Introduction to Verilog design

Lecture 2

Design flow (from the book)

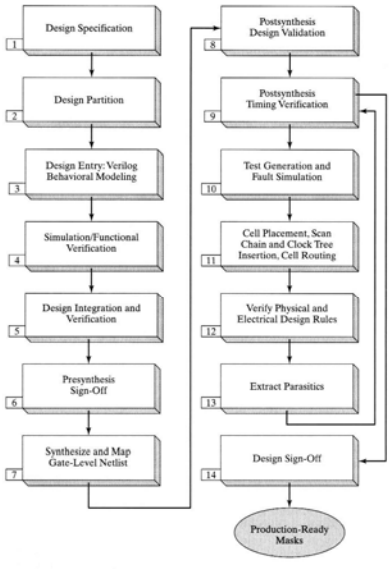
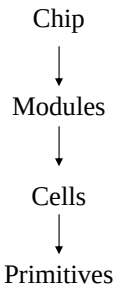


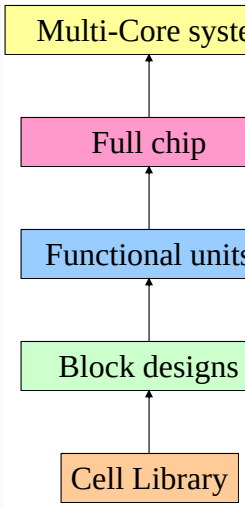
FIGURE 1-1 Design flow for HDL-based ASICs

Hierarchical Design



- A chip contain many modules
- A module may contain other modules
  - no recursion
- A module uses predefined cells
- Everything is based on primitives

Hierarchical design flow



SW/HW co-design, multiple cores
(SW/HW system verification)

Processor, DSP, MPEG, etc.
(Functionality is specified in books)

ALU, Float Point Unit, Filter, Memory Management, Scheduler, etc.
(RTL verification can be hard)

Arithmetic blocks, common bus structures, Cross-bar switches, decoders, etc.
(May be custom designed)

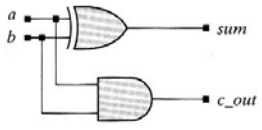
Certify that layouts can be manufactured with high yield
(This may be Fab dependent)

Advantages

- When working at a particular level of design, we don't need to worry about all detail below
  - When putting a system together, you don't need to worry about how a processor is designed
  - Optimization can be done independently at each level
- Since no detail is involved, it is easier to replace the designs with other implementations
  - You can easily change the processor to another implementation without changing the system
  - You can change the technology from 0.45 micron to 0.32 micron without changing the verilog RTL

Primtives and design model

- Verilog includes 26 predefined models of logic gates called primitives
- Primitives are the most basic functional components that can be used to build (*model*) a design
  - Their functions are built into the language by internal truth table
- The output port of a primitive is the first in the list




```
module Add_half (sum, c_out, a, b);
   input      a, b;
   output     c_out, sum;
   xor        (sum, a, b);
   and        (c_out, a, b);
endmodule
```

FIGURE 4-1 Schematic and Verilog description of a half adder.

Verilog primitives for combinational logic

n-input	n-output, 3-state
and	buf
nand	not
or	bufif0
nor	bufif1
xor	notif0
xnor	notif1

- n-input: Any number of inputs and 1 output
- n-output: Any number of outputs and 1 input

List of Verilog primitives

- Gates
  - and, nand, or, nor, xor, xnor, buf, not
- Tri-State
  - bufif0, bufif1, notif0, notif1
- MOS
  - nmos, pmos, rnmos, rpmos
- CMOS
  - cmos, rcmos
- Bi-directional
  - tran, tranif0, tranif1, rtran, rtranif0, rtranif1
- Pull
  - pullup, pulldown

Standard cells vs. Primitives

- Verilog primitives are abstract building blocks used to model your design
  - They are not real
- Standard cells are real building blocks used to implement your design
  - Each has a corresponding layout
  - Each has a timing model
  - Cells are stored in a cell library
  - Cell library is NOT free; you need to buy

Structural Verilog model

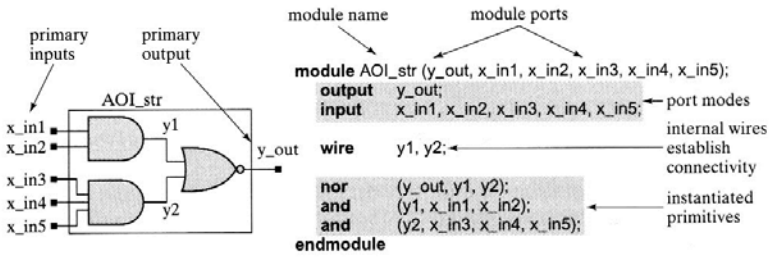
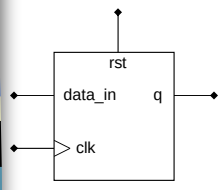


FIGURE 4-5 An AOI circuit and its Verilog structural model.

Verilog Behavior of D FF



Block Diagram

```
module Flip_flop (q, data_in,
   clk, rst);
  input data_in, clk, rst;
  output q;
  reg q;

always @ (posedge clk)
  begin
    if (rst == 1) q = 0;
    else q = data_in;
  end
endmodule
```

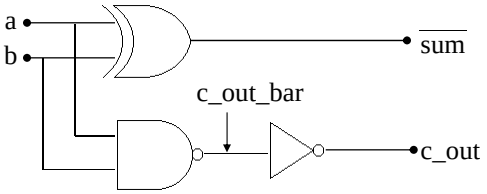
Declaration of Synchronous Behavior

Procedural Statements

Block diagram view

Block Diagram

Block Diagram



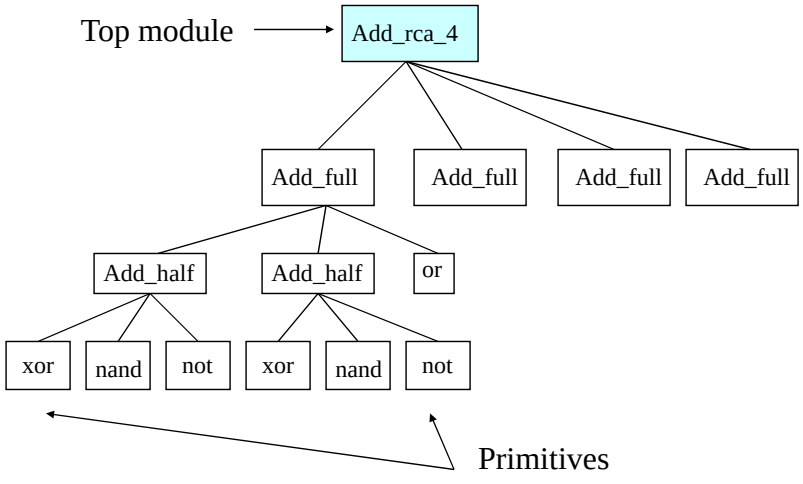
Schematic

One Implementation

Comparison

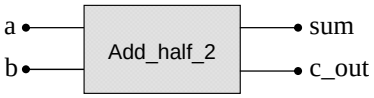
- Block vs. Implementation
  - Block view
    - Specify inputs and outputs
    - No internal detail
  - Implementation
    - Include specific detail
    - One block view can have many implementations
- Structural vs. Behavior
  - Structural model
    - Specify how a function is achieved
  - Behavior model
    - Specify only the function
    - One behavior can have many structural models

Hierarchical design example



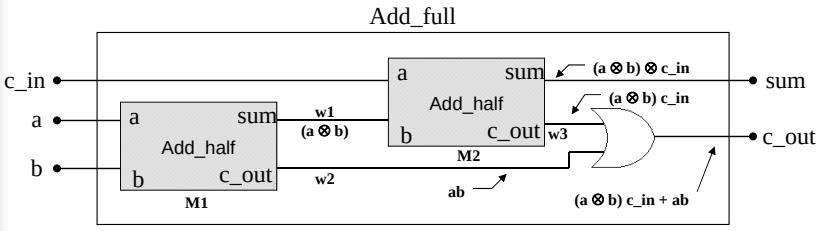
An half adder (behavior/structural)

```
module Add_half_2 (sum, c_out, a, b);
  input   a, b;
  output c_out, sum;

  assign {c_out, sum} = a + b;
endmodule
```




```
module Add_half_1 (sum, c_out, a, b);
  input   a, b;
  output sum, c_out;
  wire     c_out_bar;

  xor (sum, a, b);
  nand (c_out_bar, a, b);
  not (c_out, c_out_bar);
endmodule
```

A full adder (structural)




```
module Add_full (sum, c_out, a, b, c_in);      // parent module
  input  a, b, c_in;
  output  c_out, sum;
  wire  w1, w2, w3;

  Add_half  M1  (w1, w2, a, b);                // child module
  Add_half  M2  (sum, w3, w1, c_in);           // child module
  or (c_out, w2, w3);                          // primitive instantiation
endmodule
```

module instance name

4-bit adder

Module Add_full(sum, c_out, a, b, c_in)

```
module array_of_adders  (sum, c_out, a, b, c_in);
    input      [3:0]      a,b;
    input                 c_in;
    output     [3:0]      sum;
    output                c_out;
    wire       [3:1]      carry;

Add_full M [3:0] (sum, {c_out, carry[3:1]}, a, b, {carry[3:1], c_in});

endmodule
```

Same as saying …

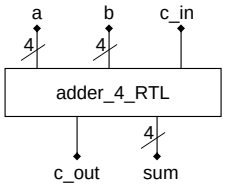
Add_full M[0](sum[0], carry[1], a[0], b[0], c_in)
Add_full M[1](sum[1], carry[2], a[1], b[1], carry[1])
Add_full M[2](sum[2], carry[3], a[2], b[2], carry[2])
Add_full M[3](sum[3], c_out, a[3], b[3], carry[3])

Behavior model of a 4-bit adder

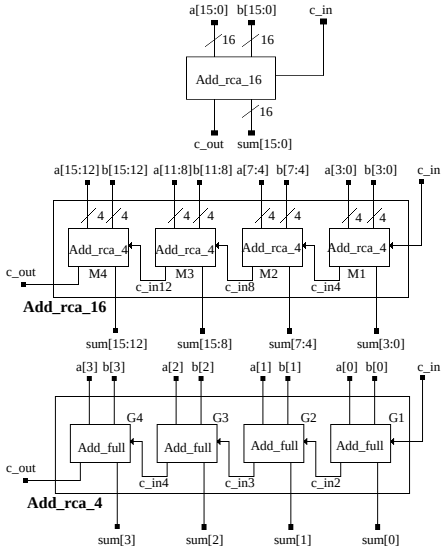
```
module adder_4_RTL (a, b, c_in, sum, c_out);
  output   [3:0]    sum;
  output            c_out;
  input    [3:0]    a, b;
  input             c_in;

  assign {c_out, sum} = a + b + c_in;

endmodule
```

Continuous Assignment

Concatenation



16-bit adder



16-bit Adder

```
module Add_rca_16 (sum, c_out, a, b, c_in);
  output    [15:0]    sum;
  output              c_out;
  input     [15:0]    a, b;
  input               c_in;
  wire                c_in, c_in4, c_in8, c_in12, c_out;

  Add_rca_4 M1 (sum[3:0],     c_in4,    a[3:0],    b[3:0],    c_in);
  Add_rca_4 M2 (sum[7:4],     c_in8,    a[7:4],    b[7:4],    c_in4);
  Add_rca_4 M3 (sum[11:8],    c_in12,   a[11:8],   b[11:8],   c_in8);
  Add_rca_4 M4 (sum[15:12],   c_out,    a[15:12],  b[15:12],  c_in12);
endmodule

module Add_rca_4 (sum, c_out, a, b, c_in);
  output    [3:0]     sum;
  output              c_out;
  input     [3:0]     a, b;
  input               c_in;
  wire                c_in4, c_in3, c_in2;

  Add_full G1 (sum[0],    c_in2,    a[0],    b[0],    c_in);
  Add_full G2 (sum[1],    c_in3,    a[1],    b[1],    c_in2);
  Add_full G3 (sum[2],    c_in4,    a[2],    b[2],    c_in3);
  Add_full G4 (sum[3],    c_out,    a[3],    b[3],    c_in4);
endmodule
```

The concept of *Design Entry*

- Design Entry
  - The entry point to do your design
  - Select design data representation(s)
    - Behavior, RTL, Gate-Transistor Schematics
- Schematic
  - Draw your design with gates/transistors/lines
  - For high-performance blocks
- Hardware Description Language
  - Describe your design with Verilog/VHDL
  - Just like writing a program
  - No need to worry about structural detail

Unit level verification

- Each block or unit should be verified through extensive testbench simulation

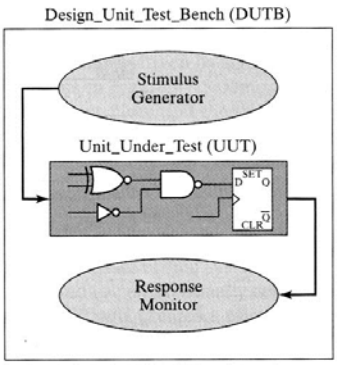


FIGURE 4-16 Organization of a testbench for verifying a unit under test.

A testbench example

```
module t_Add_half();
  wire        sum, c_out;
  reg         a, b;
  Add_half_0_delay M1 (sum, c_out, a, b);      //UUT
  initial begin                                // Time Out
  #100 $finish;
  end
  initial begin                                // Stimulus patterns
  #10 a = 0; b = 0;
  #10 b = 1;
  #10 a = 1;
  #10 b = 0;
  end
endmodule
```

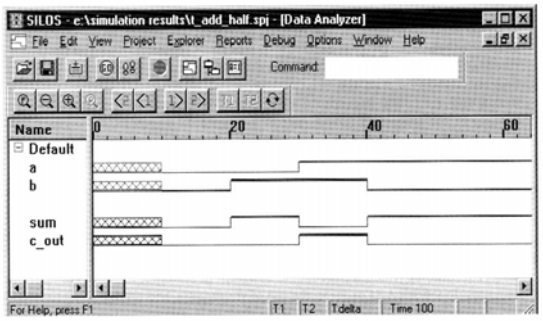


FIGURE 4-17 Waveforms produced by a simulation of *Add_half_0_Delay*, a 0-delay binary half adder.

Generic testbench design

```
module t_DUTB_name ();        // substitute the name of the UUT
  reg  ...;                   // Declaration of register variables for primary
                              // inputs of the UUT
  wire  ...;                  // Declaration of primary outputs of the UUT
  parameter                     time_out = // Provide a value
  UUT_name M1_instance_name (UUT ports go here);
  initial $monitor ( );       // Specification of signals to be monitored and
                              // displayed as text
  initial #time_out $stop;    // Stopwatch to assure termination of simulation
  initial                     // Develop one or more behaviors for pattern
                              // generation and/or
                              // error detection
  begin
                              // Behavioral statements generating waveforms
                              // to the input ports, and comments documenting
                              // the test. Use the full repertoire of behavioral
                              // constructs for loops and conditionals.
  end
endmodule
```

- New version of Verilog has a lot more features to support testbench development

Delay model in verilog

```
module Add_half_1 (sum, c_out, a, b);
  input   a, b;
  output sum, c_out;
  wire    c_out_bar;

  xor #2 (sum, a, b);
  nand #2 (c_out_bar, a, b);
  not #1 (c_out, c_out_bar);
endmodule
```

- These delays *have no physical meaning*
  - They are used only for simulation
  - The ordering of events is what being specified
    - Not the actual timing
  - Relative delays are more important than absolute numbers

Cell Delay Characterization

```
module and123 (o, a, b);
  input   a, b;
  output o;
  and (o, a, b);

  specify
    specparam
          Tpd_0_1 = 1.13, 3.09, 7.75
          Tpd_1_0 = 0.93, 2.50, 7.34
    (a => o) = (Tpd_0_1, Tpd_1_0);
    (b => o) = (Tpd_0_1, Tpd_1_0);
  endspecify
endmodule
```

- The delays are only estimations
  - Maybe from SPICE simulation of the actual layout
- This can only be done for very small cells

Inertial delay concept

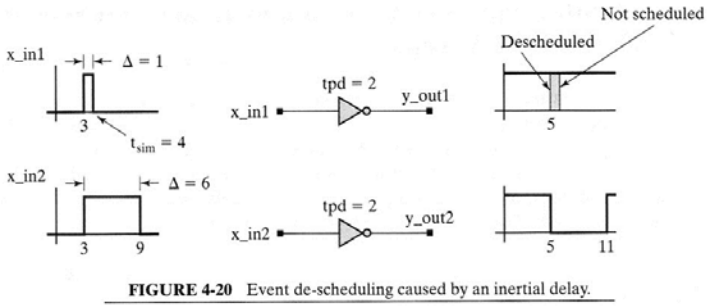


FIGURE 4-20 Event de-scheduling caused by an inertial delay.

- In the upper example, the event of rising transition is de-scheduled by the simulator because the pulse width is less than the propagation delay

User-defined primitives

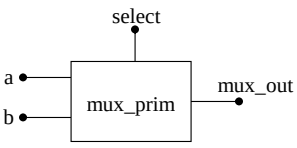
- Basic constructs in your design
- Contain detailed table-like description
- Use the keywords
  - **primitive**
  - **endprimitive**
- Differentiate themselves from Modules
  - In test, verification, and simulation, never goes inside primitives (black-boxes)

User-defined primitives

- Instantiated in the same manner as **and, or, not, nor, nand,** etc.
- *Always has one output port!*
- Output port must be declared as **net** in a combinational primitive
- Output port must be declared as **reg** in a sequential primitive
- No **inout**
- Specify everything (anything left unspecified will be treated as X)

2-bit multiplexer

- Z input is treated as X!




```
primitive mux_prim (mux_out, select, a, b);
   output  mux_out;
   input   select, a, b;
   table
//  select      a       b       :  mux_out
      0        0       0       :     0;     // Table order = port order
      0        0       1       :     0;     // One output, multiple inputs, no inout
      0        0       x       :     0;     // Only 0, 1, x on input and output
      0        1       0       :     1;     // A z input in simulation is treated as x
      0        1       1       :     1;     // Last column is the output
      0        1       x       :     1;

//  select      a       b       :  mux_out
      1        0       0       :     0;
      1        1       0       :     0;
      1        x       0       :     0;
      1        0       1       :     1;
      1        1       1       :     1;
      1        x       1       :     1;
      x        0       0       :     0;     // Reduces pessimism
      x        1       1       :     1;
   endtable                    // Note: Combinations not explicitly specified will drive "x"
endprimitive                   // under simulation
```

Shorthand notation “?”

```
table
// Shorthand notation;
// ? represents iteration of the table entry over the values 0, 1, x.
// i.e., don't care on the input

//   select     a     b     :   mux_out

      0         0     ?     :     0;    // ? = 0, 1, x shorthand notation
      0         1     ?     :     1;

      1         ?     0     :     0;
      1         ?     1     :     1;

      ?         0     0     :     0;
      ?         1     1     :     1;
endtable
```

Primitive notations

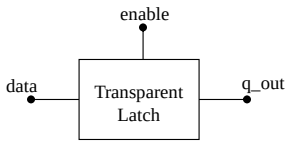
- 0, 1, x:
- ? : {0, 1, x}
- b : {0, 1}
- - : no change
- (vw) : v∈{0, 1, x} ---> w∈{0, 1, x}
- * : {0, 1, x} ---> {0, 1, x}
- r : (01)
- f : (10)
- p : (01), (0x), (x1), (0z), (z1)
- n : (10), (1x), (x0), (1z), (z0)

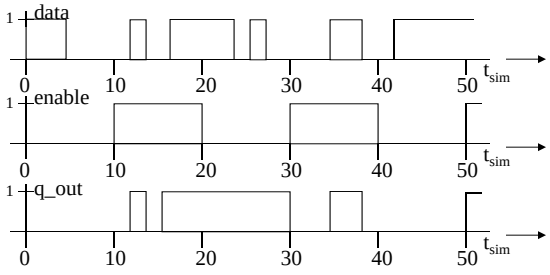
Sequential Primitives

- n inputs
- 1 state and 1 output
- n+2 columns
  - <n inputs>, <state>, <output>=<next state>
- Two types:
  - Level sensitive
    - Achieve transparent latch (a latch can be made "invisible")
  - Edge sensitive

A Transparent Latch

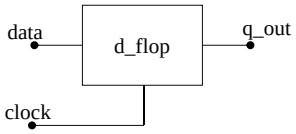
```
primitive latch (q_out, enable, data);
   output  q_out;
   input   enable, data;
   reg     q_out;

   table
//    en     data        state:   q_out/next state
      1       1      :     ?   :        1;
      1       0      :     ?   :        0;
      0       ?      :     ?   :        -;
// Note: '-' denotes "no change."
// The state is the residual value of q_out
   endtable                    // Note: Combinations not explicitly specified will drive "x"
endprimitive                   // under simulation
```





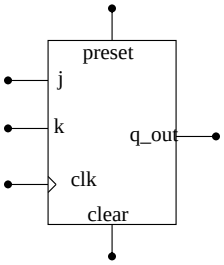
Rising Edge-Sensitive Latch

```
primitive d_prim1 (q_out, clock, data);
   output          q_out;
    input clock, data;

   reg    q_out;

   table
   //      clock      data      state   :   q_out/next state
           (01)       0     :    ?    :        0;      // Rising clock edge
           (01)       1     :    ?    :        1;
           (0x)       ?     :    ?    :        X;
           (?0)       ?     :    ?    :        -;      // Falling clock edge

            ?         (??)  :    ?    :        -;      // Steady clock,
    ignore data
   endtable
endprimitive
```



Level-S Dominates Edge-S

```
primitive jk_prim (q_out, clock, j, k, preset, clear);
   output   q_out;
   input    clock, j, k, preset, clear;
   reg      q_out;

   table
//      clk    j    k    pre   cir     state     q_out/next_state
   // Preset Logic
           ?    ?    ?    0     1   :   ?   :     1;
           ?    ?    ?    *     1   :   1   :     1;
   // Clear Logic
           ?    ?    ?    1     0   :   ?   :     0;
           ?    ?    ?    1     *   :   0   :     0;
// Normal Clocking
//      clk    j    k    pre   cir     state     q_out/next_state
           r    0    0    0     0   :   0   :     1;
           r    0    0    1     1   :   ?   :     -;
           r    0    1    1     1   :   ?   :     0;
           r    1    0    1     1   :   ?   :     1;
           r    1    1    1     1   :   0   :     1;
           r    1    1    1     1   :   1   :     0;
           f    ?    ?    ?     ?   :   ?   :     -;
// j and k cases
//      clk    j    k    pre   cir     state     q_out/next_state
           b    *    ?    ?     ?   :   ?   :     -;
           b    ?    *    ?     ?   :   ?   :     -;
           p    0    0    1     1   :   ?   :     -;
           p    0    ?    1     ?   :   0   :     -;
           p    ?    0    ?     1   :   1   :     -;
           (?0) ?    ?    ?     ?   :   ?   :     -;
           (1x) 0    0    1     1   :   ?   :     -;
           (1x) 0    ?    1     ?   :   0   :     -;
           (1x) ?    0    ?     1   :   1   :     -;
           x    *    0    ?     1   :   1   :     -;
           x    0    *    1     ?   :   0   :     -;
   endtable
endprimitive
```



Summary

- Each row specify a transition on only 1 input
- All no-effect transitions should be stated or the results will be X
- Only 0,1,x,- are allowed for outputs (no ?)
- Z is X in a primitive
- Input order = specification order

State initialization

- **You can use the "initial" command**
- **In reality, it won't initialize itself**
- **You better not count on it in real implementation**

```
primitive d_prim2 (q_out, clock, data);
   output  q_out;
    input  clock, data;

   reg     q_out;
   initial q_out = 0; // Optional -- Declares
                      // initial value of q_out
   table
//   clk     data      state   :  q_out/next state
     (01)     0     :    ?    :       0;          // Rising clock edge
     (01)     1     :    ?    :       1;
     (0?)     1     :    1    :       1;

     (?0)     ?     :    ?    :       -;          // Falling clock edge

      ?     (??)    :    ?    :       -;          // Steady clock, ignore data
   endtable
endprimitive
```