drive out
assign PORTB[2] = B_dir[2] ? B_out[2] : 1'bz ; // if port dir = 1, drive out
assign PORTB[1] = B_dir[1] ? B_out[1] : 1'bz ; // if port dir = 1, drive out
assign PORTB[0] = B_dir[0] ? B_out[0] : 1'bz ; // if port dir = 1, drive out
```

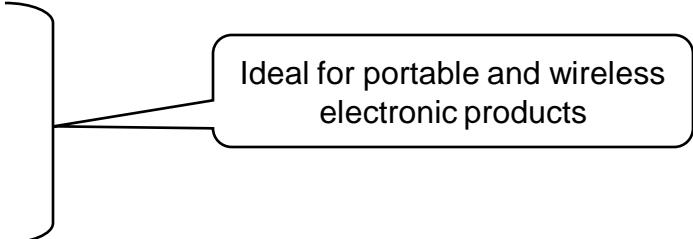
```
endmodule
```

Top Down/Bottom Up Design Flow

- Several different design techniques are used for HDL coding and integration.
 - “Top Down” refers to developing the top level block and identifying and develop sub-blocks in the hierarchy until all of the blocks have been described by function and pinout.
 - Most common flow for ASIC and System on a Chip designs.
 - Top level of chip (top_asic.v) is typically only the pinout with no logic only Input/Output buffers and a core (top_core.v) containing all the subblocks
 - “Bottom Up” refers to the developing the bottom layer of hierarchy first then combining cells until the hierarchy builds up to the top level.
 - Most common flow for standard parts (memory,logic,..) and some ASSP (Application Specific Standard Part).

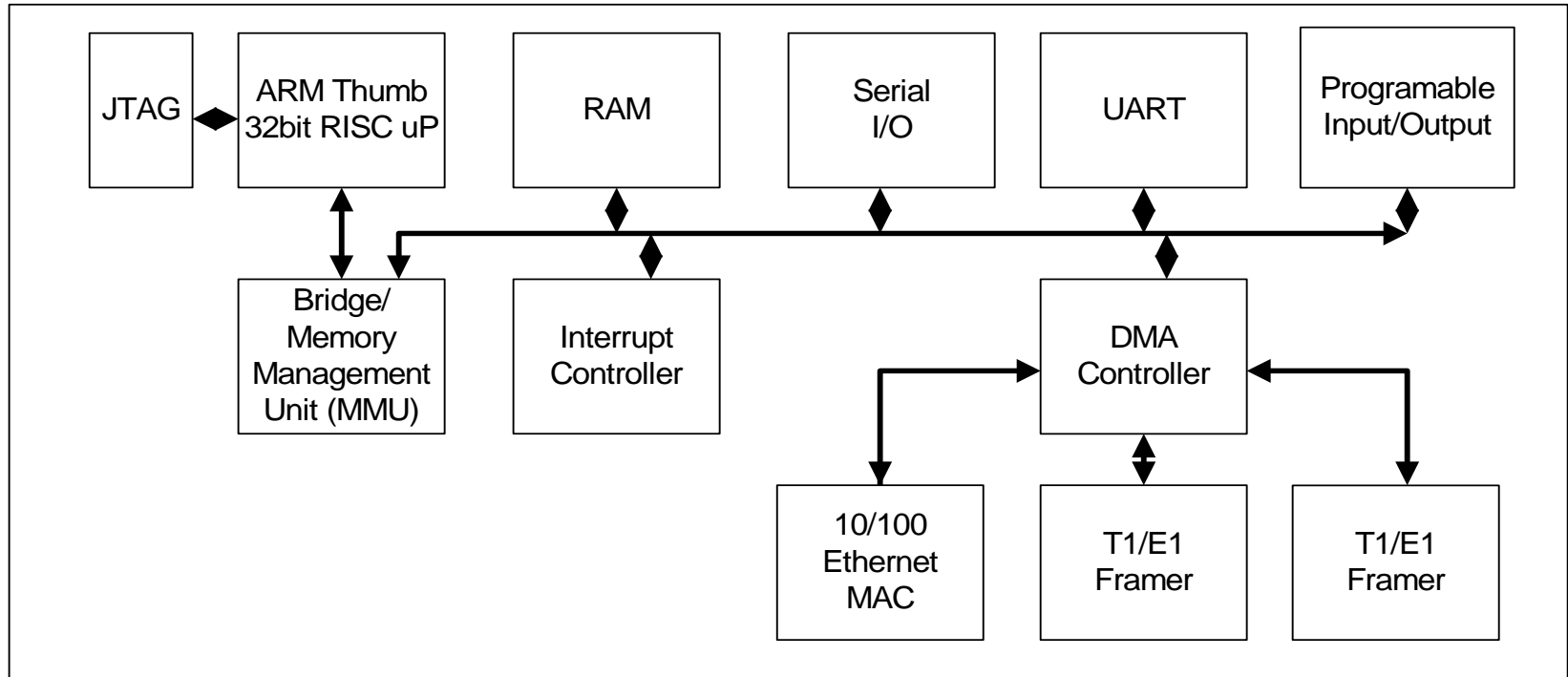
System Level Integration

- System Level Integration (SLI) or “System on a Chip” design is the integration of traditional board level components into one chip.
 - Great reduction in end product.
 - Cost
 - Power
 - Weight
 - Volume
 - Manufacturing time/overhead
 - Increase in end product.
 - Development risk
 - Reliability
 - Vendor independence
- A vast array of building blocks are used to integrate and develop complete silicon solutions.



Ideal for portable and wireless electronic products

System Level Integration



System On a Chip (SOC) Example

System Level Integration

- Microprocessor/Microcontrollers
 - 8/16/32bit processors
 - ARM/AVR/x86
 - CISC/RISC
- Digital Signal Processing
 - Floating/Fixed point
 - Programmable/Fixed
 - ALU (MAC's, barrel, shifters)
- Memory
 - SRAM (single, dual, multi-port)
 - FIFO
 - ROM
 - FLASH
 - EEPROM
- Peripherals
 - UARTS
 - DMA
 - Smart Card
 - Interrupt Controller
- Analog
 - PLL
 - ADC
 - DAC
 - Filters
- Error Detection/Correction
 - Reed/Solomon
 - Viterbi
 - Turbocode
- Security
 - DES
 - RSA
- Media
 - JPEG
 - MPEG
 - Voice Codecs (G.7XX)
- Interfaces
 - PCI
 - USB
 - Firewire
 - Bluetooth
 - CAN
 - Framers
 - Ethernet
 - Fibrechannel
- Communications
 - Demodulation
 - Modulation
 - NCO
 - Ethernet
 - QPSK/QAM
 - DSL

System Level Integration

- HDL's are ideal for SLI designs since design architecture/hierarchy can be broken down and assigned to engineering teams.
 - Allows concurrency in design development and shorter time to market.
 - Excellent Design Reuse with Intellectual Property (IP) coreware libraries
 - Complete tested and verified blocks of code to implement a standard function.
 - In-house development
 - Purchase third party
 - Design lead role of organizing interfaces between IP blocks and random logic specific to design.
 - Minimize number of clocks
 - Common synchronous interfaces

System Level Integration

- A key element of SLI designs is the use of uP/uC which provide intelligent control of design.
 - Requires S/W support for host code/drivers
 - Emulator/Debugger for allowing control and status of processor.
 - Co-simulation of HDL (i.e. Verilog) and S/W code (i.e. “C”)
 - Allows real-world testing of intergrated S/W and H/W solution
- Crucial task of design verification
 - Need to verify complete functionality of system
 - Requires greater system understanding to design and verify

Design for Reuse

- To improve design productivity, partition architecture such that HDL modules which can be used again are self contained.
 - i.e. a UART (16550), HDLC, PCI, USB
- Document any Verilog module for reuse with a full description in the header section describing functionality and port signal definitions.
- Verify the design completely with test bench and preferably in silicon
 - After verification archive database with design HDL, testbench HDL, synthesis scripts, and documentation.
 - Allow a common directory location for HDL libraries.

Practical Coding Techniques

- Code layout and format
- Signals
- Design Hierarchy and partitioning
- always blocks for combinatorial logic
- HDL versus gates
- Multiple assignments for reg or wire
- Revision Control
- Design Directory Organization

Code layout and format

- Use common standard for layout order for all HDL code
 - Header with author, revision, copyrights, description
 - module declaration
 - include files
 - defines and parameters
 - port declarations with descriptions
 - wire, reg, and integer variables
 - assign statements for wires
 - always blocks for for regs
 - functions
- File name should equal module name with “.v” extension
- Keep one module per file, simpler file and configuration management

Code layout and format

- Keep code width to 80 character maximum
- Use judgement to keep file length reasonable.
- Comment code
 - Header section should contain description
 - Port designations should contain port description and function
 - Sections of code should be commented to explain logic
- Indent code for structure and readability, use 2-3 character spaces (and not 8 like most tab settings)
 - “if-else” structures
 - “case” statements
 - “begin-end” code

Code layout and format

- Place blank lines between sections in code to improve readability.
 - Module and port declaration
 - reg and wire section
 - each major section of logic
- Use one global “.h” include file for defines which are global in the design.
 - Register/Memory map
 - Databus bit widths
- Always assign value to corresponding bit widths. Use parameters and defines for bitwidth.

```
`define adc_high 8'hf4
...
if (addr == `adc_high)
    data_out = result;
```

Code layout and format

- Organize code that relates together into sections.
- Use either one large single or multiple “always @(posedge CLK..” sections for synchronous logic.
 - Individual allows better grouping and slightly faster synthesis run times
 - Single section shows all state variable in one location

Signals

- Keep signal length to reasonable limit
 - 8-12 characters typical
- Practical names and abbreviations
 - clock = “CLK”, reset = “RST”, write = “WR”, output enable = “OE”
- Use all capitals for port names
- Use last character of “n” or “N” for active low signal designation, i.e. “wrn”
- Signals which pass through module to lower hierarchy should not change name.

Design Hierarchy and partitioning

- First level of hierarchy (top_asic) in VLSI design should only contain I/O pins, buffers and “top_core” module
 - Top 2 levels of hierarchy (top_asic, top_core) should only contain interconnects (no logic).
- Second level of hierarchy (top_core) should contain a clock/reset module, test module (JTAG/Scan), and major modules.
- Code coupling and cohesion
 - Keep code that is related together.
 - Create addition hierarchy if code becomes too complex or long
- Always use explicit port naming and not port order for module connections in hierarchy
 - Revision and changes to module port order do not affect higher levels of hierarchy.

Always blocks for combinatorial logic

- Use caution whenever using regs for asynchronous combinatorial logic.
 - Put all inputs for function in always block or else latches will be inferred.

```
always @(a or b or c or d or e)
    f = a ? (b & c) : (d | e);
```

- If using CASE statements with multiple combinatorial outputs make certain all outputs are on every case decode line or latches will be inferred.

```
case (select)
2'd0: begin out = a ; parity = a ^ b ; end
2'd1: begin out = b ; parity = b ^ c ; end
2'd2: begin out = c ; parity = c ^ d ; end
2'd3: begin out = d ; parity = 1'b0 ; end
endcase
```

HDL versus gates

- HDL level simulation results will not always match gate level simulation.
 - It's always important to remember to design logic and write HDL code which can be properly initialized, tested, and synthesized.
 - A “if” and “casex” statement can evaluate “x” unknown variables in HDL but not always in gates.
 - A “AND” gate with a logic “0” on one pin or a “OR” gate with a logic “1” on a signal will always block off a unknown “X”.
 - Initializing all internal states to known values through asynchronous clears and test bench register initialization.
 - Datapath logic with registers can avoid resets as long as pipeline registers are “flushed” until all states are known.
 - Any control logic with synchronous feedback must have a asynchronous reset to get states to known state.

HDL versus gates

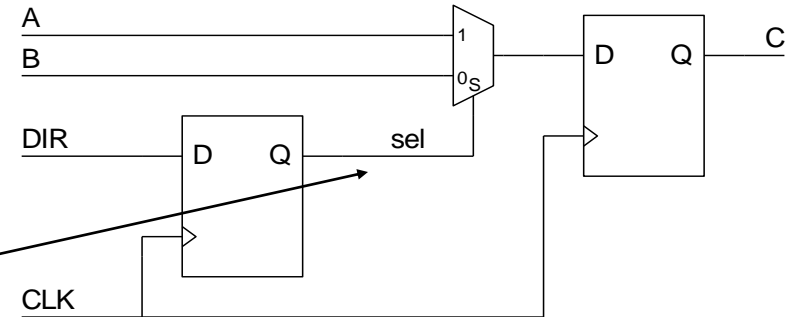
HDL

Synthesis

gates

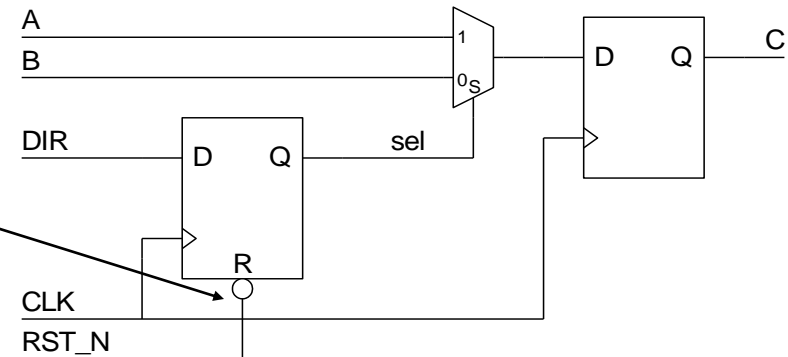
```
reg sel, C;
always @(posedge CLK)
begin
  sel <= DIR;
  if (sel)
    C <= A;
  else
    C <= B;
```

In HDL simulation C will register B since "sel" != 1 on the 1st rising clock. This will occur even before SEL becomes a known value. However a "X" on the select input to the mux will cause a "X" on the output and C after the clock.



```
reg sel, C;
always @(posedge CLK or negedge RST_N)
if (!RST_N)
  sel <= 1'b0;
else
  sel <= DIR;
always @(posedge CLK)
if (sel)
  C <= A;
else
  C <= B;
```

The solution is to asynchronously clear "SEL" with "RST_N". Then a known value will get clocked into "C" register.



Multiple assignments for reg or wire

- Verilog allows multiple assignment statements to regs in always and wires in assign. Care needs to be observed in order to prevent unintended logic anomalies and errors.
- With the exception of tri-state signals only use one assign statement for a combinatorial signal.
- Multiple assignment statements can be used for always blocks with a few requirements.
 - Multiple assignments in always block should be part of an “if-else” or “case” structure for a “reg” signal.
 - Do not use multiple always blocks for the same “reg” signal
 - Parallel always reg assignments with non-blocking procedural assignments (“<=”) will get assigned to in-determinant value.
 - Parallel wire “assign”s will get assigned to the unknown “x”s for different logic values.

Multiple assignments for reg or wire

```
wire [1:0] E;
reg [1:0] D;
```

```
assign E = C;
assign E[1] = B;
assign E[0] = A;
```



Two different "assign"
statements for the
same wire bit

Two different procedural
statements for the
same reg bit

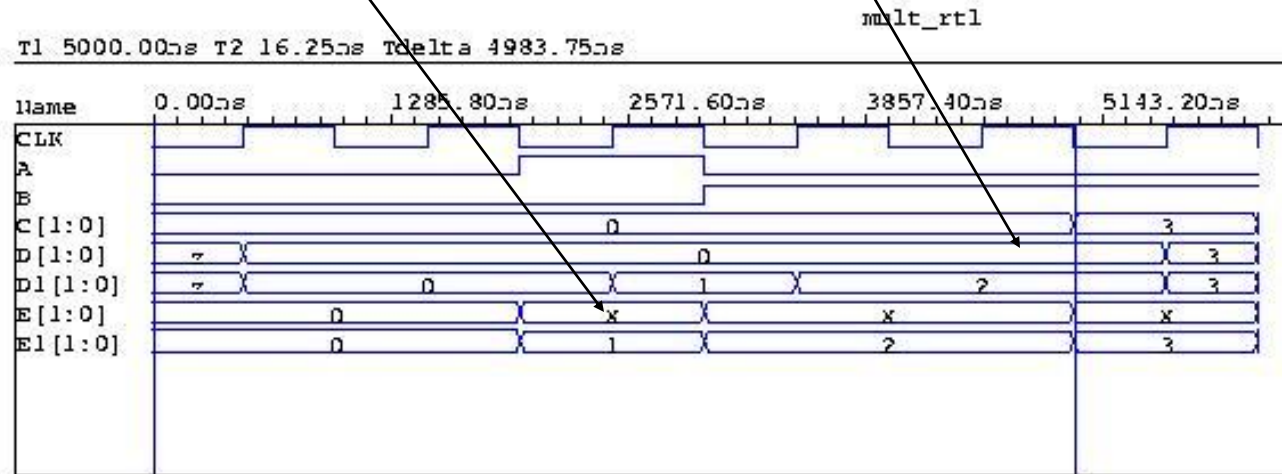
```
always @(posedge CLK)
begin
D[1] <= B;
D[0] <= A;
D <= C;
end
```



All variables
in one assign

```
wire [1:0] E1;
reg [1:0] D1;
assign E1 = C | {B,A};
always @(posedge CLK)
if (C >= 2'd2)
D1 <= C;
else
D1 <= {B,A};
```

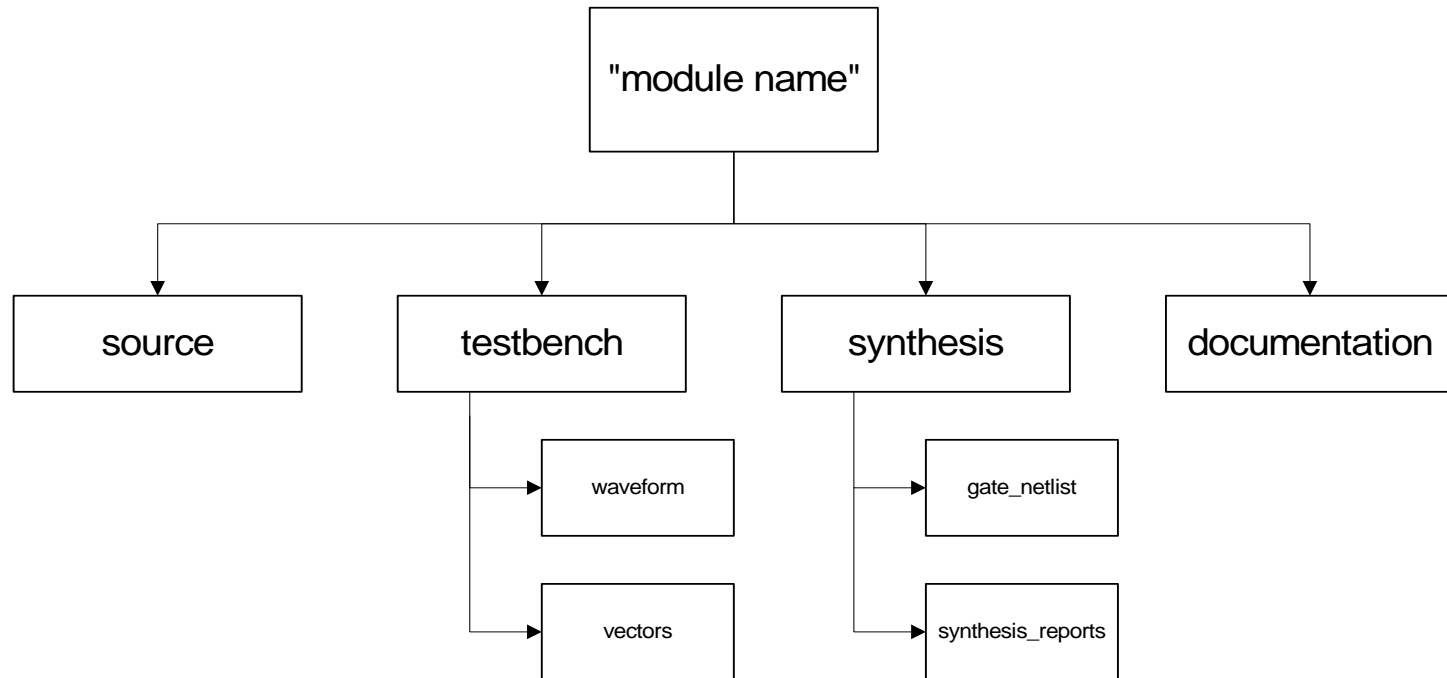
All assignments
in one "if"
structure



Revision Control

- Use a Revision Control System (RCS) to maintain code versions and checkin/out control.
 - Increment Revision of code after preliminary module level test and debug complete.

Design Directory Organization



Suggested directory structure for Verilog HDL to design, test, and synthesis into gates.

Practical Design Techniques

- Synchronous design
- Resets and clocks
- Clock enables versus gated clocks
- Tristate versus muxed busing
- Inputs and Outputs
- Interfacing asynchronous signals
- Asynchronous set and clear

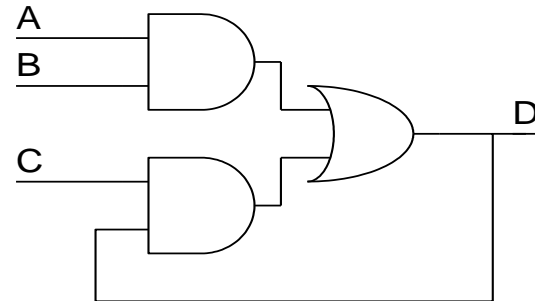
Synchronous design

- Keep design as synchronous as possible using as few clocks as needed.
 - Simpler design flow
 - Timing analysis reduced
 - Design portability
 - Fewer clock trees
 - Setup/Hold timing issues
 - Higher quality designs (testability, production - not silicon specific)
 - Use flip-flops and avoid latches
 - edge sensitive versus level sensitive design
 - latches are faster and smaller but require careful design practice and consideration
 - flip-flops only require one clock versus two non overlapping clocks for latches

Synchronous design

- Strictly avoid any combinatorial or asynchronous logic with feedback including cross-coupled gates.
 - Race conditions, unpredictable, untestable, and unproducable

```
wire A,B,C,D;  
assign D = (A & B) | (C & D);
```



- Use memory, flipflops or latches to hold logic states
 - No need in Verilog to code states being held without change

```
always @(posedge CLK)
```

```
  if (EN)
```

```
    Q <= D;
```

```
  else
```

```
    Q <= Q;
```

Not needed



Synchronous design

- Use registered outputs for all non-asynchronous logic in module ports.
 - Simpler timing paths (only clock) for synthesis.
- For larger blocks in SLI designs register inputs if possible.
 - Long interconnect timing delays can upset timing requirements. Using a clocked register output of one large block in a design to a register input of another makes timing requirements easier.

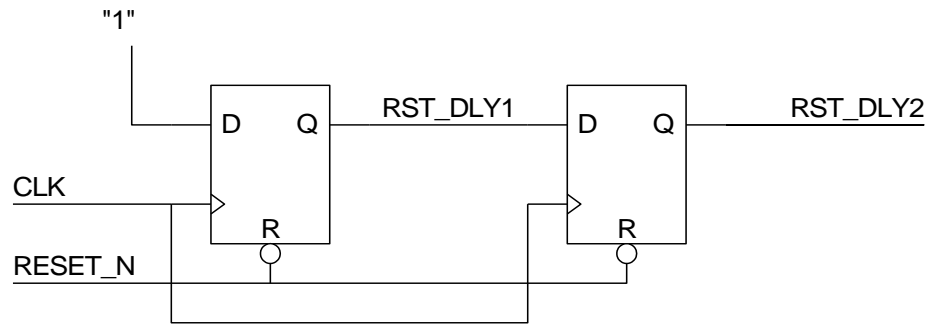
Resets and Clocks

- Minimize the number of clocks in the design
 - If harmonic frequencies are used (80MHz, 40MHz, 20MHz,...) then use the highest frequency clock and use clock enables for lower frequency.
- Limit use of one clock per Verilog module (unless interface module for signals between two clock domains)
- Use only **one** edge (rising/falling) of any given clock
- **Never** asynchronously gate together multiple clocks to form a single clock.
- Use one common block at core level hierarchy for clock and reset generation.
 - Simple management of clock tree's
 - Use common reset buffer driver

Resets and Clocks

- Use a common reset that is asynchronously applied and synchronously released with the clock used for the logic.

```
always @(posedge CLK or negedge RESET_N)
if (!RESET_N)
begin
RST_DLY1 <= 1'b0;
RST_DLY2 <= 1'b0;
end
else
begin
RST_DLY1 <= 1'b1;
RST_DLY2 <= RST_DLY1;
end
end
```

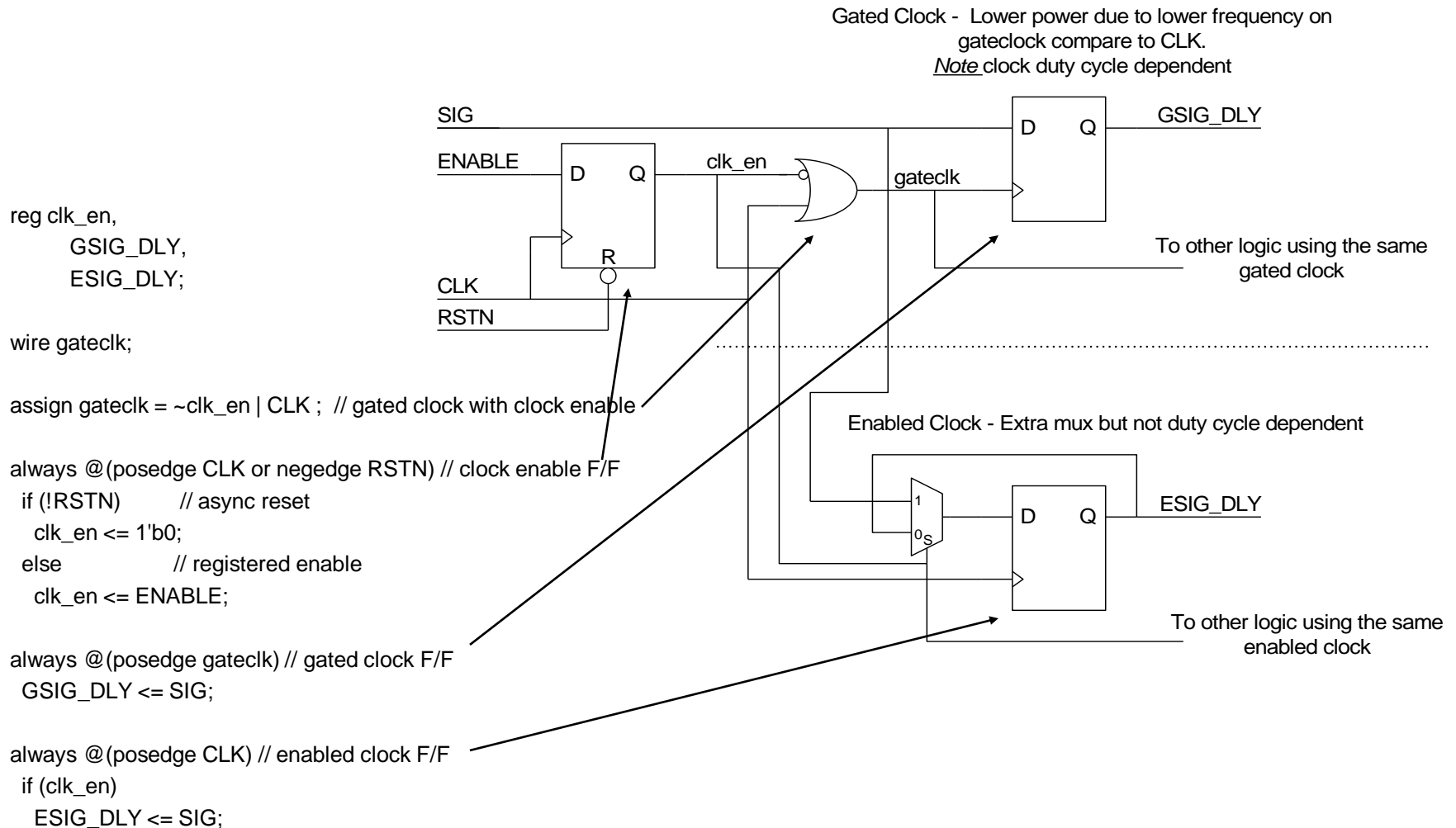


- Use async. reset on all control logic, state machines, and control/status registers.
 - Extremely useful for gate level versus HDL simulation results
- Datapath logic can avoid resets as long as no feedback paths.

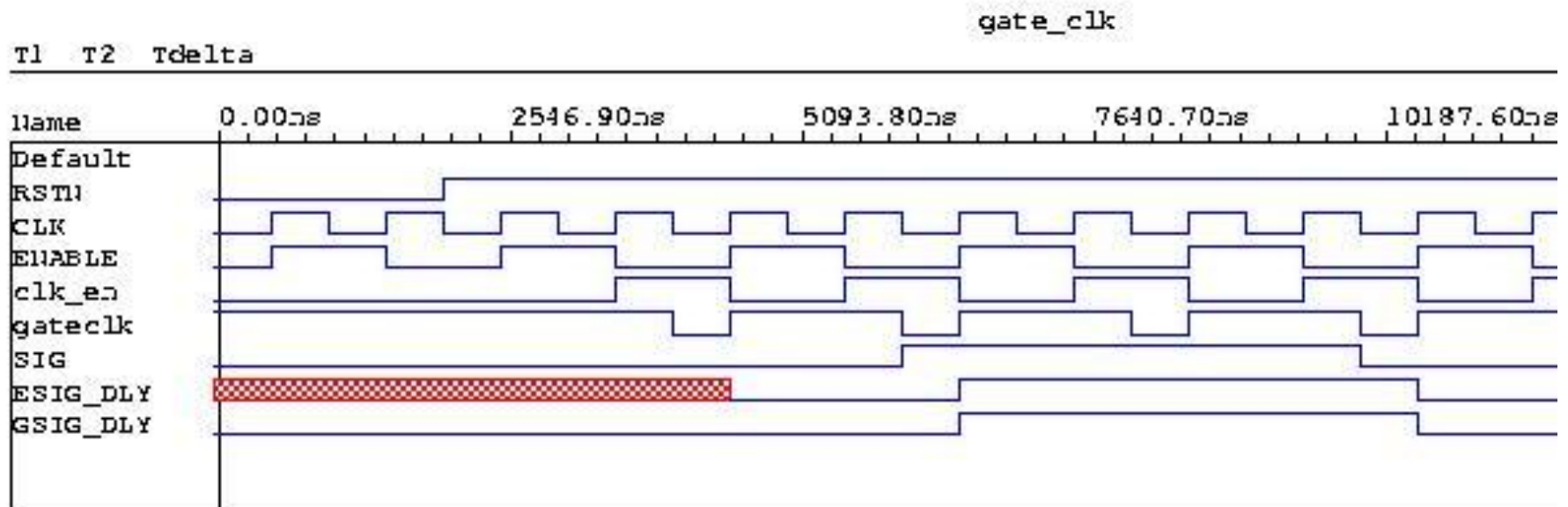
Clock enables versus gated clocks

- Avoid gated clocks unless required by power constraints.
- Gated clocks require careful design flow to avoid timing issues.
 - If “Scan Testability” is used then a gated clock disable is required to be inserted to allow non-gated clocking for test.
 - Latches should be used for gated clock enables.
 - Timing checks are needed to ensure race conditions don’t exist.
 - Numerous clock trees for gated clocks are cumbersome to place&route and minimize skew.
- Clock enables are simple to implement in code and design
 - major disadvantage of power consumption
 - slight increase in area over gated clocks due to mux flip-flops.
 - Simple single clock tree
 - Much simplified timing analysis

Clock enables versus Gated clocks



Clock enables versus Gated clocks



Tristate versus muxed busing

- For internal nets in small to medium size modules avoid tri-state muxing for signals, use logic muxes.
 - Bus holders needed for tri-state
 - More difficult place and route
 - Bus contention potential
- Tri-state muxing typical for core level integration with embedded memories, uP/uC, and cores.
 - Typical for “System on a Chip” and System Level Integration
 - Integration of peripherals to uP/uC databus for setup/control/data transfer/status.
 - Limit tri-state muxing to ~10 source/destinations for a single signal to control loading and slew problems.

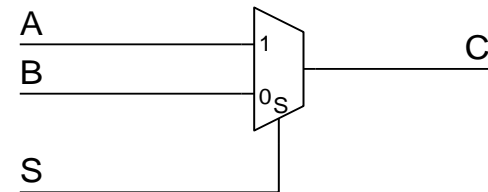
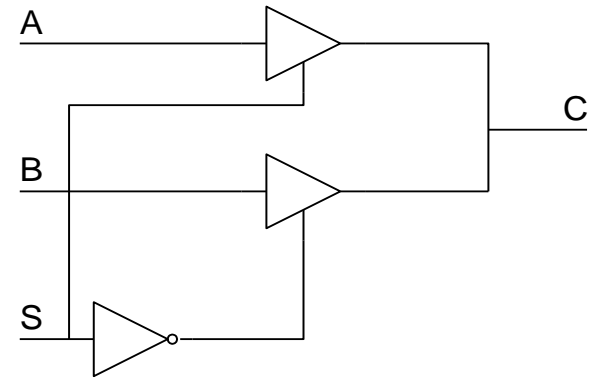
Tristate versus muxed busing

```
wire A,B,C,S;  
assign C = S ? A : 1'bz;  
assign C = ~S ? B : 1'bz;
```

“or”

```
wire A,B,S;  
reg C;  
always @(S or A)  
if (S) C = A;  
else C = 1'bz;  
always @(S or B)  
if (!S) C = B;  
else C = 1'bz;
```

```
wire A,B,C,S;  
assign C = S ? A : B;
```

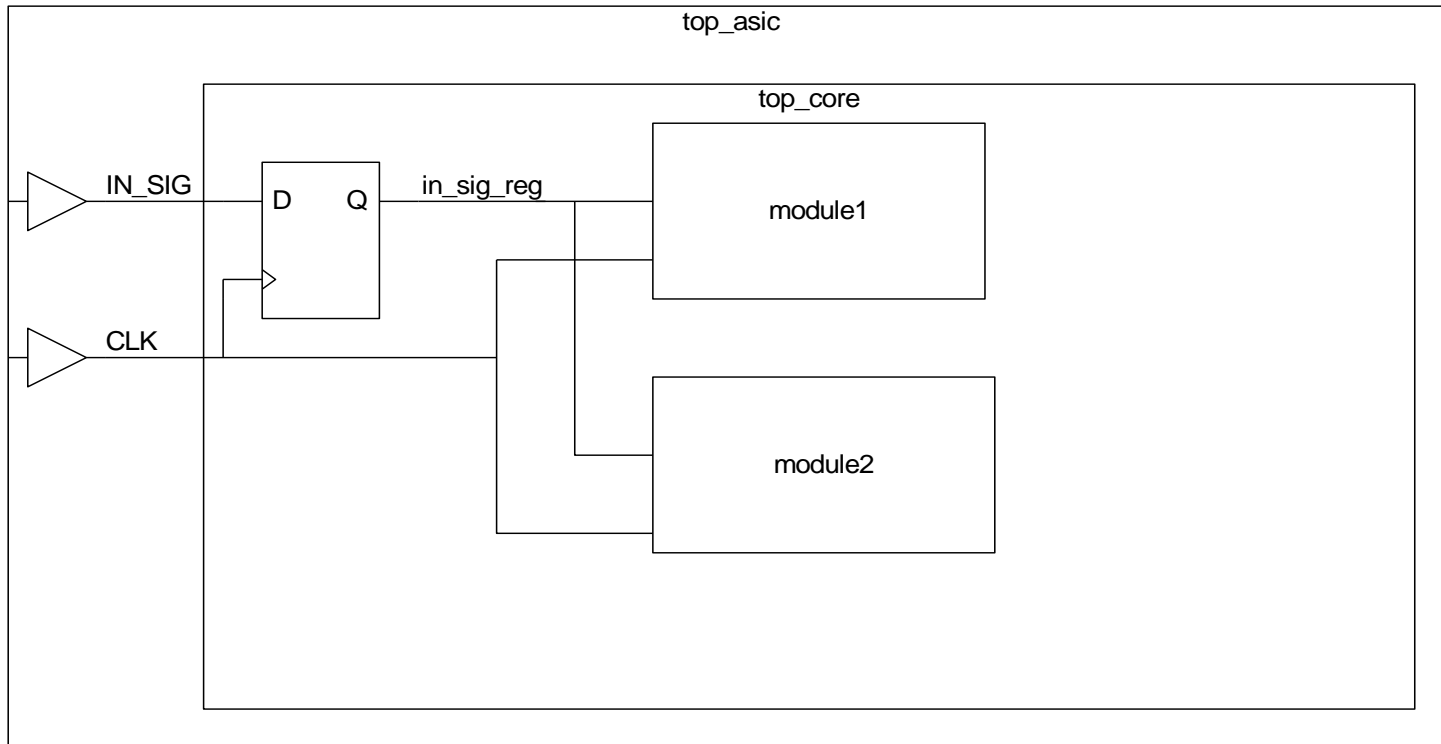


Inputs and Outputs

- Registering all top design level inputs and outputs makes for a cleaner design for timing and meeting requirements.
 - Not always possible for all interfaces.
 - asynchronous uP/uC address/data busses
 - UARTs
 - Serial interfaces
 - Clocked inputs and outputs have only one timing reference (clock) making specification.
 - Setup and Hold requirements easier to design and test
 - Propagation delay requirements easier to design and test
 - Faster designs

Inputs and Outputs

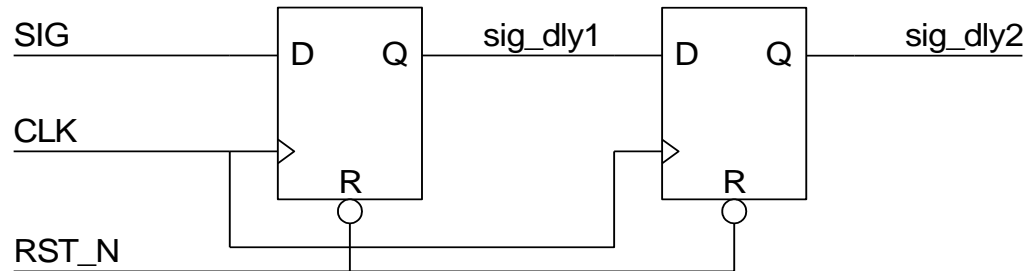
- Use only one register to clock in a single input signal.
 - Eliminate risk of clocking in two different logic values for asynchronous signals.
 - Quite often one chip input goes to multiple modules in hierarchy.



Interfacing asynchronous signals

- Double registering of asynchronous signals synchronizes the signals to the design clock.

```
reg sig_dly1, sig_dly2;
always @ (posedge CLK or negedge RST_N)
if (!RST_N) // async reset
begin
sig_dly1 <= 1'b0;
sig_dly2 <= 1'b0;
end
else // double register SIG
begin
sig_dly1 <= SIG;
sig_dly2 <= sig_dly1;
end
end
```



- For interfacing multiple control signals between two non-phase locked clocks use gray-scale coding techniques to ensure accurate bit level changes.
 - Gray-scale coding has only one bit changing for any bus.

Asynchronous set and clear

- Only use asynchronous set **or** clear for reset initialization. Do not use for other logic unless absolutely necessary.
 - Reset needed for control logic and state machines.
 - Do not use both clear and set pins on registers and latches
 - Unpredictable state

Logic Synthesis

- Introduction
- Typical Synthesis flow
- Synopsys, Ambit, Exemplar
- Goals and Constraints
- Scripts
- Timing
- Electrical
- Compile Directives
- Reports
- Read and Write
- Test
- Example
 - Script
 - HDL Code
 - Block Symbol
 - Schematic

Introduction

- Logic synthesis is the compiling of HDL code into a logic gates.
 - The primary goal is reduce the amount of effort and time to achieve the goals (area,speed) set forth to the design.
 - Time to market greatly reduced
 - Number of gates designed per day greatly increased
 - Larger designs possible with fewer designers
 - Logic verification and validation efforts are greatly reduced since boolean mistakes are eliminated, however coding errors and system specifications errors are possible
 - Most synthesis tools very rarely made any errors transforming HDL into gates
 - Designer concentrates on design aspects versus boolean equivalency of logic and system functions.
 - Some synthesis tools can also incorporate test logic into gate level netlist and Automatic Test Pattern Generation (ATPG)

Introduction

- Design changes are simplified by having to only re-code and re-synthesize HDL.
- Synthesis can also translate an existing gate level netlist into a different technology.
 - Very useful for old technologies becoming obsolete
- A variety of “target” gate logic can be used.
 - FPGA, Gate Array, or Standard Cell
 - CMOS, Bi-CMOS, Bipolar, SiGe, or GaAs
 - A synthesis library for the synthesis tool is created for target technology that contains
 - Boolean logic description of logic gates
 - Timing information of logic gates
 - Loading (Fanin) of logic gate inputs
 - Drive (Fanout) of logic gate outputs
 - Size or area of logic gates
 - Wireload model for interconnect wire delays
 - Target vendor creates library for synthesis tools

Introduction

- The input data into the Synthesis tools consists of the following.
 - HDL “source” code to be synthesized from
 - Gate “target” library to be synthesized into
 - Synthesis “script” file which contains compile directive for synthesis tool
- The output data from the synthesis consists of the following
 - Gate level netlist of equivalent boolean representation of HDL code
 - Binary database which represents the design and constraints.
 - Analysis reports of design
 - Warnings and Errors of design (open/shorted wires, async. feedback, error in code syntax,...)
 - Design reports
 - Timing
 - Area
 - Cells used
 - Standard Delay Format (SDF) file for pre-place&route timing estimation.
 - Schematic drawing of design
 - Good for block diagrams, debug and checking

Synthesis Strategy

- In general a hierarchy of HDL modules with a total of 10-20K gates can be synthesized with most tools.
 - Synthesis can be performed hierarchy or flatten (remove hierarchy).
 - All of the Verilog files from the current level of hierarchy are read in and synthesized together.
 - Write synthesis and constraint scripts which will reflect interface timing with other blocks and I/O in design.
 - Using register outputs makes interfacing and timing analysis simpler.
 - Register inputs to
 - Synthesis individual blocks in design then perform top-level synthesis which links together gate level netlist.
 - Top level synthesis typically does not perform any logic synthesis, only electrical and timing checks.

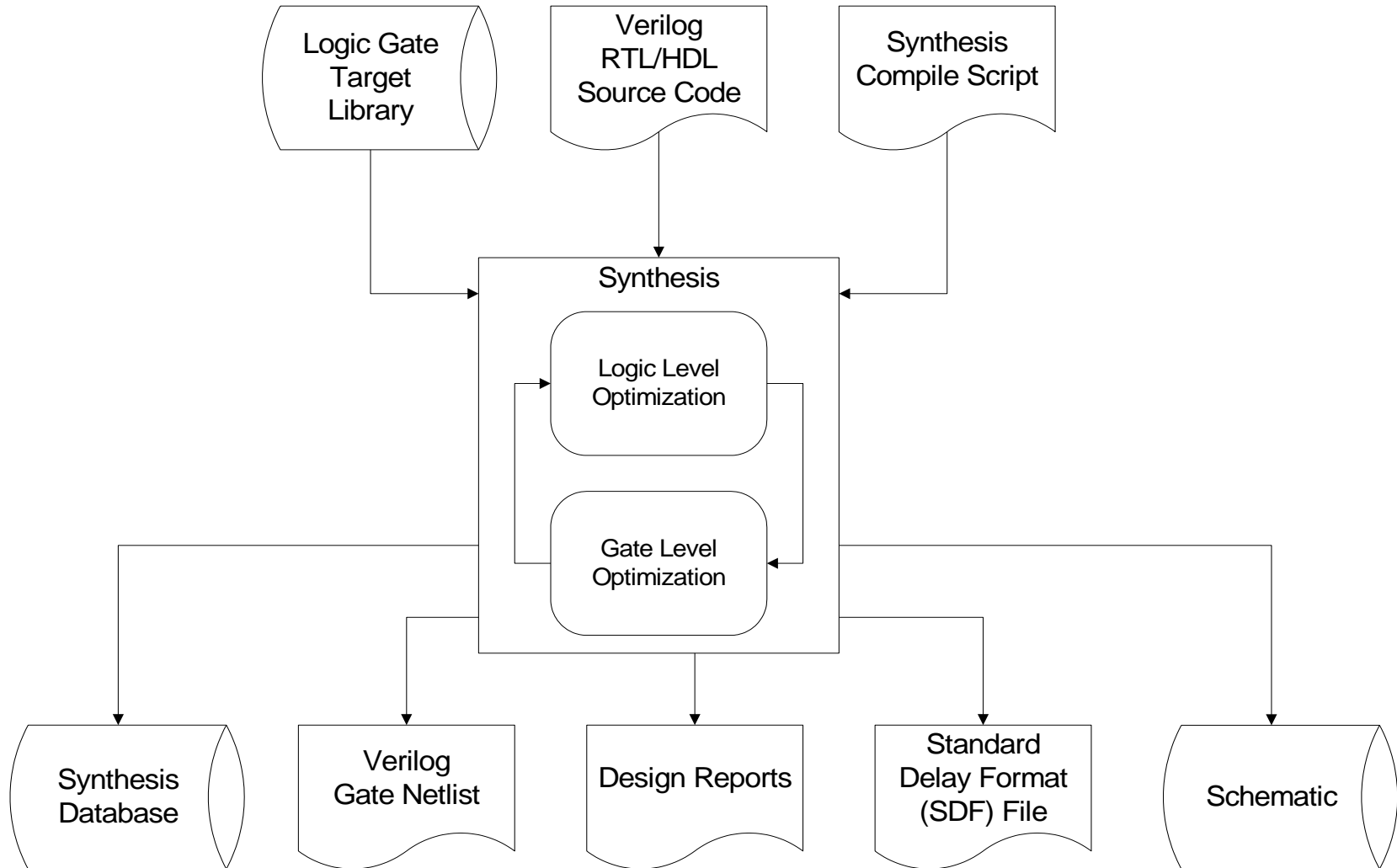
Synthesis Strategy

- Clocks are typically not synthesized.
 - A clock tree synthesis tool in place&route builds a clock tree with buffer cells from library
 - Control of placement yields better clock skew and path delay
- Memory are typically not synthesized.
 - A RAM/ROM compiler will generate a custom layout used in place/route using
- Input/Output buffers are not synthesized
 - Select from ASIC/FPGA offerings to meet external interface requirements (speed, load, levels)

Synthesis Tools

- Some of the synthesis tools in the EDA market.
 - Synopsys Design Compiler - Most popular ASIC compiler on the market, industry leader with 14+ years. Unix workstation based.
 - Cadence Ambit - Recent entry into high-end ASIC synthesis market. Unix workstation based.
 - Mentor Graphics Exemplar - FPGA/ASIC synthesis tool for low-end market. PC and Unix workstation based.
 - Synario Simplicity -FPGA synthesis tool. PC based.
 - Synopsys FPGA Express - FPGA synthesis tool. PC based.

Synthesis Flow

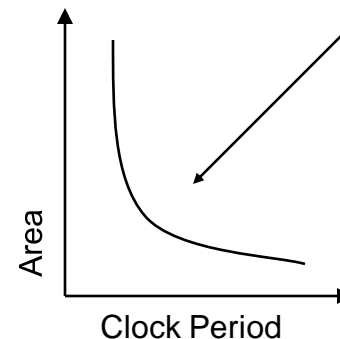
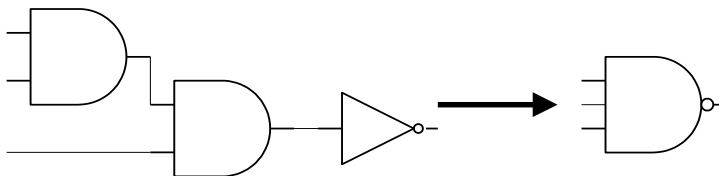
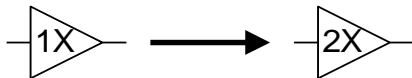


Goals and Constraints

- Goals and Constraints are used by the synthesis tools to achieve not only the boolean equivalent of the HDL code but to also achieve the timing and electrical requirements needed for the design.
 - After initial mapping of HDL into boolean equivalence, the synthesis tools map the equivalence into gates which have a intrinsic delay (logic delay).
 - The gate count is matched to a “wireload” model which adds output loading to gates (increasing delay) and interconnect “wire” delay (increasing delay).
 - The “wireload” model is a pre-place&route approximation of the design in silicon.
 - Actual net loading and delays will not be known until place&route.
 - Wireload models are typically more conservative than actual place&route timing.
 - Design Rule Checks (DRC) are made to ensure design does not violate loading and drive capacity.
 - Net loading and slew (rise/fall) time.
 - The delays are summed and checked against the timing constraints in the synthesis script which describe the timing of the design.

Goals and Constraints

- In addition a gate count or area is totaled for the design and compared against gate count goal (if any) in synthesis scripts.
- If timing/area goals or electrical rules are not met the synthesis tool restructures and re-maps into a new gate level netlist. This is repeated until goals and constraints are met or CPU time limit reached. Re-mapping usually involves
 - Increasing drive strength of gates
 - Using more parallel approach to boolean function to reduce number of gates in critical timing path.
 - Meeting timing constraints and electrical rules typically increases gate count/area. Timing has higher priority over area.



Goals and Constraints

- If goals can still not be met other synthesis technique can be used to optimize design.
 - “Flatten” design hierarchy to minimize common logic terms.
 - Increase CPU effort for synthesis run.
- Last resort is to modify HDL Code for better timing performance.
 - Change logic and system design to increase performance.
 - Less combinatorial logic between registers
 - Lower clock speed.
 - Introduce pipeline for arithmetic stages.
 - Use “gray-scaling” or “one-hot” state machines.
 - Use carry look-ahead logic for counters and adders.

Scripts

- Synthesis scripts contain all the design information and directives needed for the synthesis tool to compile the HDL code into gates
 - Design constraints
 - Design timing
 - Clock period and skew
 - Input and Output timing
 - Design Electrical characteristics
 - Input and Output loading
 - Wireload model and type
 - Operating conditions (process, temperature, voltage)
 - Best/Typical/Worst case process
 - Military/Industrial/Commercial conditions
 - Area
 - Don't touch/use directives
 - Clock nets
 - Reset nets
 - reserved cells for scan insertion (scan F/F)
 - Low output drive.

Scripts

- Compile directives and properties
 - Read in HDL and Library files
 - Check and analyze design
 - Link design(s)
 - Set constraints
 - Set compile directives
 - Compile design
 - Analyze gate level design
 - Write gate level netlist and design database
 - Write design reports
- **Note:** All examples for scripts in presentation use Synopsys Design Compiler commands

Timing

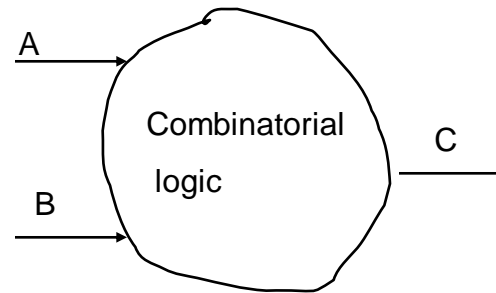
- Timing is described to the synthesis tools for the inputs/outputs and the design itself. Timing relationships are described with respect to signals that then become the timing constraints for the design.
 - Asynchronous Paths
 - inputs to outputs
 - Clocks
 - clock period
 - clock skew
 - multi-cycle relationship
 - Synchronous Paths
 - input to clock
 - clock to output
 - False Paths
 - allows no timing constraints on selected paths

Asynchronous Paths

- Set maximum and minimum path delays through asynchronous ports in design.
 - Care must be used in “min_delay”

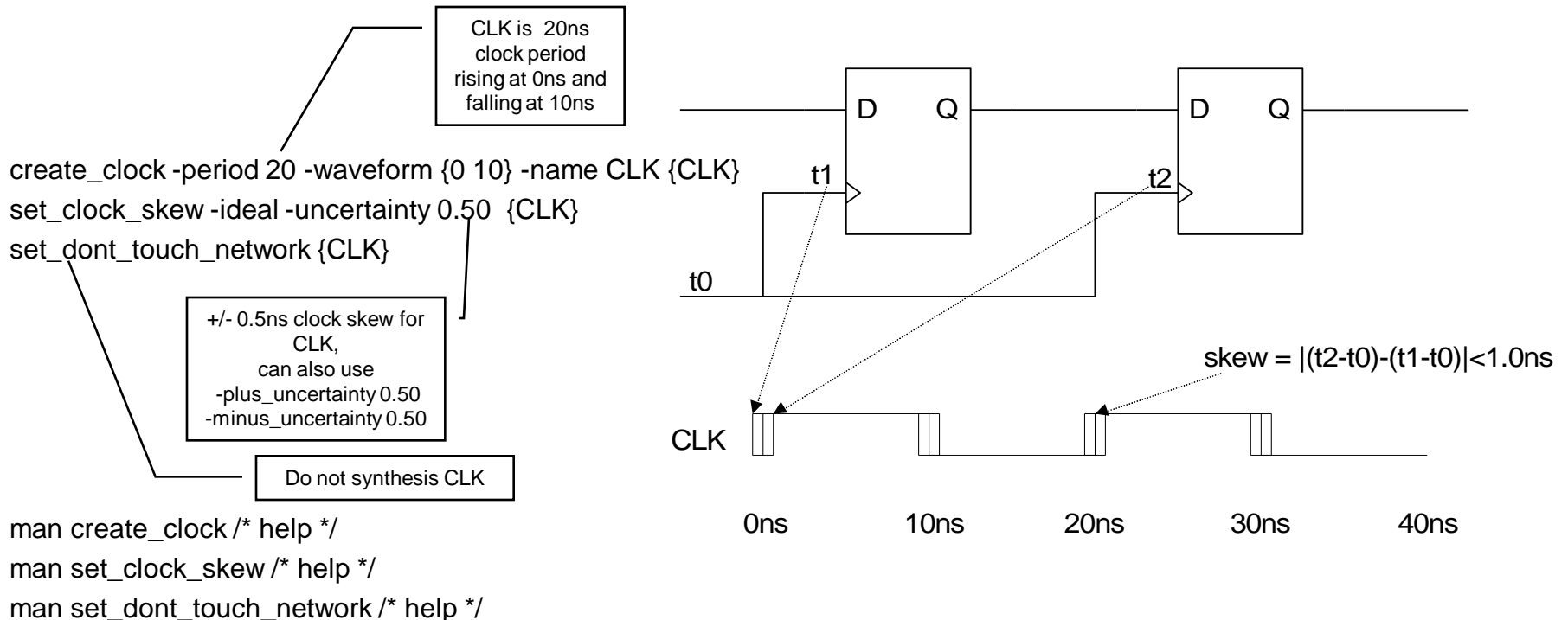
```
max_delay 10 -from A -to C
max_delay 5 -from B -to C
min_delay 2 -fall -from B -to C
min_delay 3 -rise -from B -to C
```

```
man min_delay /* help */
man max_delay /* help */
```



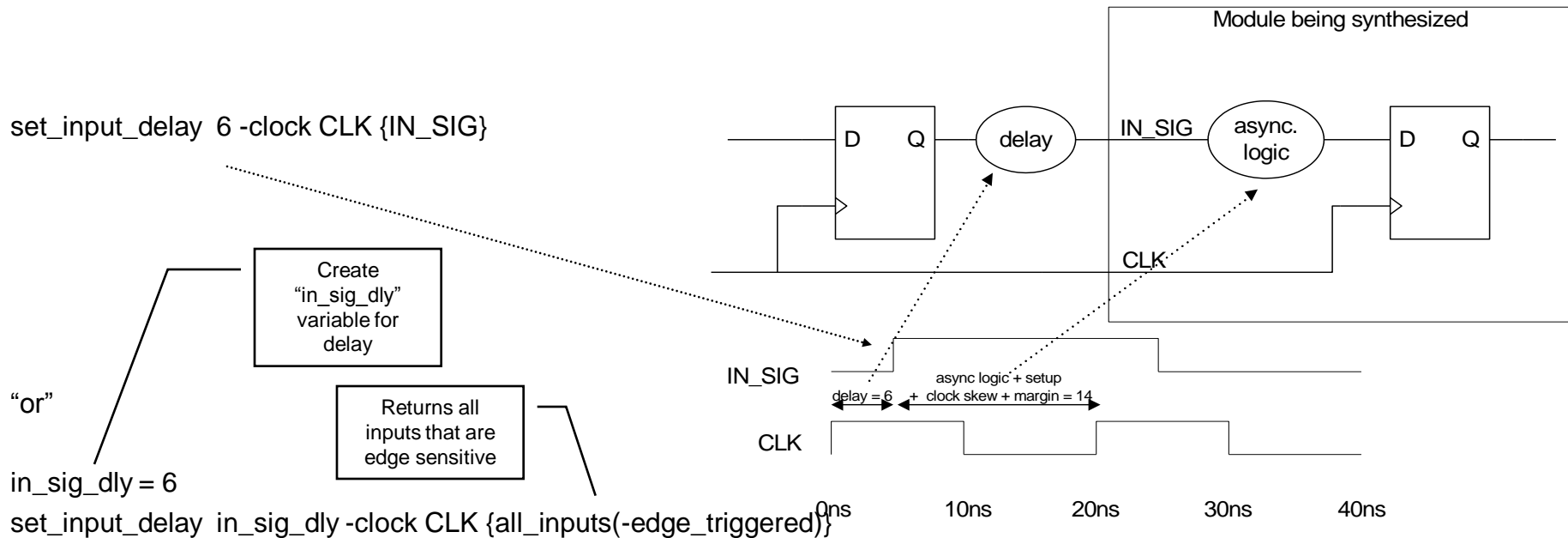
Clocks

- Clocks are described with respect to actual operational environment.
 - Clock period and waveform
 - Clock skew estimated from clock tree synthesis tools.
 - Clock nets are typically not synthesized



Synchronous Paths - Inputs

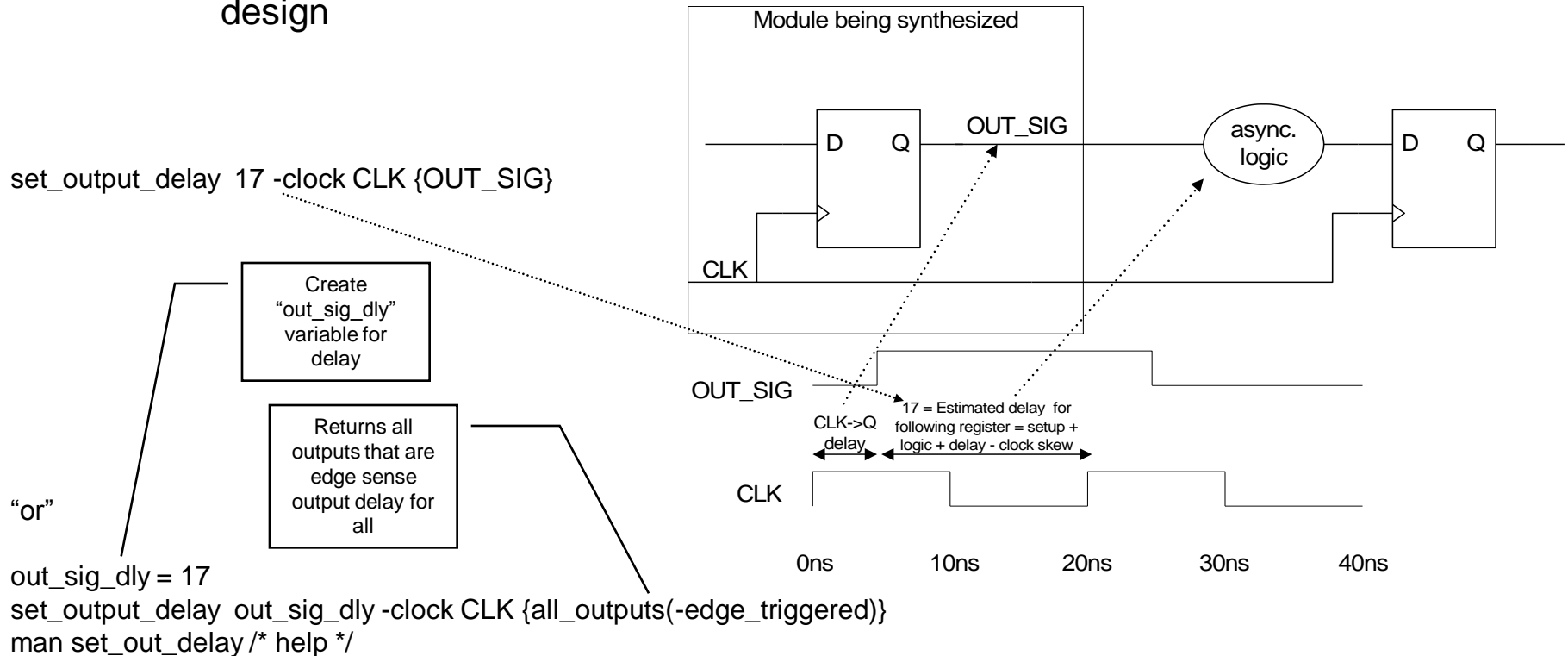
- Input delays are described with respect to clock signal.
 - Delay is with respect to previous edge of input clock
 - Estimate of input signal delay from source (not needed for higher hierarchy synthesis)
 - clk->q delay and interconnect delay for typical synchronous design



man set_dont_touch_network /* help */

Synchronous Paths - Outputs

- Output delays are described with respect to clock signal and following destination.
 - Delay is with respect to setup needed for next edge of clock
 - Estimate of output signal delay (not needed for higher hierarchy synthesis)
 - setup of destination register, logic, and interconnect delay for typical synchronous design



False Paths

- Some paths in logic are not crucial for timing or operation. It is advantageous for static signals which are not performance related to be declared “false paths” which have no timing constraints placed upon them.
 - Test scan and JTAG signals
 - Reset signals
 - uP/uC control registers

```
set_false_path -from RSTN -to all_outputs()
```

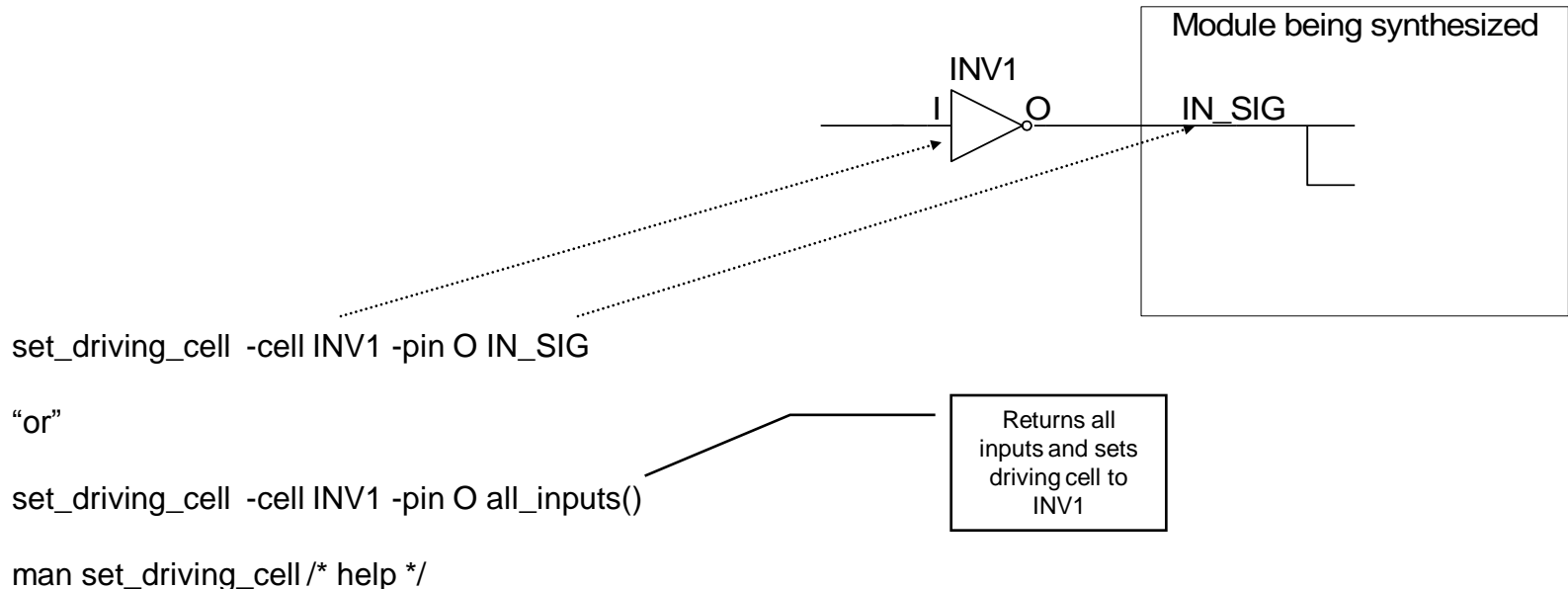
```
man set_false_path /* help */
```

Electrical

- Electrical parameters which affect timing and design rules are described to the synthesis tools for the inputs/outputs and the design itself.
 - Input loading
 - Output loading
 - Wireload
 - Transition

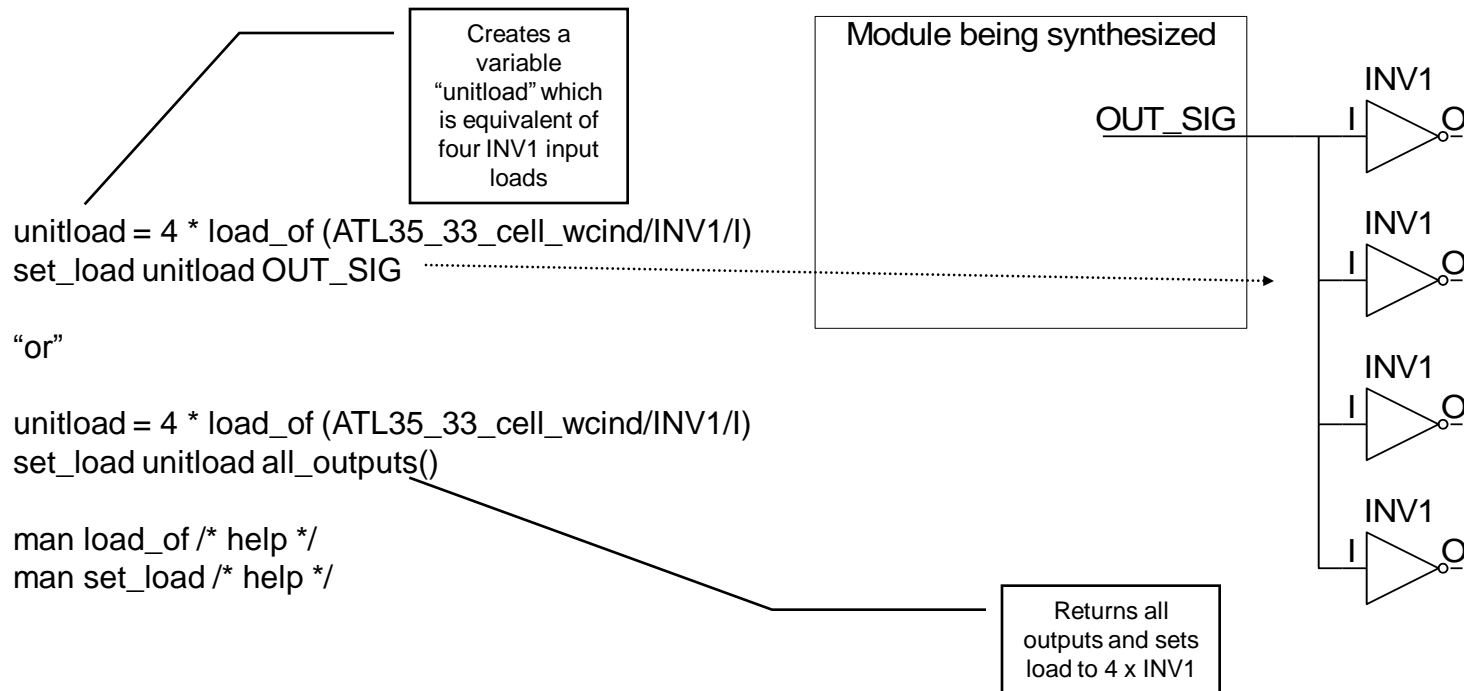
Input Loading

- Drive strength of input signal is described to synthesis tool in order to estimate timing and provide necessary buffering.
 - Used at lower level hierarchy synthesis for estimate of input port and Bi-Di port drive strength. Usually not needed for higher level hierarchy synthesis (module drive strength known when integrating).
 - Use a simple gate output from target logic library as drive source



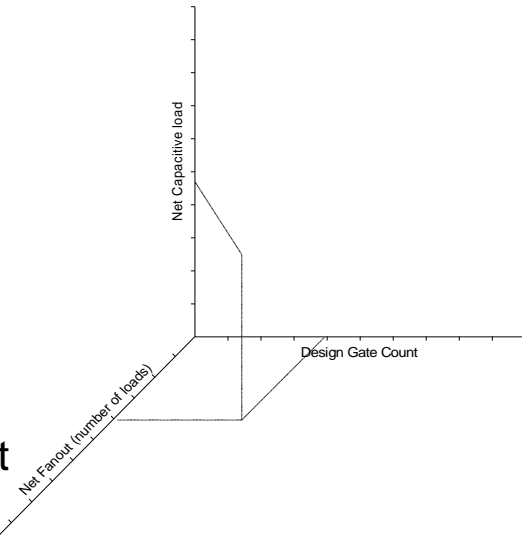
Output Loading

- Load of output signal is described to synthesis tool in order to estimate timing and provide necessary buffering.
 - Used at lower level hierarchy synthesis for estimate of output port and Bi-Di port drive strength. Usually not needed for higher level hierarchy synthesis (module drive strength known when integrating multiple modules).
 - Use a simple gate input from target logic library as a load destination



Wireload

- A wireload model is used to estimate the interconnect load and timing for nets in a given design
 - As HDL is synthesized into gates, the gate count compared to wireload table for appropriate loads and delays
 - The total number of gates for the design and number of loads for each net determines the capacitance for each net
 - The drive strength of the gate and capacitance are used to calculate the delay
 - The delay is added to intrinsic gate delay of the each net
 - Gate type or drive strength are changed to met timing constraints and electrical rules
 - Wireload models can be automatically chosen through synthesis library or selected manually
 - Extremely important for adequately selecting proper gates and drive strength to meet post-route timing



Wireload

- Different modes of wireload are available for describing floorplanning. i.e. Synopsys definitions
 - “top” - The wire capacitance of all nets is calculated using the wire load model set on the top-level design. (most conservative for timing)
 - “enclosed” - The wire capacitance of each net is calculated using the wire load model set on the smallest subdesign that completely encloses that net. (most liberal for timing)
 - “segmented” -For each net that crosses hierarchical subdesigns, the wire capacitance is calculated for each segment of the net based on the wire load model set on the subdesign that contains that segment.

```
set_wire_load_model -name 25K  
set_wire_load_mode top
```

Use 25K wireload for design

```
report_wire_load          /* use to get available wireload models for selected library */  
man set_wire_load_mode /* help */
```


Transition

- Rise and fall time of signals should be limited to prevent long delays and increased power consumption.
 - Synthesis library typically has automatic limits set for transition time

```
set_max_transition 3.0 clk_en
```

```
man set_max_transition /* help */
```

Compile Directives

- Different synthesis compile directives and techniques are available to obtain different goals and objectives
 - Area
 - Uniquify
 - Don't Use/Touch
 - Flatten
 - Ungroup
 - Structure
 - Effort
 - Link
 - Characterize

Area and Uniquify

- A maximum area goal can be placed on the design. This goal is usually a secondary priority to timing goal when compiling for most synthesis tools (more important to meet timing goal over area).
 - Useful to reduce gate area after timing performance goals met
 - Area is in gate count or cell area

```
set_max_area 4000
```

```
man set_max_area      /* help */
```

- If there are multiple instantiation of a module in a design they can be made unique for each instance.
 - Module synthesis is customized for each instance
 - Options for uniquifying certain cells or reference names

```
uniquify
```

```
man uniquify /* help */
```

Don't Use/Touch

- Often it is necessary to restrict certain cells in library from being used in synthesis.
 - Useful to restrict scan flip/flops and latches before first pass synthesis
 - Restrict low drive strength cells

```
set_dont_use { atl55_3_wcind/DSS* atl55_3_wcind/JK* atl55_3_wcind/DFFB* }
```

```
man set_dont_use      /* help */
```

- Gates and cells that are not synthesized should not be “touched” during synthesis to prevent their removal.
 - Input/Output buffers, IP, memories, clock trees, critical gate design, existing gate-level netlist

```
set_dont_touch {snapshot mult vhdl_logic}
```

```
man set_dont_touch    /* help */
```

Flatten and Ungroup

- To remove the design hierarchy use the “flatten” command which creates a flat netlist when compiling.
 - Useful to potentially make small reductions in timing and gate area
 - Lost of hierarchy makes gate level debug difficult

```
set_flatten true -design top_core
```

```
man set_flatten          /* help */
```

- To reduce the module hierarchy use the “ungroup” command which can remove a level of hierarchy.
 - Useful to potentially make small reductions in timing and gate area
 - Lost of hierarchy makes gate level debug difficult

```
ungroup uart
```

```
man ungroup             /* help */
```

Structure and Effort

- Logic structuring is addition of intermediate variables for which sub-functions of logic are produced. During logic synthesis these sub-functions are minimized in the most efficient logic.
 - On by default, only disable for no structuring in compile

```
set_structure -false
```

```
man set_structure      /* help */
```

- Compile effort control of the logic synthesis allows addition CPU time to further optimize design.
 - Useful to further reduce gate area if timing/area goals not met

```
compile -map_effort high /* low , medium , high */
```

```
man compile /* help */
```

Link and Characterize

- To resolve all design references before logic compilation, perform the “link” command.
 - On by default, only disable for no structuring in compile

link

```
man link      /* help */
```

- To capture information on the environment of specific cell instances and assign information as attributes on the design to which the cells are linked.

characterize

```
man characterize    /* help */
```

Reports and Checks

- A large variety of reports can be made for the design, library, and goals.
 - Consult synthesis documentation for further help
 - Static timing analysis can be performed by using report_timing commands

report_area
report_wire_load

report_lib
report_internal_loads

report_design
report_timing

report_clock
report_hierarchy

report_cell
report_annotated_delay

- Checks can be made for the design.
 - Consult synthesis documentation for further help

check_design

check_timing

check_test

Read and Write

- Data files can be read and written along with setting up search paths for locating files.
 - Multiple HDL files can be read in same script file
 - Outputs of reports can be “piped” unix style (>) into ASCII files
 - Variety of file formats for designs
 - HDL level (Verilog, VHDL)
 - Bit map lookup table (“.pla” - useful for synthesizing small ROM’s)
 - Gate level (Verilog, Vital/VHDL, EDIF)
 - Binary database (“.db” file contains all aspects of design)

```
read -format verilog “../src/timer.v”
```

```
write current_design -format db -output “../db/timer.db”
```

```
write current_design -format verilog -output “../gate/timer.v”
```

```
/* current_design is a Synopsys reserved variable */
```

```
/* current_design is a Synopsys reserved variable */
```

“or”

```
design = timer
```

```
read -format verilog “../src/” + design + “.v”
```

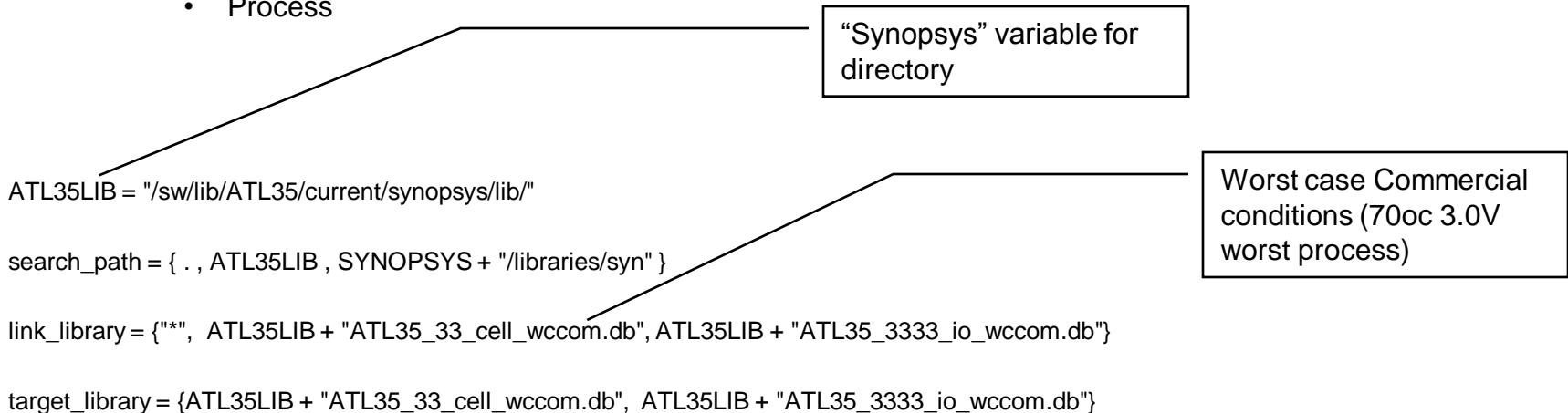
```
write current_design -format db -output “../db/” + design + “.db”
```

```
write current_design -format verilog -output “../gate/” + design + “.v”
```

Create variable “design”
and use for script

Libraries and Paths

- Libraries are specified for “link” (existing gate level netlist and IP) and “target” (desired gate technology).
 - Search paths are also specified similar to Unix \$PATH
 - Different libraries may be selected for different operational environment and process (dependent on silicon vendor). Use worst case conditions for environment for synthesis.
 - Temperature
 - Voltage
 - Process



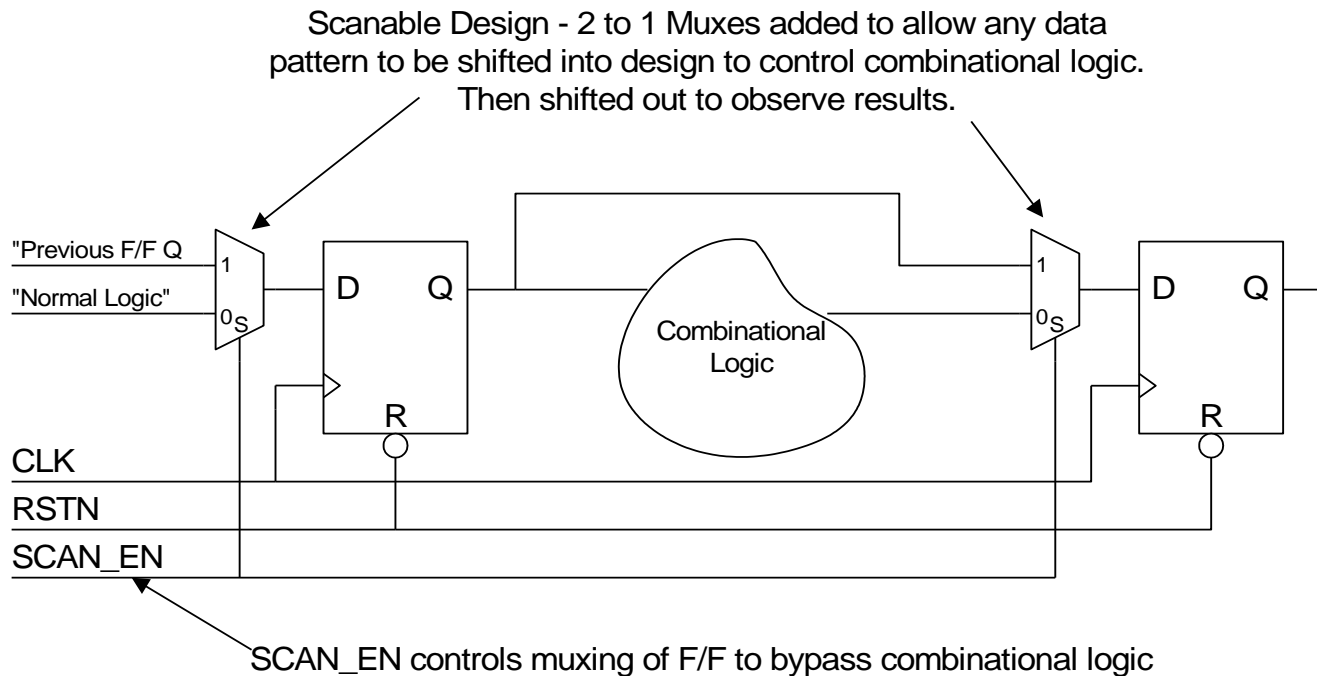
Libraries and Paths

- Single library may contain all operating conditions. Use “set_operating_conditions” to select conditions.

set_operating_conditions -max wccom

Test

- Synthesis tools can also insert test logic into the design using a “scan path”. This allows all of the registers and latches to be observable and controllable.
 - The “scan path” takes all of the registers in the design and puts a 2 to 1 multiplexer before the “D” input.



Test

- This greatly reduces the amount of time to make a design testable and produce test vectors
- General flow for scan test insertion
 - Test methodology (“set_test_methodology scan_test_implementation”)
 - Restrict scan flip/flops and latches from compile (“set_don’t_use ...”)
 - Synthesize design into gates (“compile ...”)
 - Check design for testability (“check_test”)
 - Re-enable scan flip/flops and latches (“remove_attribute ...”)
 - Insert scan path into design (“compile_test ...”)
 - Re-optimize design (“compile ...”)
 - Create test vectors (“create_test_vectors”)
 - Write out test vectors (“write_test ...”)
 - Write out test report (“report_test ...”)

Example Synthesis

- 10 bit down counter with 10bit pre-load register for clock divider
 - Active low asynchronous reset
 - Rising edge synchronous detect on load signal
 - Active high enable signal
 - Bypass mode for counter when counter disabled (clock output)
- Synthesized to Atmel ATL35 Gate Array technology
 - worst case industrial conditions
- Synopsys Script
- Verilog HDL
- Block Diagram
- Schematic

Example Script

```
/*****
** Setup Synopsys variables                                **
*****/

ATL35LIB = "/sw/lib/ATL35/current/synopsys/lib/"

search_path = { . , ATL35LIB , SYNOPSYS + "/libraries/syn" }

link_library = {"", ATL35LIB + "ATL35_33_cell_wcind.db", ATL35LIB + "ATL35_3333_io_wcind.db"}

target_library = {ATL35LIB + "ATL35_33_cell_wcind.db", ATL35LIB + "ATL35_3333_io_wcind.db"}

verilogout_higher_designs_first = true
verilogout_single_bit = false

/*****
** Read in all required Verilog files                      **
*****/

remove_design -all
design = counter10B
read -format verilog "./src/" + design + ".v"
current_design = design
```

Example Script

```
/*****
```

```
** Perform pre-compile check_design **
```

```
*****/
```

```
check_design > reports/check_design.pre
```

```
check_test > reports/check_test
```

```
/*****
```

```
** Clock definitions **
```

```
*****/
```

```
create_clock -period 10 -waveform {0 5} {CLK} -name CLK
```

```
set_clock_skew -ideal -uncertainty 0.50 {CLK}
```

```
set_dont_touch_network {CLK}
```

```
set_dont_touch {clk_invert clk_buffer}
```

```
/*****
```

```
** Timing definitions **
```

```
*****/
```

```
set_wire_load_mode top
```

```
set_wire_load_model -name 25K
```

```
set_input_delay 4 -max -clock CLK all_inputs()
```

```
set_output_delay 2 -max -clock CLK all_outputs()
```

```
set_false_path -from RSTN -to all_outputs()
```


Example Script

```

/*****
** Input port definitions                                **
*****/

set_driving_cell -cell INV1 -pin O { all_inputs() }

/*****
** Output port definitions                                **
*****/

unitload = 4 * load_of (ATL35_33_cell_wcind/INV2/I)
set_load unitload { all_outputs() }

/*****
** First pass compile                                    **
*****/

set_fix_multiple_port_nets -all
uniquify
set_flatten -design current_design

compile -map_effort medium

```

Example Script

```

/*****
** Save temporary database and Verilog netlist      **
*****/

write current_design -format db    -hierarchy -output "./db/" + design + ".db"
write current_design -format verilog -hierarchy -output "./gate/" + design + "_gate.v"

/*****
** Perform post-compile check_design, timing, and misc reports  **
*****/

check_design > reports/check_design.post
report_timing > reports/timing.rpt
report_cell > reports/cells.rpt
report_area > reports/area.rpt

quit

```

Example Verilog HDL

```

/*****
** 10 bit counter                               **
*****/
module counter10B (CLK,    // Rising-edge clock
                  RSTN,    // Active-low RESET
                  PRELOAD, // counter preload
                  EN,      // Active high counter enable
                  LOAD,    // Active high preload
                  CLK_DIV // counter divide by N
                  COUNT); // counter bits

parameter bw = 10 ,
          all0 = 10'd0,
          dec1 = 10'd1;

/*****
** Input signal declarations                     **
*****/
input  CLK;    // Rising-edge reference clock
input  RSTN;   // Active-low RESET
input  EN;     // Counter enable
input  LOAD;   // Active high load enable
input [bw-1:0] PRELOAD; // D preload value

/*****
** Output signal declarations                   **
*****/
output CLK_DIV; // Clock divided output
output COUNT;   // counter bits

```

Example Verilog HDL

```

/*****
** Wires and Regs                                     **
*****/

reg tc_reg,    // Terminal count register
  load_1,      // 1st LOAD shift reg
  load_2;      // 2nd LOAD shift reg
reg [bw-1:0] COUNT, // Q bus of counter
  reload; // Preload counter value

wire clk_inv, // inverted clock
  clk_buf,    // buffered clock
  CLK_DIV;    // Clock divided output

/*****
** gate instances and wire assign                       **
*****/

INV2 clk_invert(.I(CLK),.O(clk_inv)); // select inverter for clock pre-buffer
INV4 clk_buffer(.I(clk_inv),.O(clk_buf)); // select strong inverter for clock driver

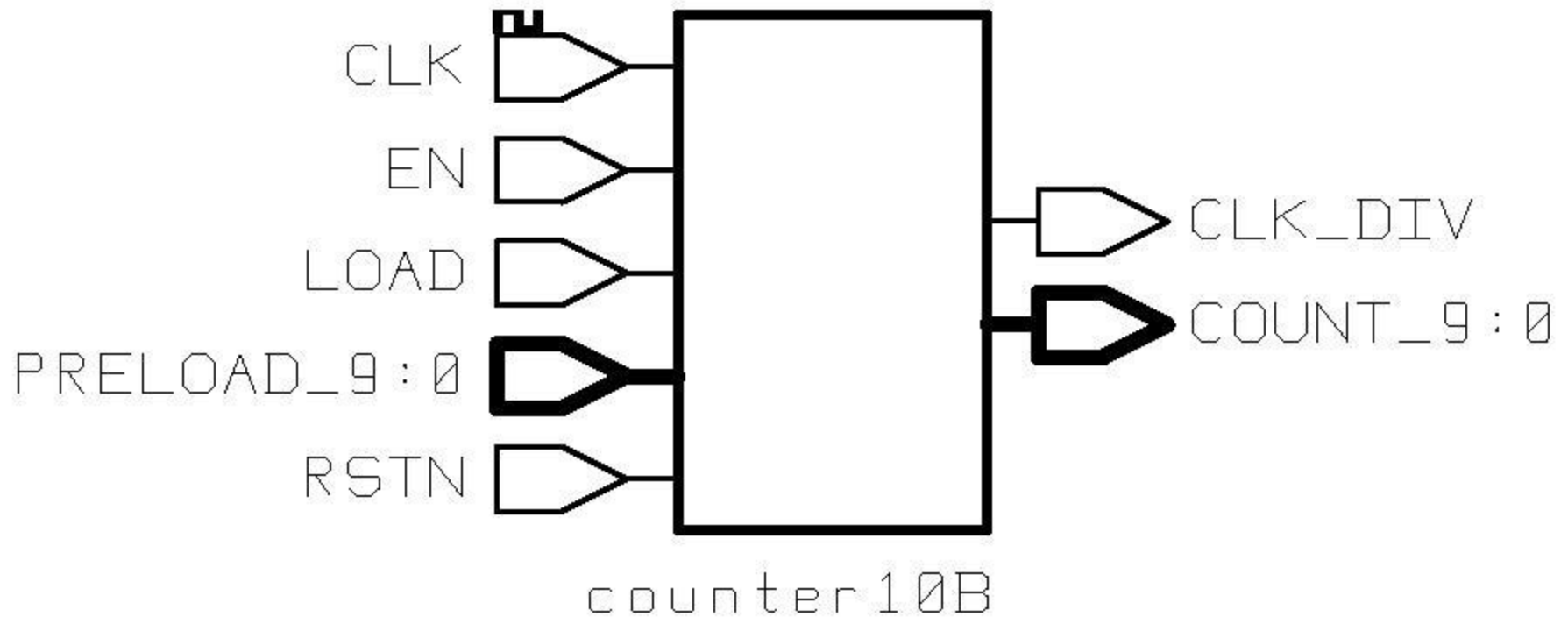
assign CLK_DIV = ~(EN ? ~tc_reg : clk_inv); // select clock or divided clock

/*****
** synchronous logic                                   **
*****/
```

Example Verilog HDL

```
always @(posedge clk_buf or negedge RSTN)
  if (!RSTN)
    begin // clear all reg's on negedge reset
      tc_reg <= 1'b0;
      load_1 <= 1'b0;
      load_2 <= 1'b0;
      reload <= all0;
      COUNT <= all0;
    end
  else begin
    tc_reg <= (COUNT == all0); // registered terminal count
    load_1 <= LOAD; // sync up LOAD signal
    load_2 <= load_1; // 2nd sync of LOAD signal
    if (load_1 & ~load_2) // rising edge of sync'ed LOAD
      begin
        reload <= PRELOAD; // store reload on rising edge of LOAD signal
        COUNT <= all0; // clear counter
      end
    else if (EN) // if enabled
      begin
        if (COUNT == all0) // if terminal count
          COUNT <= reload; // preload counter
        else
          COUNT <= COUNT - dec1; // else decrement counter
      end // if (EN)
    else
      COUNT <= all0; // else no counter activity for no enable
  end
end
```

Example Block Symbol



Example Schematic

