Synthesizable Verilog-HDL Code

Huai-Yi Hsu
yuki@access.ee.ntu.edu.tw

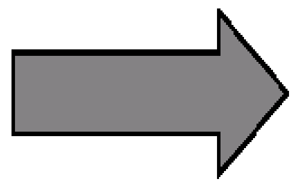
Oct. 15, 2001

Outline

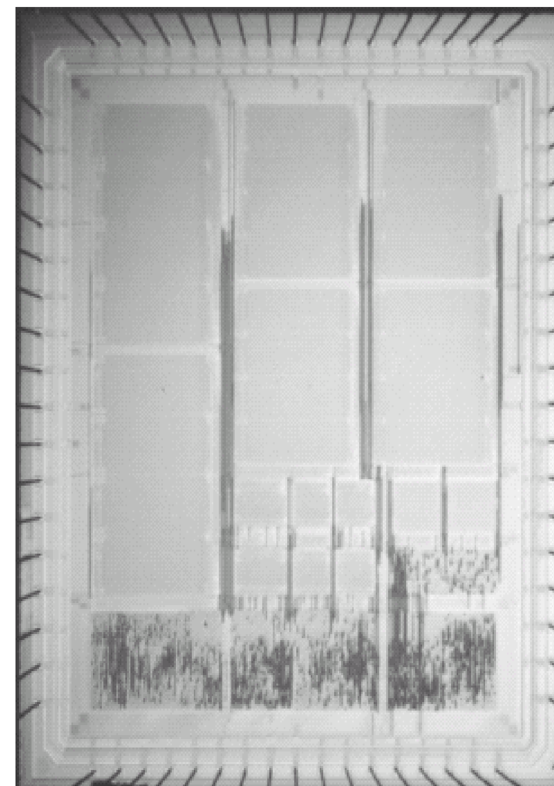
- Introduction
- Verilog-HDL Circuit Design
  - Behavior Level
  - Register-Transistor Level
  - Gate Level
  - Circuit Level
- Synthesis
- Coding Style

IC Design and Implementation

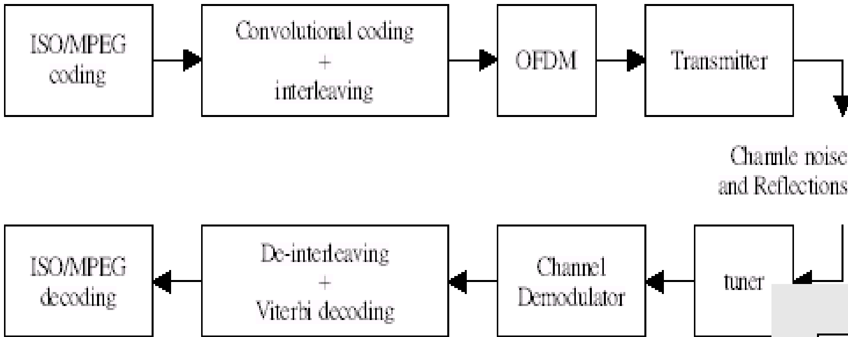
Idea

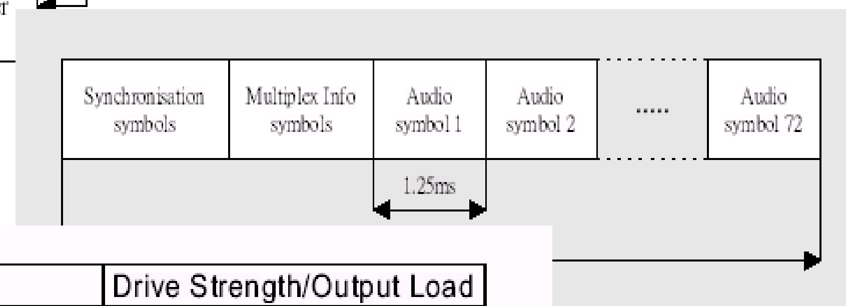


Chip

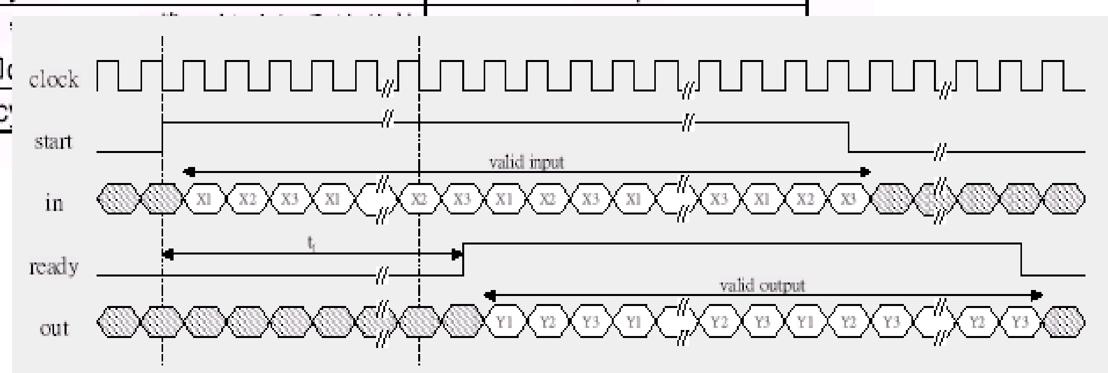


System Specification

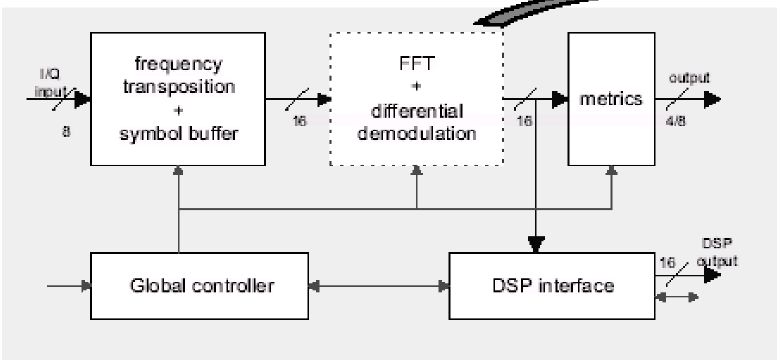




腳位名稱		描述	Drive Strength/Output Load
clk	輸入	系統時脈	assume infinite
reset	輸入	系統重置訊號，high active	1 ns/pf
din	輸入	每個clock cycle輸入一個16-bit 正整數	1 ns/pf
ready	輸出	reset為1時 … 出前的半個 …	
dout	輸出	每個clock c…	



Algorithm Mapping

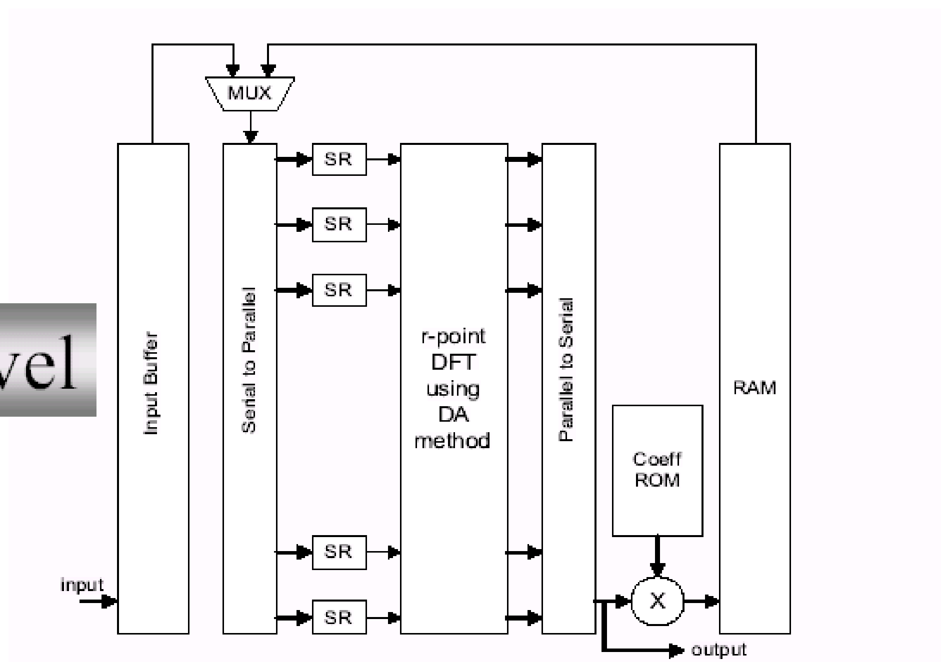


System Level

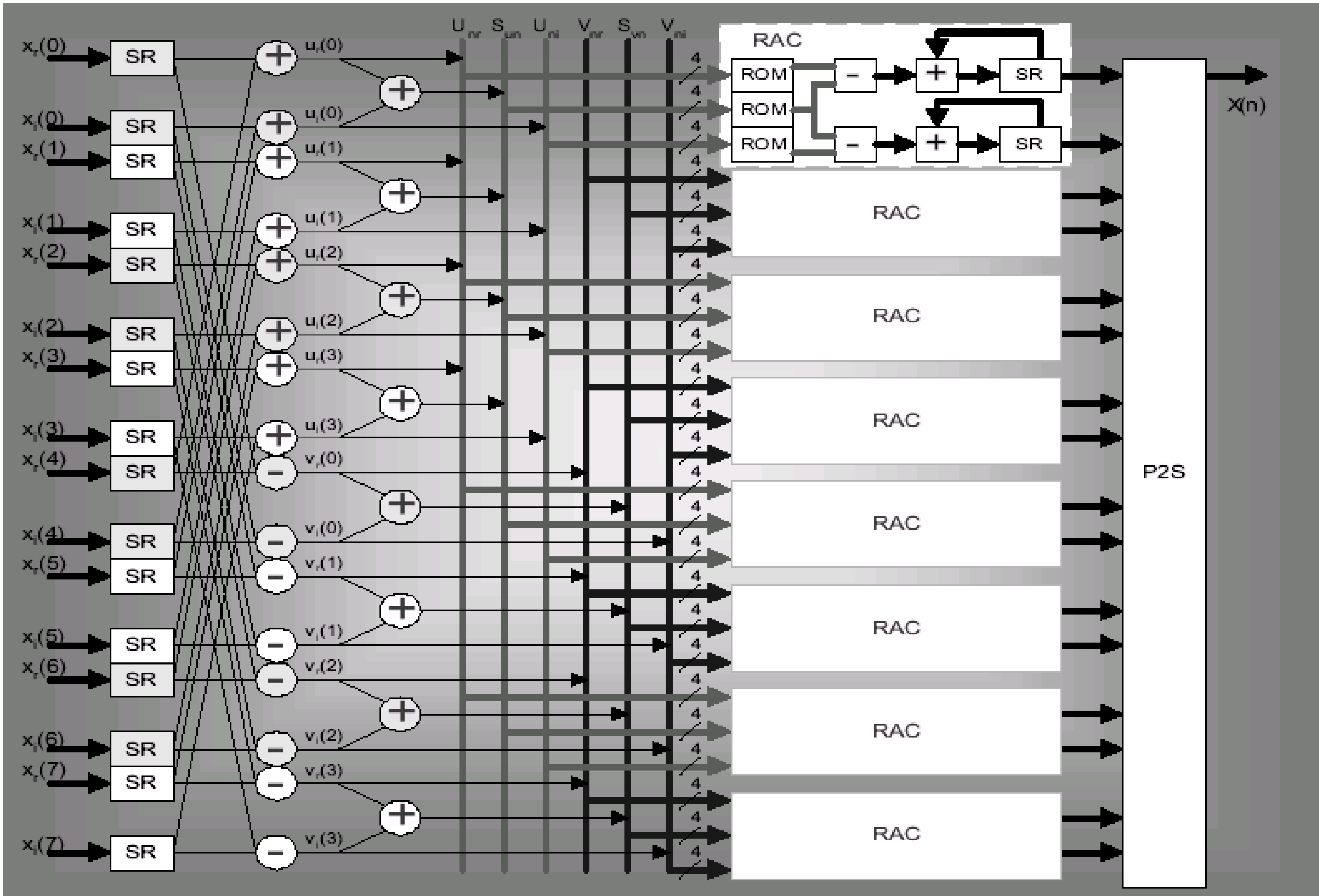
$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk} \qquad k = 0,1,\ldots,N-1$$

$$X(k) = \sum_{n=0}^{N/2-1} x(2n) W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x(2n+1) W_{N/2}^{nk}$$

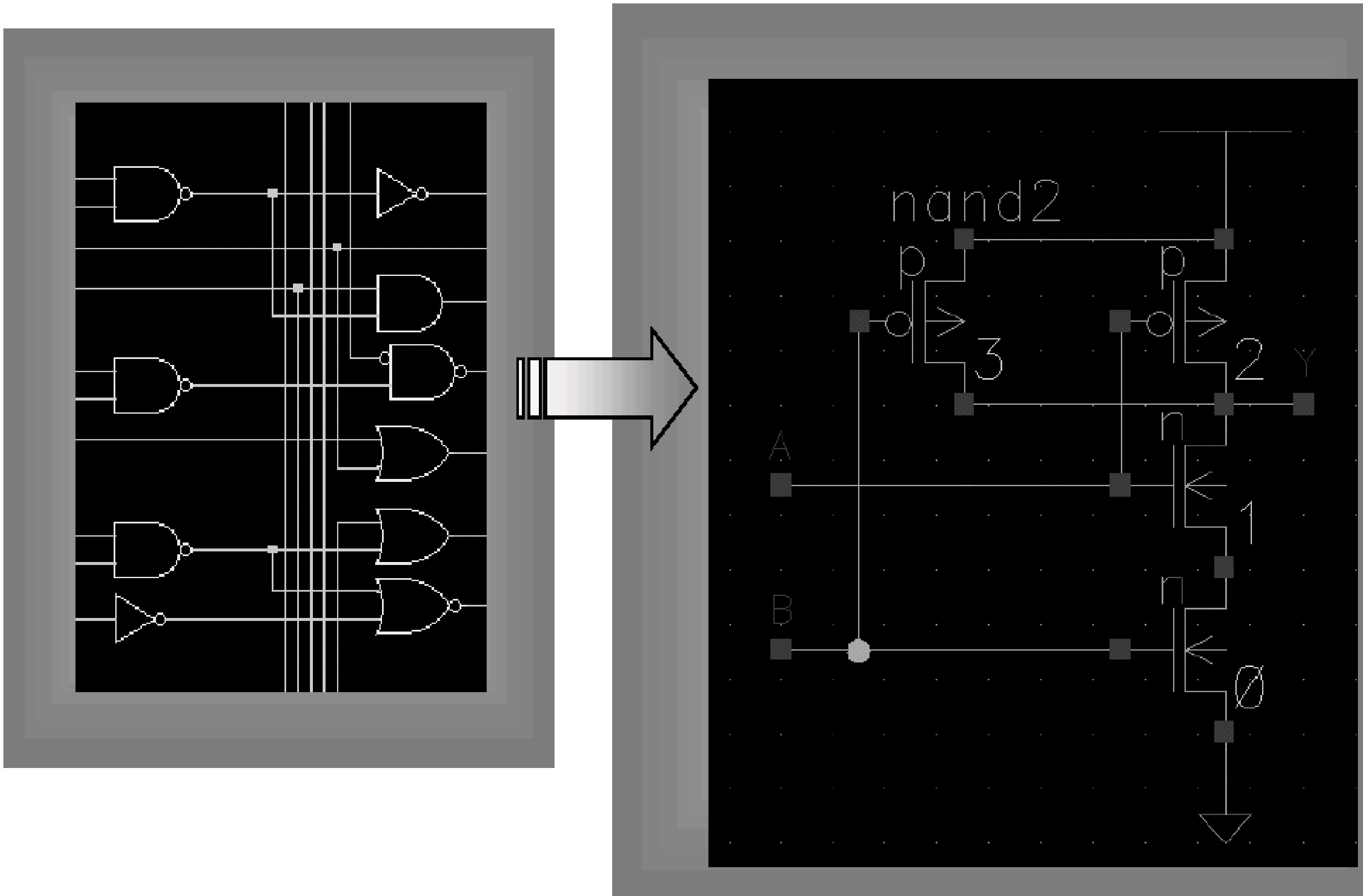
RTL Level



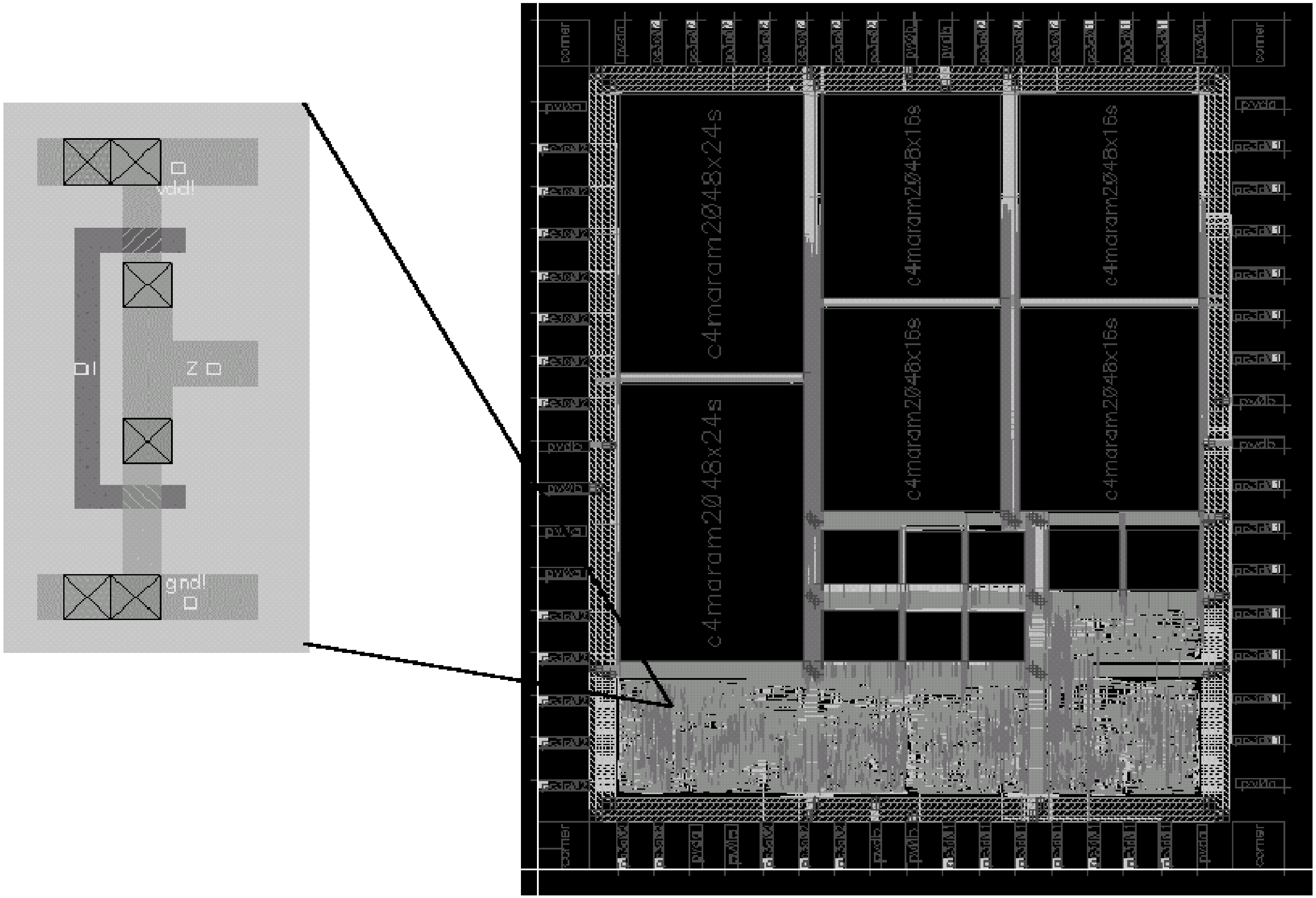
Hierarchy Design



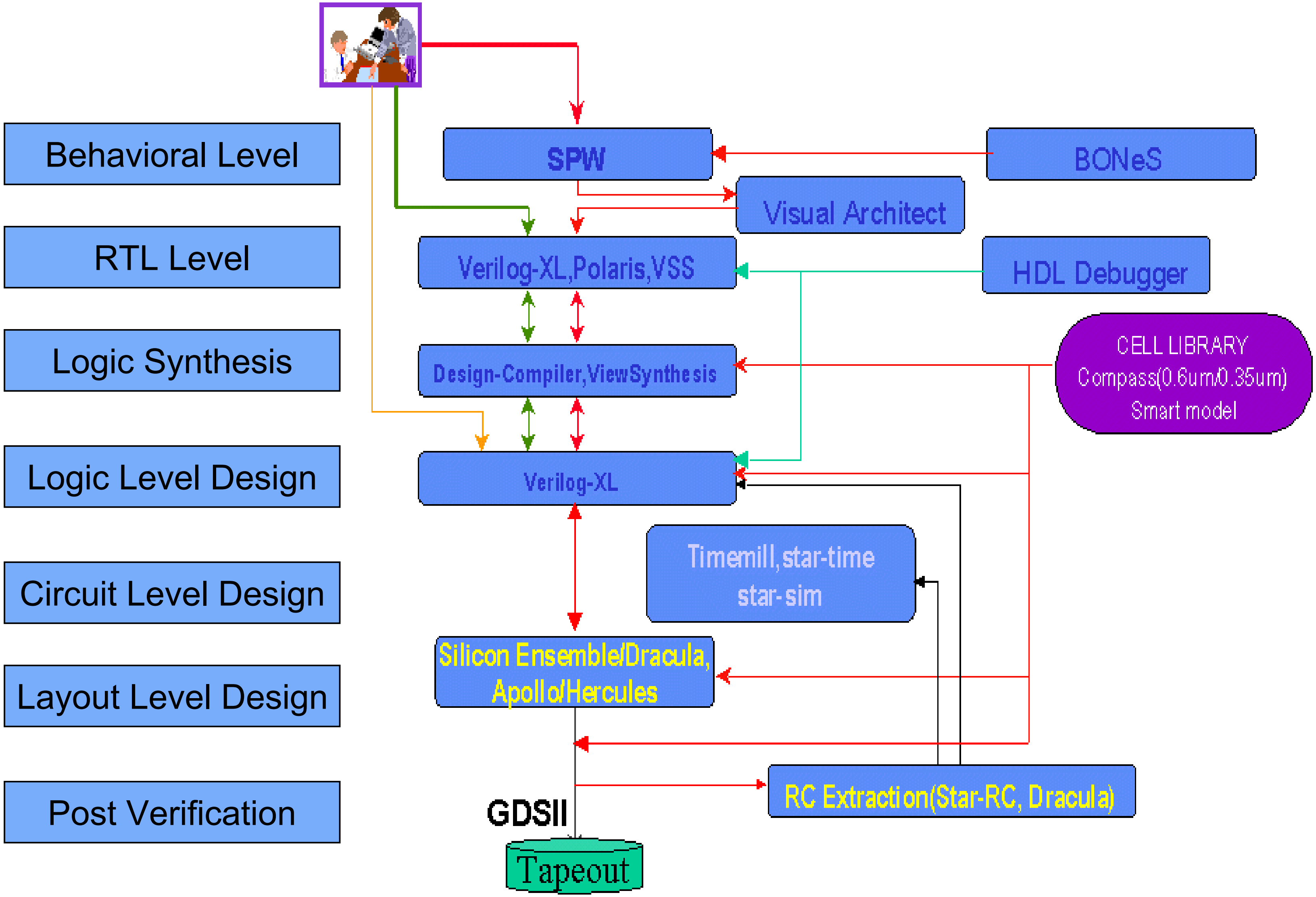
Gate and Circuit Level Design



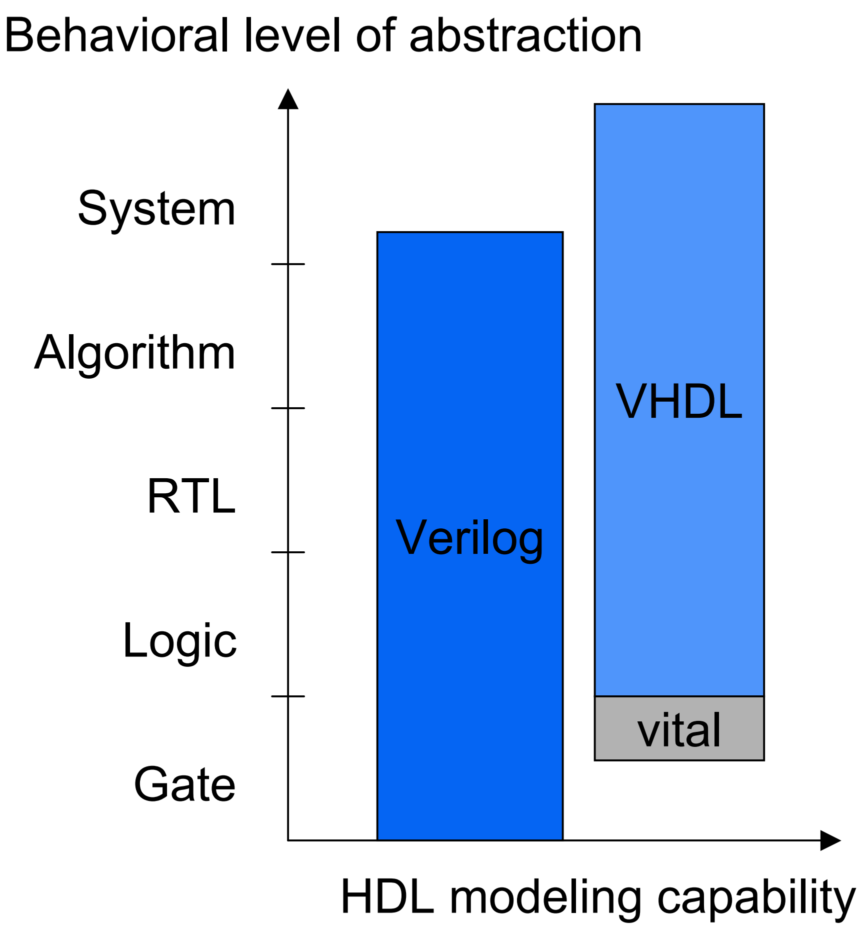
Physical Design



Cell Based Design Flow

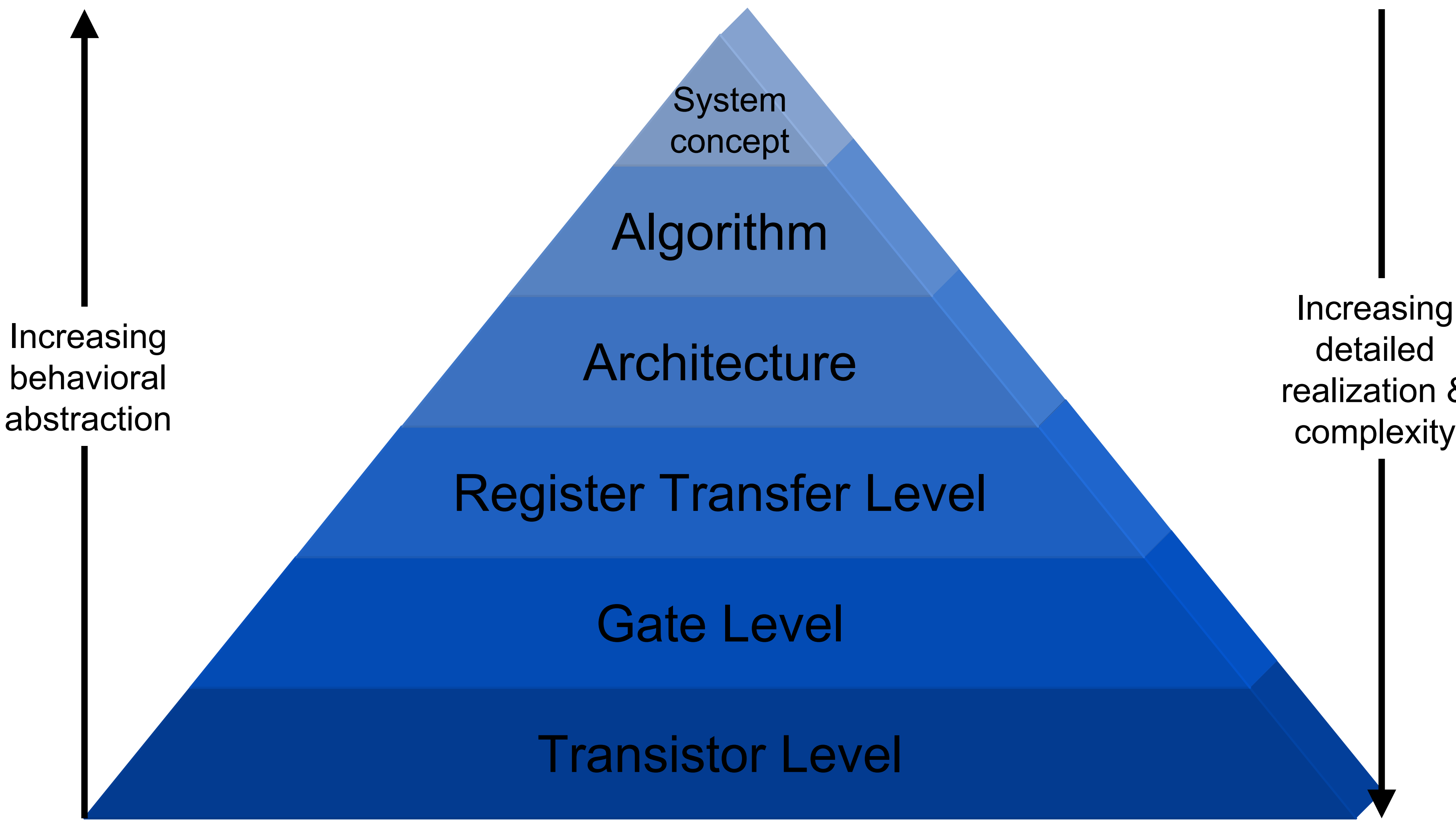


Verilog-HDL vs. VHDL

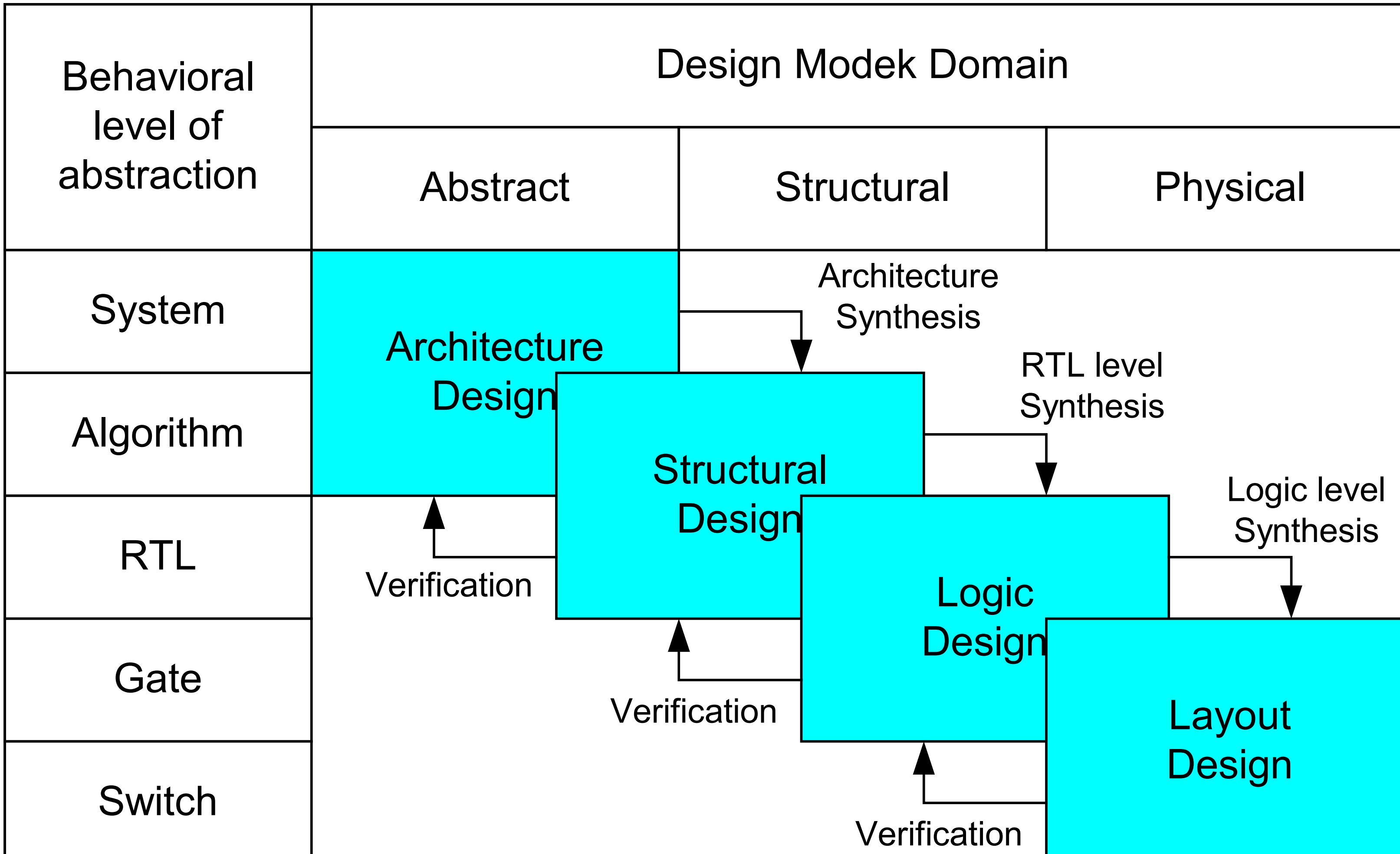


	Verilog	VHDL
Compilation	interpretative	Compile
Libraries	No	Yes
Reusability	`include	Package
Readability	C & ADA	ADA
Easy to Learn	Easy	Less intuitive

Behavioral Level



Design Domain



Overview of Verilog Module

module module_name (port_name);

port declaration

data type declaration

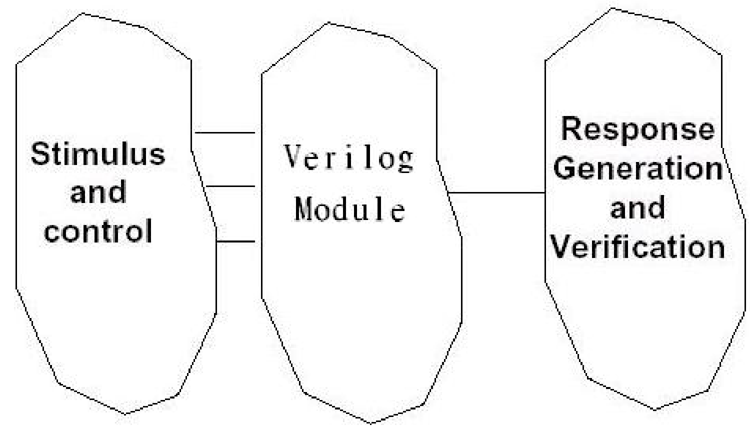
module functionality or structure

endmodule

```
module  test ( Q,S,clk );
output  Q;
input   S,clk;
reg     Q;
always@(S or clk)
        Q<=(S&clk) ;
endmodule
```

Overview of Test Bench

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



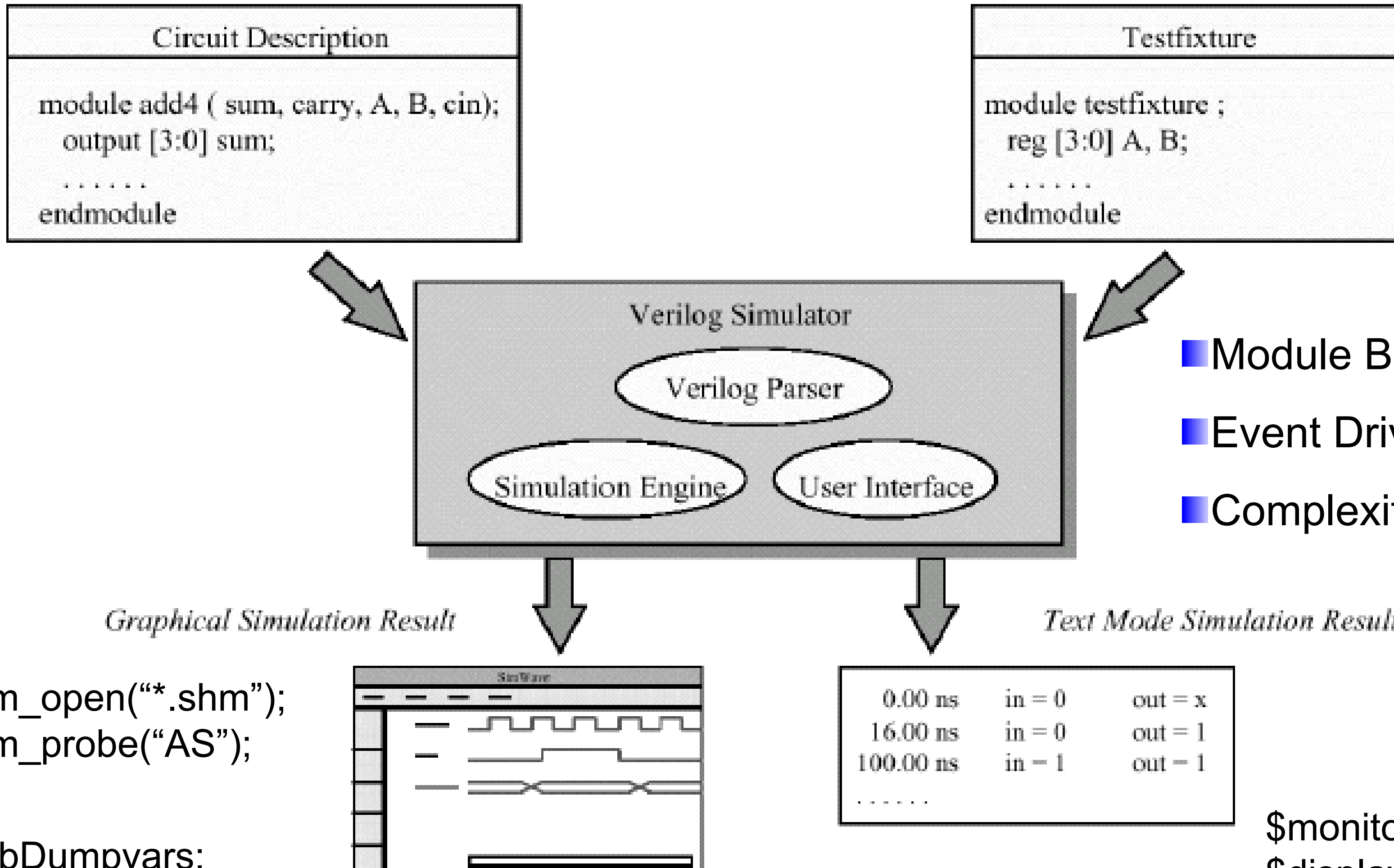
- A test bench is a top level module without inputs and outputs
- Data type declaration
  - Declare storage elements to store the test patterns
- Module instantiation
  - Instantiate pre-defined modules in current scope
  - Connect their I/O ports to other devices
- Applying stimulus
  - Describe stimulus by behavior modeling
- Display results
  - By text output, graphic output, or waveform display tools

A Complete Test Fixture

```
module testfixture;
//Data type declaration
reg a,b,sel;
//MUX instance
MUX2_1 mux(out,a,b,sel);
//Apply stimulus
initial begin
a = 0; b = 1; sel = 0;
#5 b = 0;
#5 b = 1; sel = 1;
#5 a = 1;
#5 $finish;
end
//Display results
initial $monitor($time,"out=%b a=%b b=%b sel=%b", out, a, b, sel);
endmodule
```

time	a	b	sel
0	0	1	0
5	0	0	0
10	0	1	1
15	1	1	1
20	1	1	1

Verilog Design & Verification



- Module Based
- Event Driven
- Complexity

$shm_open("*.shm");
$shm_probe("AS");
…
$fsdbDumpvars;
$fsdbDumpfile("*.fsdb");
…

$monitor(…);
$display(…);
…

Starting the Verilog-XL Simulation

- You can type *verilog* under UNIX to see various command line options and their corresponding actions.
  - unix % verilog
  - -f <filename> read host command arguments from file
  - -v <filename> specify library file

- For example
  - unix % verilog +lic_ncv file1.v file2.v testfile.v
  - unix % verilog –f run.bat
  - unix % verilog +lic_ncv testfile.v

run.bat

```
+lic_ncv
file1.v
file2.v
testfile.v
```

testfile.v

```
`include “file1.v”
`include “file2.v”
module testfile;
…
```

Outline

- Introduction
- Verilog-HDL Circuit Design
  - Behavior Level
  - Register-Transistor Level
  - Gate Level
  - Circuit Level
- Synthesis
- Coding Style

Verilog Module

module

- Module Name & Ports List
- Definitions: Ports, Wires, Registers, Parameter, Integer, Function
- Module Instantiations
- Module Statements & Constructs

endmodule

```
module testckt(a, b, c, d, z, sum);
  input          a, b;       //Inputs to nand gate
  input   [3:0]  c, d;       //Bused Input
  output         z;          //Outputs from nand gate
  output  [3:0]  sum;        //Bused output
  wire           and_out;    //Output from and gate

AND  instance1(a, b, and_out);
INV  instance2(and_out, z);

always @(c or d)
  begin
    sum = c+d;               //2's complement adder
  end

endmodule
```

Recall Verilog Structure

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

- Port Declaration
  - input port
  - output port
  - inout port
- Data Type Declaration
  - wire (wand, wor…)
  - reg (trireg…)
  - integer
  - time, real, realtime

wire & reg

- wire(wand, wor, tri)
  - Physical wires in a circuit
  - Cannot assign a value to a wire within a function or a begin…end block
  - A wire does not store its value, it must be driven by
    - —by connecting the wire to the output of a gate or module
    - —by assigning a value to the wire in a continuous assignment
  - An undriven wire defaults to a value of Z(high impedance)
  - input, output, inout port declaration—wire data type(default)
- reg
  - A variable in Verilog
  - Use of "reg" data type not exactly synthesized to really register

Sub-module Mapping

```
module adder (in1,in2,cin,sum,cout);
.......
endmodule
```

Position Mapping

```
module adder8(....) ;
adder  add1(a,b,1'b0,s1,c1) ,
       add2(.in1(a2),.in2(b2),.cin(c1),.sum(s2)
            ,.cout(c2)) ;
.....
endmodule
```

Name Mapping

Verilog Build-In Primitives

- Verilog primitive cells build basic combinational circuit
- Verilog primitives cells
  - and, nand, or, nor, xor, xnor, buf, not
  - bufif0, bufif1, notif0, notif1
  - pullup, pulldown
  - tran, tranif0, tranif1
  - nmos, pmos, cmos
  - rnmos, rpmos, rcmos rtran, rtranif0, rtranif1

Function Declarations

- Function declarations are one of the two primary methods for describing combinational logic
- Be declared and used within a module

- Function construct

```
function [range] name_of_function ;
        [func_declaration]
        statement_or_null
endfunction
```

```
function [8:0] adder;
input [7:0] a, b;
reg c;
reg [7:0] temp;
integer i;
begin
c = 0;
for (i = 0; i <= 7; i = i + 1) begin
temp[i] = a[i] ^ b[i] ^ c;
c = a[i] & b[i] | a[i] & c | b[i] & c;
end
end
adder = { c , temp};
endfunction

assign {cout, sum} = adder(a,b);
```

Operators Precedence

{ }	concatenation
!	logical negation
~	bitwise negation
* / %	arithmetic multiply / divide / modulus
+ -	arithmetic add / substrate (Two's complement)
<< >>	left shift / right shift
== !=	logical equal / inequal
=== !===	case equal / inequal
&	bitwise and
^ ^~ ~^	bitwise xor / xnor
|	bitwise or
&& ||	logical and / or
?:	condition

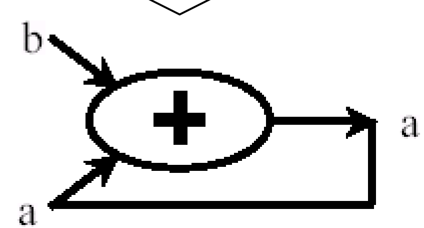
Continuous Assignment

- Drive a value onto a wire, wand, wor, or tri
- Used for datapath descriptions
- Used to model combinational circuits
- Avoid logic loop

```
wire a;            //declare
assign a=b&c;      //assign
```

```
wire a=b&c;        //declare and assign
```

```
assign a=b+a;
```



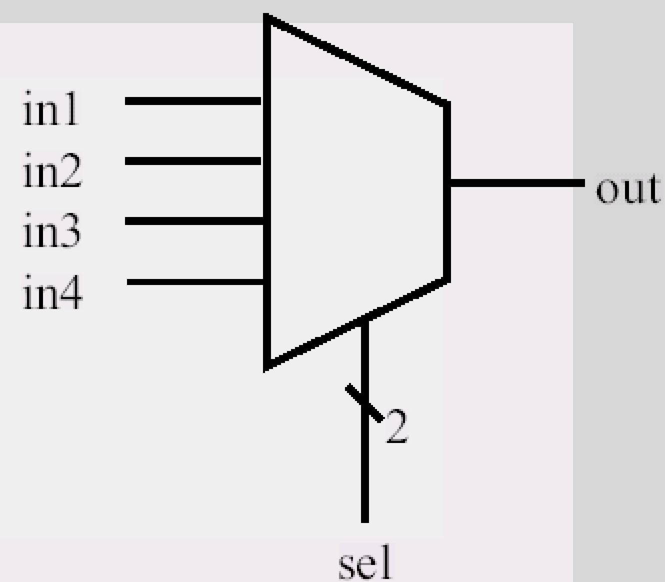
Bit-wise, Unary, Logical Operator

- a=4’b1011   b=4’b0010
- Bit-wire Operator
  - a|b   => 4’b1011
  - a&b   => 4’b0010
  - ~a   => 4’b0100
- Unary reduction Operator
  - |a   => 1’b1
  - &b   => 1’b0
- Logical Operator
  - a||b   => true
  - a&&b   => true
  - !a   => false

Conditional Operator

- The value assigned to LHS is the one that results TRUE from the expression evaluation
- This can be used to model muxiplelexer
- Can be nested

```
assign out = ( sel == 2'b00) ? in1 :
             (sel == 2'b01 ) ? in2 :
             (sel == 2'b10 ) ? in3 :
             (sel == 2'b11 ) ? in4 :
                1'bx;
```

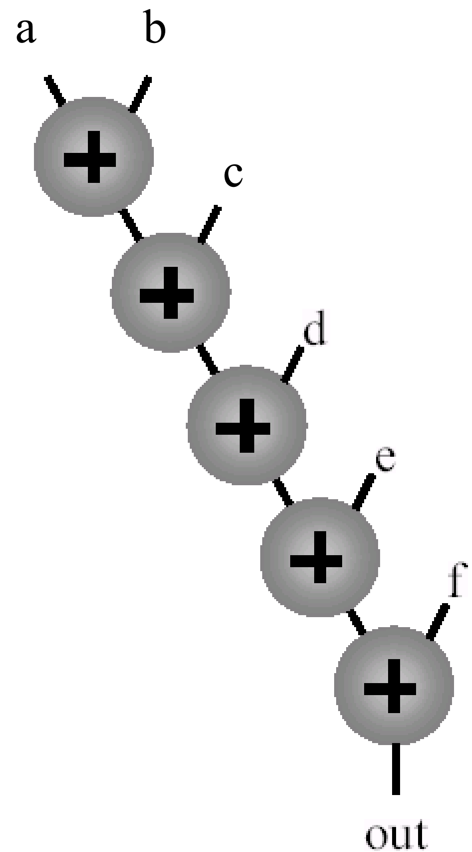


Concatenation Operator

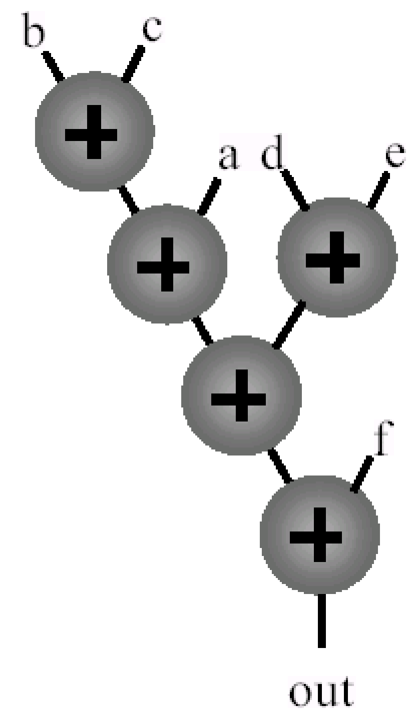
- Combine one or more expressions to form a larger vector
- If you want to transfer more than one data from function construct, concatenation operator is a good choice
  - 3’b100 => {1’b1,{2{1’b0}
  - {w,w,w,w} => {4{a}}
  - {x,y,z,y,z} => {x,{2{y,z}}}

Use Parentheses Properly

- out = a+b+c+d+e+f;

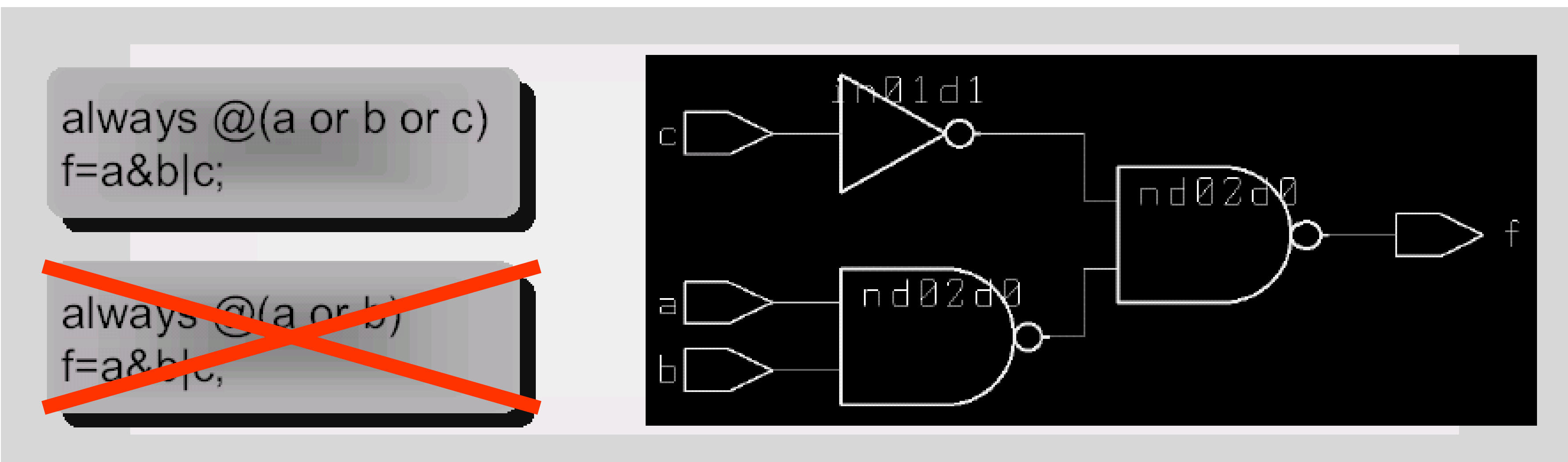


- out = ((a+(b+c))+d+e)+f;



Combinational Always Block

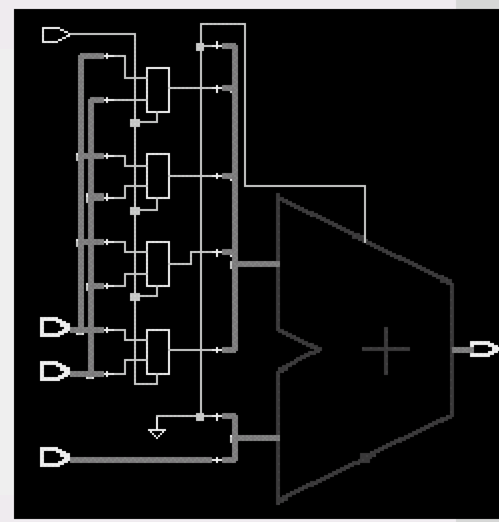
- Sensitivity list must be specified completely, otherwise synthesis may mismatch with simulation



Resource Sharing

- Operations can be shared if they lie the same always block

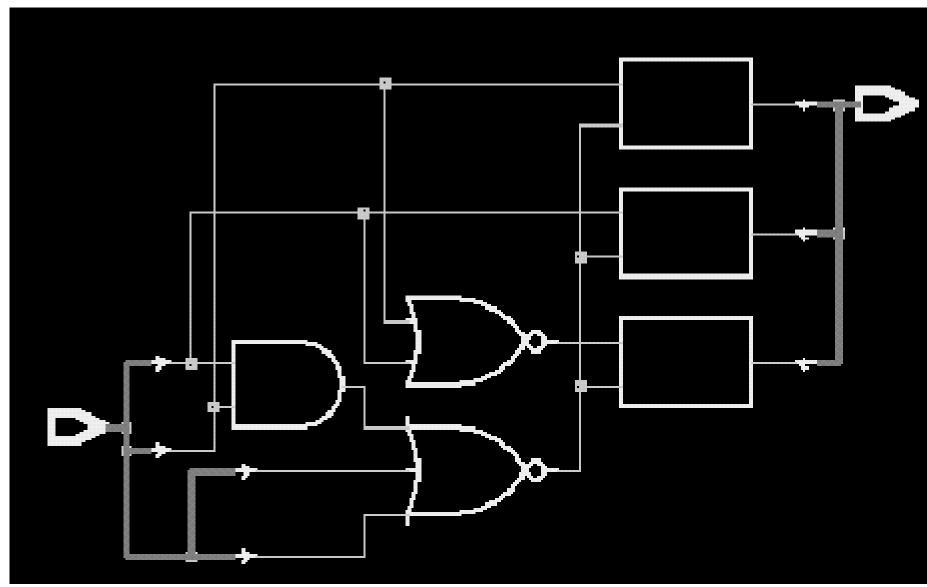
```
always @(a or b or c or sel)
if (sel)
        z = a + b ;
else
        z = a + c ;
```



Avoid Latch Inference

- If both “if” and “case” statement is not a full case, it will inferred latch

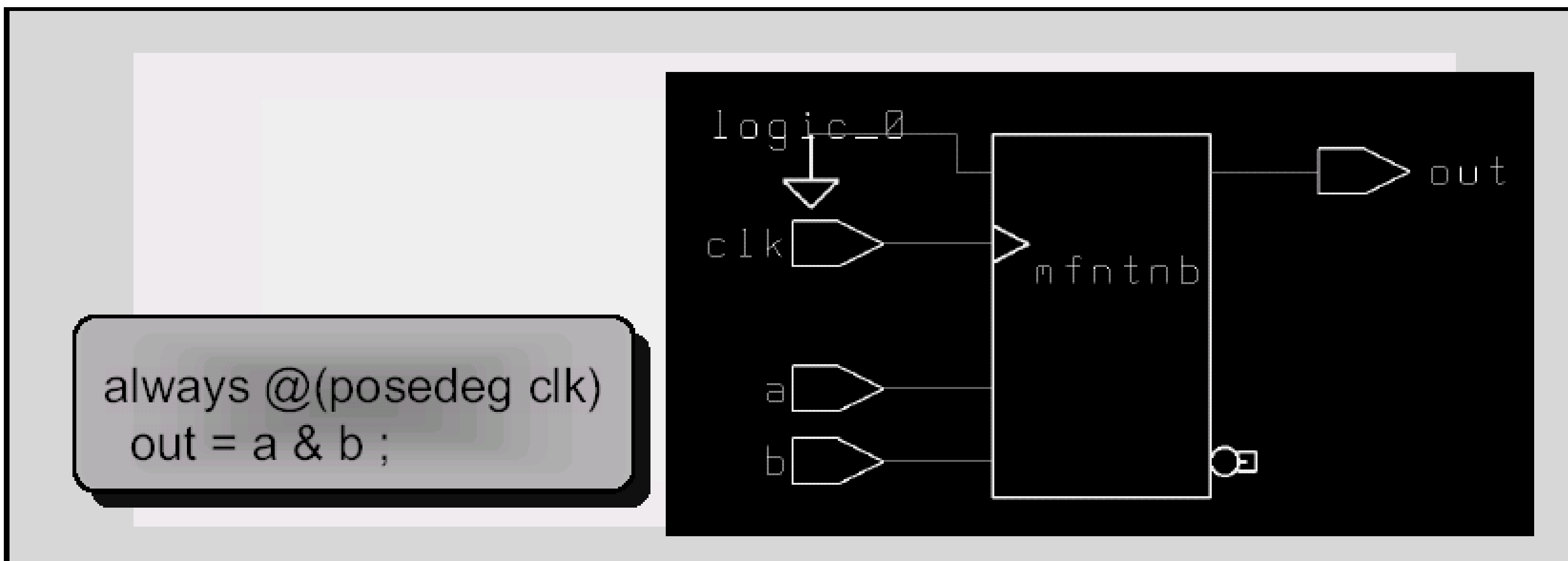
```
always @(bcd)
  begin
    if (bcd==4'd0)
        out <= 3'b001;
    if (bcd==4'd1)
        out <= 3'b010;
    if (bcd==4'd2)
        out <= 3'b100;
  end
```



```
always @(bcd)
  begin
    case(bcd)
        4'd0: out<=3'b001;
        4'd1: out<=3'b010;
        4'd2: out<=3'b100;
    endcase
  end
```

Register Inference

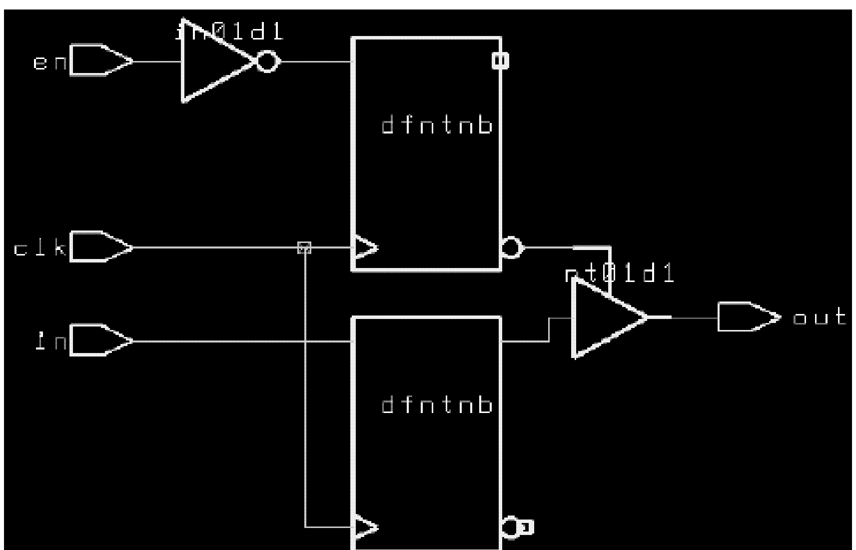
- A register (flip-flop) is implied when you use the @(posedge clk) or @(negedge clk) in an always block
- Any variable that is assigned a value in this always block is synthesized as a D-type edge-triggered flip-flop

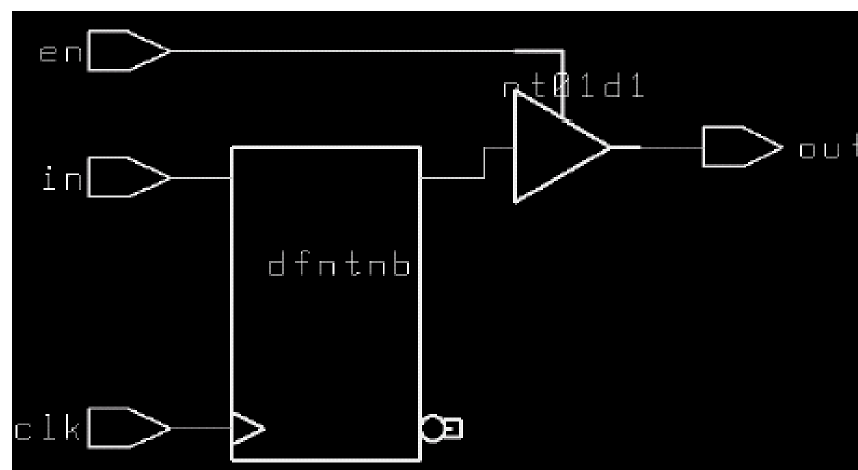


Separate Comb. & Seq. Assignment

```
always @(posedge clk)
begin
  if (en)
    out = in;
  else
    out = 1'bz;
end
```

```
always @(posedge clk)
    temp = in;

always @(posedge clk)
begin
  if (en)    out = temp;
  else    out = 1'bz;
end
```

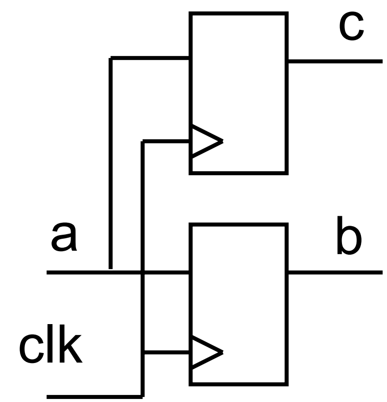




Blocking & non-Blocking

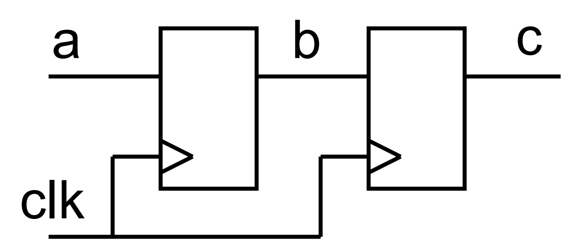
- Blocking assignments

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



- Non-blocking assignments

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



- Ex: initial(a=0,b=1,c=0), a: 0=>1

Parameters

- Use paremeters to declare run-time constants
- Syntax
  - parameter <list-of-assignments>
- You can use a parameter anywhere that you can use a literal

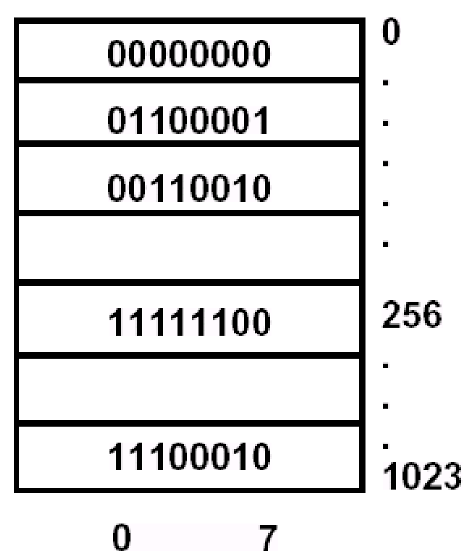
```
module mod1(out, inq, in2);
  …
  parameter p1=7,
                  real_constant = 1.432,
                  x_word = 16'bx,
                  file = "/usr/design/mem_file.dat";
  …
  wire  [p1:0]  w1;  // a wire declaration using paremeter
  …
endmodule
```

Memory

- Memory declaration
  - reg [wordsize-1:0] MEM [0:memsize-1];
    - —reg [15:0] MEM [0:1023]
- Read file format
  - $readmemb & $readmemh
    - —$readmemb("mem_file.txt",mem);

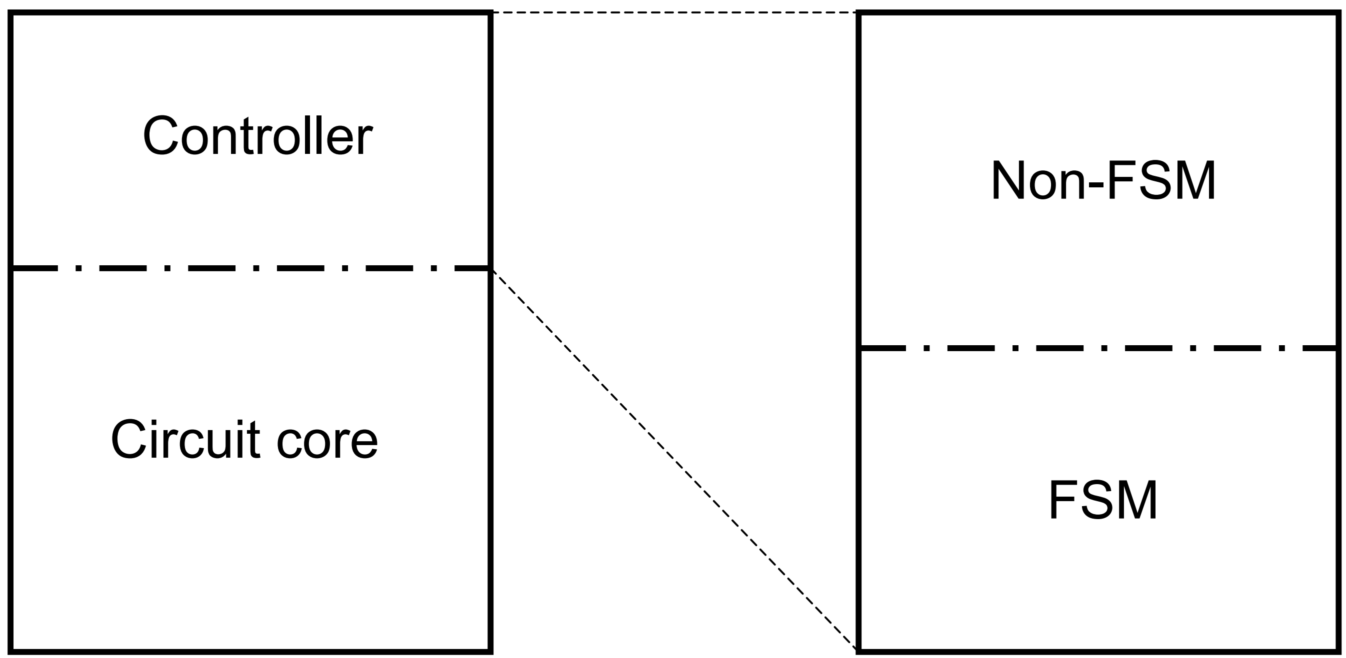
mem_file.txt **reg[0:7] mem [0:1023]**

```
0000_0000
0110_0001 0011_0010
//addresses 3-255 are not
//defined
@100
1111_1100
/*addresses 257-1022 are not
defined */
@3FF
1110_0010
```

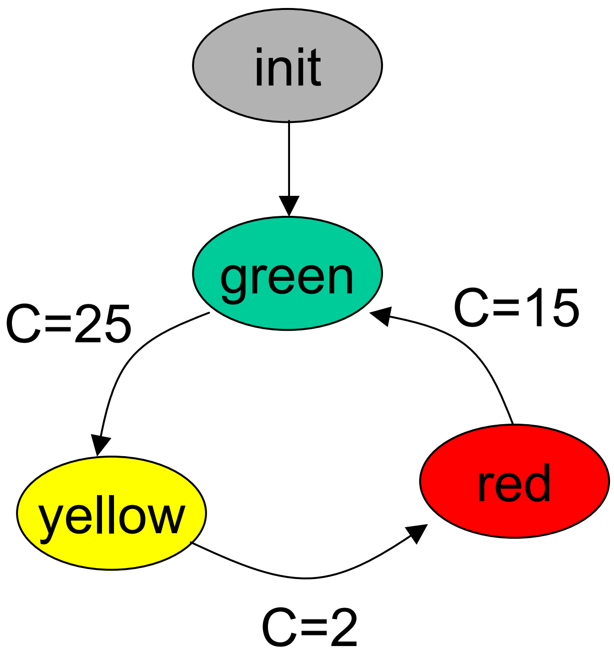


Finite State Machine

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



Example of FSM



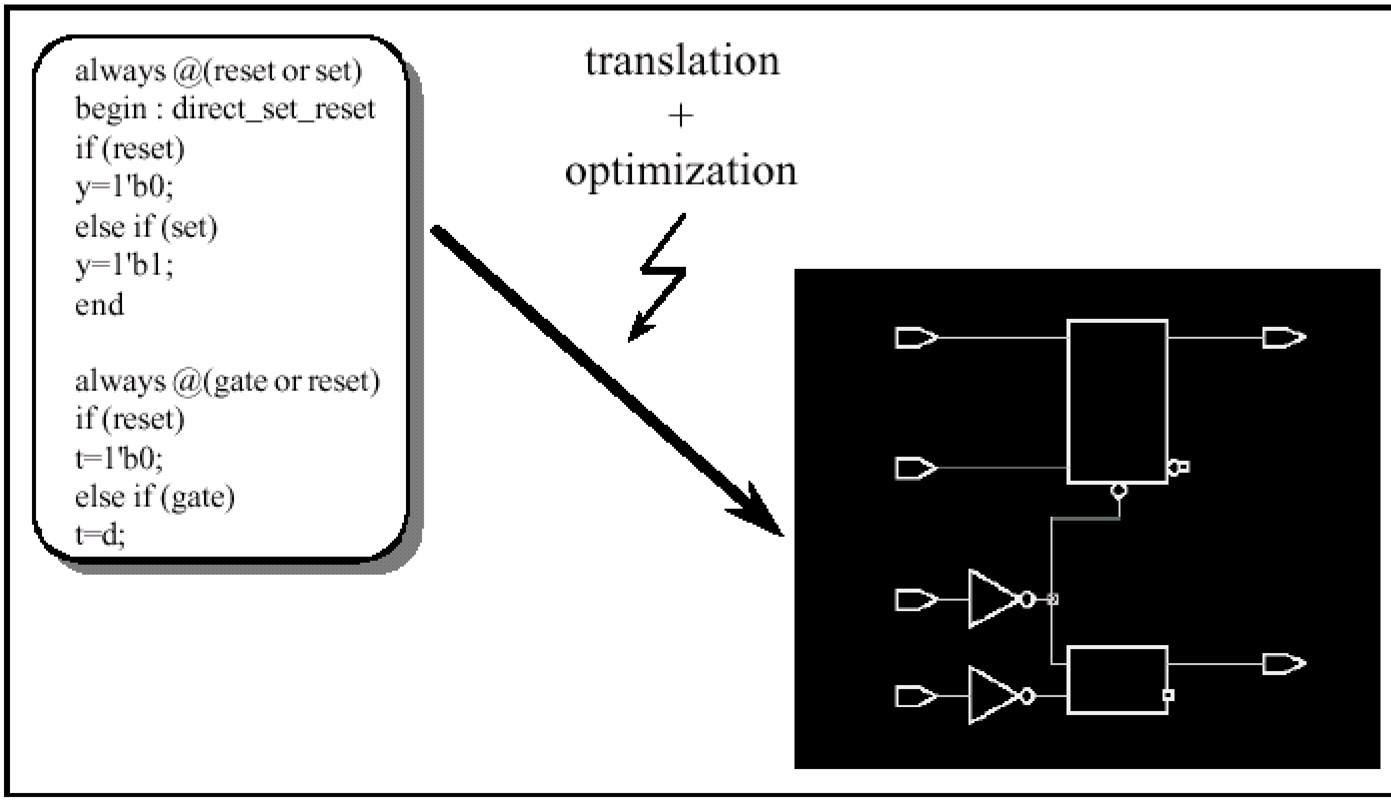

```
always @(fsm or count)
  begin
  parameter [1:0] init = 0, g = 1, y = 2, r = 3;
  red = 0; greed = 0; yellow = 0;
  fsm_nxt = fsm; start = 0;

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

Outline

- Introduction
- Verilog-HDL Circuit Design
  - Behavior Level
  - Register-Transistor Level
  - Gate Level
  - Circuit Level
- Synthesis
- Coding Style

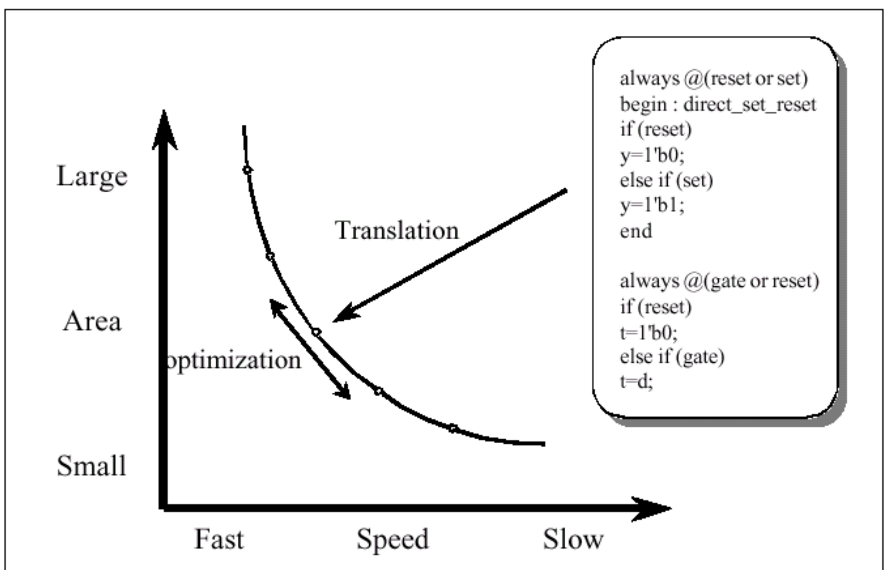
What is Synthesis

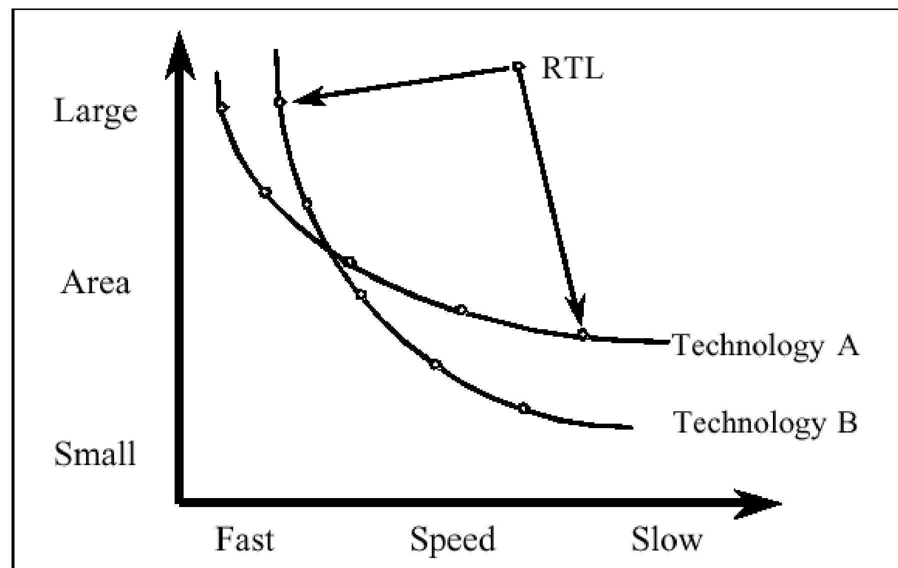
- Synthesis = translation + optimization



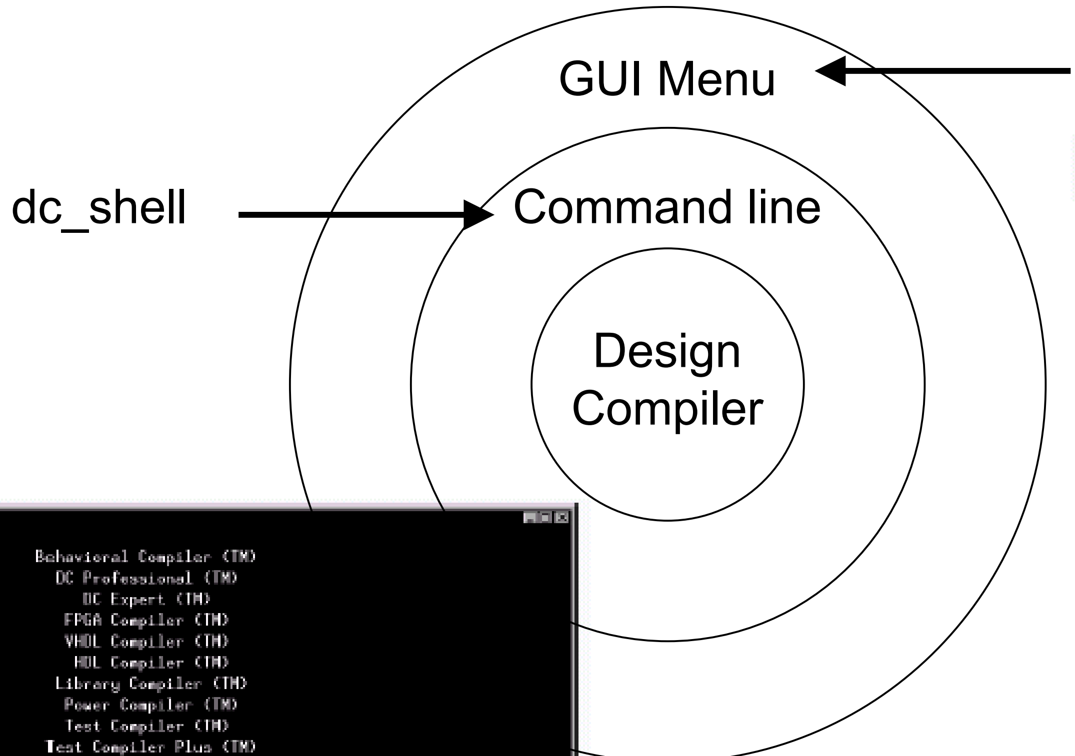
Translation & Optimization

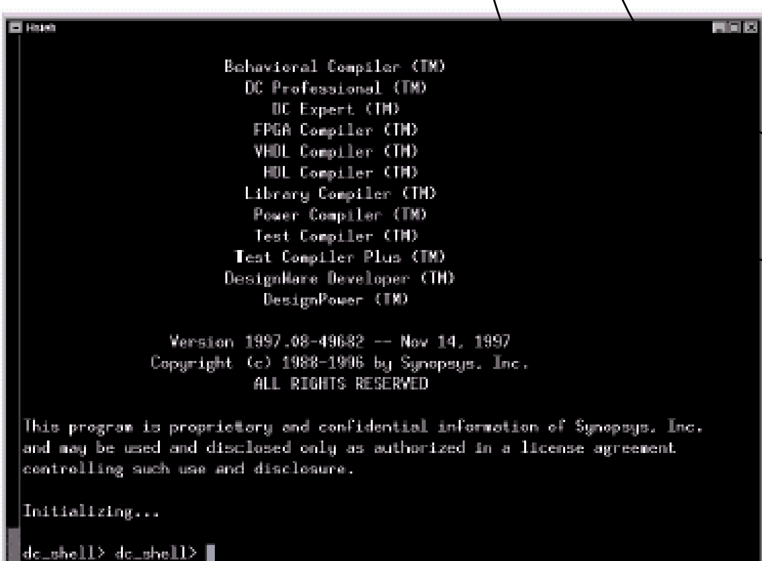
- Synthesis is Constraint Driven
- Technology Independent



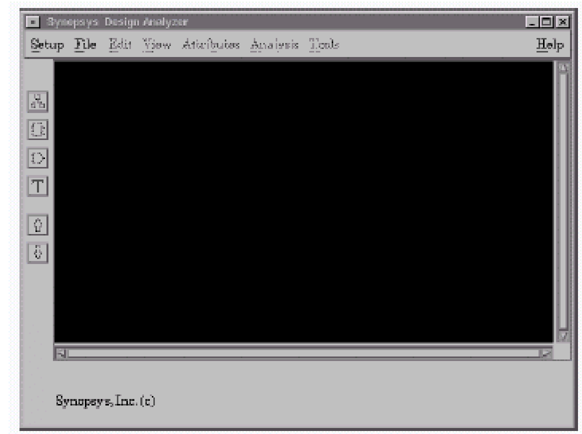


Design Compiler Interaction

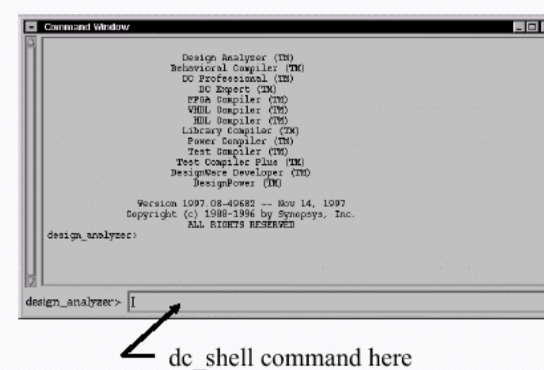




- % design_analyzer &

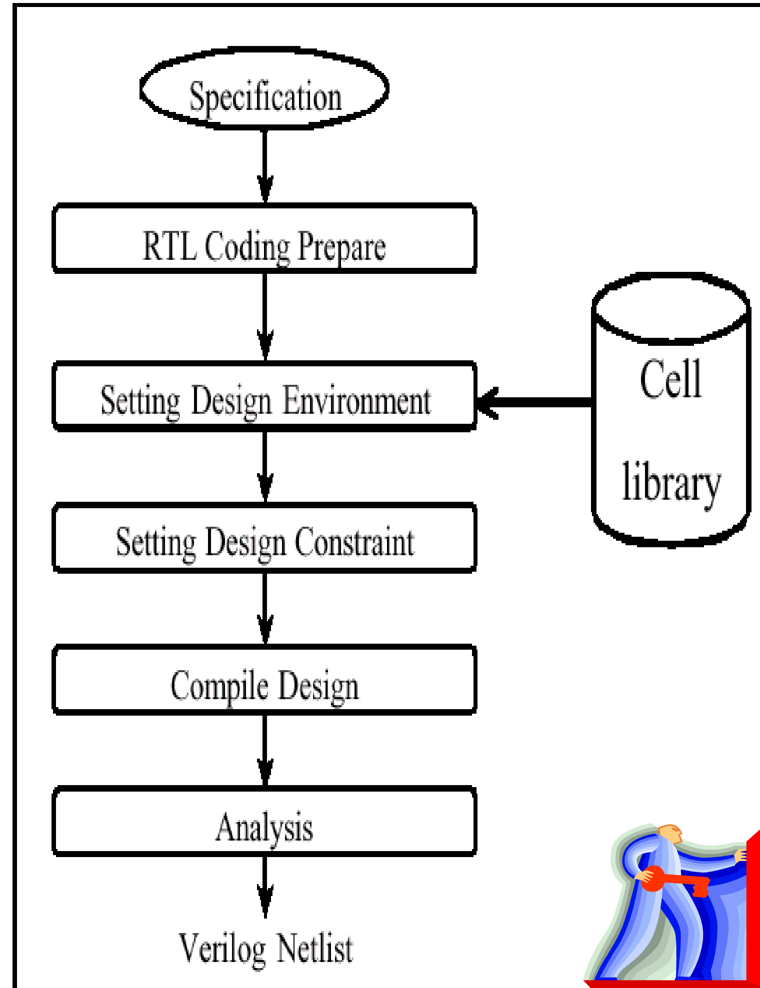


- Setup → Command Window



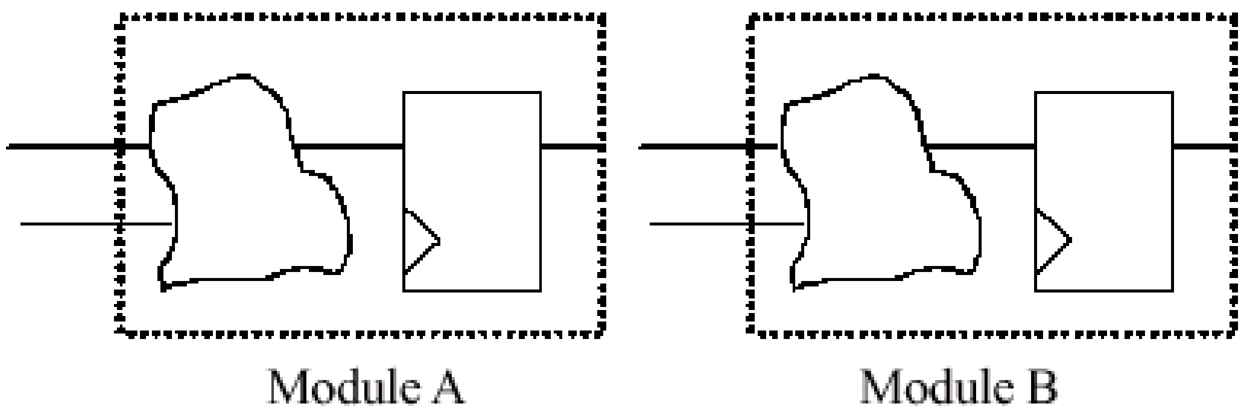
ASIC Synthesis Design Flow

- Develop the HDL design description and simulate the design description to verify that it is correct.
- Set up the .synopsys_dc.setup file.
  - Set the appropriate technology, synthetic, and symbol libraries, target libraries, and link libraries.
  - Set the necessary compilation options, including options to read in the input files and specify the output formats.
- Read the HDL design description.
- Define the design.
  - Set design attributes
  - Define environmental conditions
  - Set design rules
  - Set realistic constraints (timing and area goals)
  - Determine a compile methodology



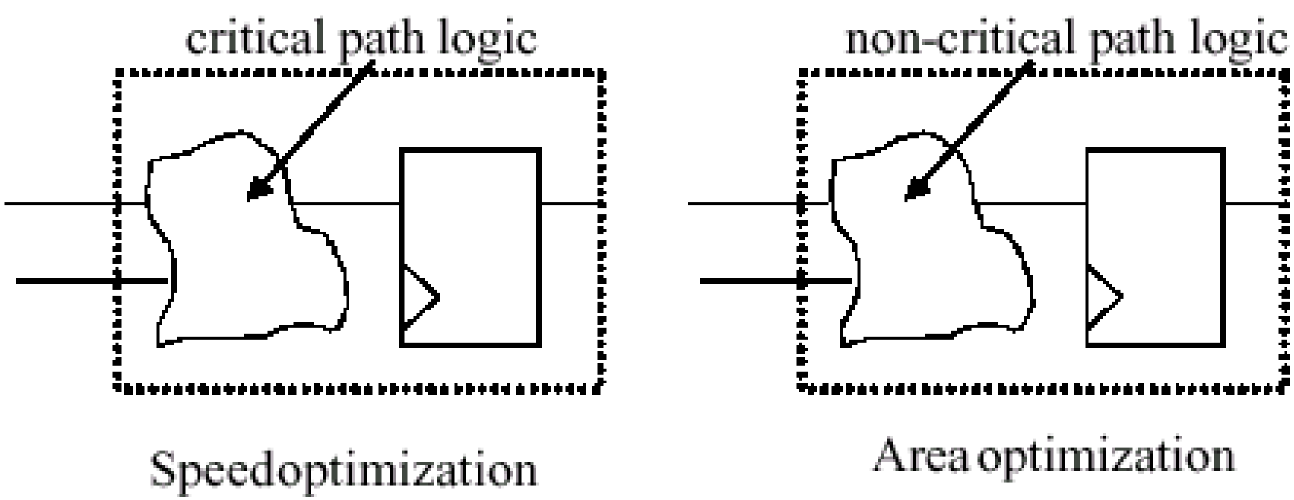
Register at Hierarchical Output

- Keep related combination logic in a single module.
- Register all at output make input data arrival time and output drive strength predictable.



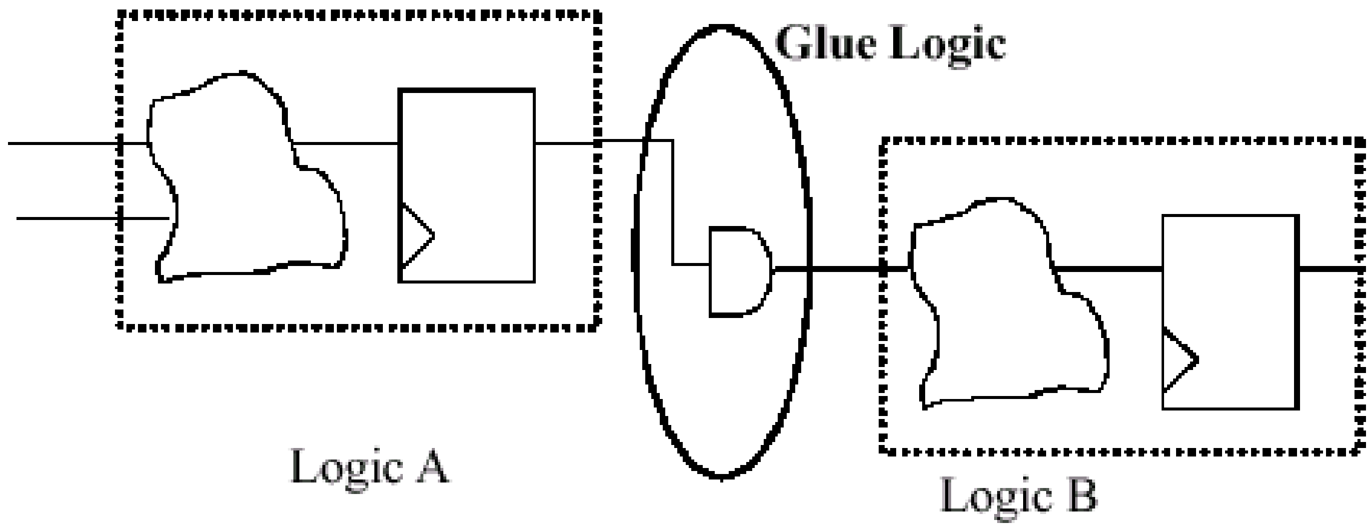
Partition by Design Goals

- Optimize the critical path logic for speed
- Optimize non-critical path logic for area.



Avoid Glue Logic

- No Cells except at leaf levels of hierarchy
- Any extra gates should be grouped into a sub-design



Outline

- Introduction
- Verilog-HDL Circuit Design
  - Behavior Level
  - Register-Transistor Level
  - Gate Level
  - Circuit Level
- Synthesis
- Coding Style

Principles of RTL Coding Styles

- Readability
- Simplicity
- Locality
- Portability
- Reusability
- Reconfigurability

File Header

- Should be included for all source files
- Contents
  - author information
  - revision history
  - purpose description
  - available parameters
  - reset scheme and clock domain
  - critical timing and asynchronous interface
  - test structure
- A corporation-wide standard template

Naming Conventions

- Lowercase letters for signal names
- Uppercase letters for constants
- Case-insensitive naming
- Use *clk* for clocks, *rst* for resets
- Suffixes
  - _*n* for active-low
  - _*a* for async
  - _*z* for tri-state
- Identical names for connected signals and ports
- Do not use HDL reserved words
- Consistency within group, division and corporation

Ports

- Ordering
  - One port per line with appropriate comments
  - Inputs first then outputs
  - Clocks, resets, enables, other controls, address bus, then data bus…
- Mapping
  - Used **named** mapping instead of **positional** mapping

Coding Practices

- Little-ending for multi-bit bus
- Operand sizes should match
- Expression in condition must be a 1-bit value
- Use parentheses in complex statements
- Do not assign signals don’t case value
- Reset all storage elements

Portability

- Do not use hard-coded numbers
- Avoid embedded synthesis scripts
- Use technology-independent libraries
- Avoid instantiating gates

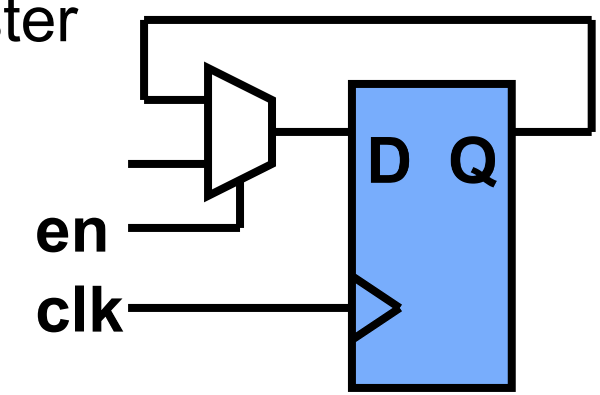
Clocks and Resets

- Simple clocking is easier to understand, analyze, and maintain
- Avoid using both edges of the clock
  - Duty-cycle sensitive
  - Difficult DFT process
- Do not buffer clock and reset networks
- Avoid gated clock
- Avoid internally generated clock and resets
  - Limited testability

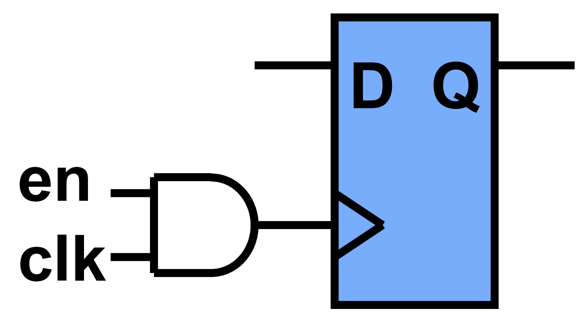
Low Power

- Clock gating
  - 50%~70% power consumed in clock network reported
  - Gating the clock to an entire block
  - Gating the clock to a register

```
always @(posedge clk)
    if (en)
        q <= q_nxt;
```




```
Assign clk1 = clk & en;
always @(posedge clk1)
        q <= q_nxt;
```



Synchronicity

- Infer technology-independent registers
  - (positive) single edge-triggered registers
- Avoid latches intentionally
  - Except for small memory and FIFO
- Avoid latches unintentionally
  - Avoid incomplete assignment in case statement
  - Use default assignments
  - Avoid incomplete if-then-else chain
- Avoid combinational feedback loops
  - STA and ATPG problem

Combinational and Sequential

- Combinational block
  - Use blocking assignments (=)
  - Minimize signals required in sensitivity list
  - Assignment should be applied in topological order
- Sequential block
  - Use non-blocking assignments (<=)
  - Avoid **race** problems in simulation
- Com./Seq. logic should be separated

Coding for Synthesis

- Specify complete but no redundant sensitivity lists
  - Simulation coherence
  - Simulation speed
- *If-then-else* often infers a cascaded encoder
  - Inputs signals with different arrival time
- *Case* infers a single-level mux
  - *Case* is better if priority encoding is not required
  - *Case* is generally simulated faster then *if-then-else*
- Conditional assignments (?:)
  - Infer a mux, with slower simulation performance

Coding for Synthesis

- FSM
  - Partition FSM and non-FSM logic
  - Partition combinational part and sequential part
  - Use parameter to define names of the state vector
  - Assign a default (reset) state
- No # delay statements
- Use *full_case* and *parallel_case* judiciously
- Explicitly declare wires
- Avoid glue logic at the top-level
- Avoid expressions in port connections

Partitioning

- Register all outputs
  - Make output drive strengths and input delay predictable
  - Ease time budgeting and constraints
- Keep related logic together
  - Improve synthesis quality
- Partition logic with different design goals
- Avoid asynchronous logic
  - Technology dependent
  - More difficult to ensure correct functionality and timing
  - As small as possible and isolation
- Keep sharable resources in the same block

Q&A