EECS 427
Lecture 16: Synthesis-Based Design Flow & Verilog Overview

Reading: Inserts E and F, handouts

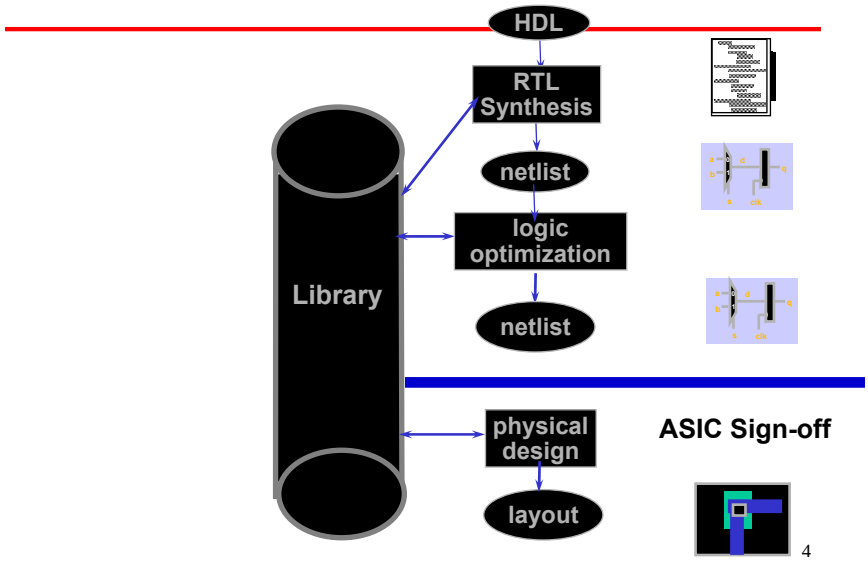
Last Time

- Testing is an important part of designing integrated circuits
- Many engineers specialize in DFT techniques and are always in demand
- Fault models are abstractions of physical defects and are used to assess their impact on circuit behavior
  - Stuck-at 0/1 are most common
  - Test vectors can be created to determine whether a node is actually stuck at 0 or 1
- Key design for test techniques include:
  - Scan: load data into registers, run through logic, then scan out to compare to expected result
  - Self-test (or built-in self-test BIST): Incorporate everything on-chip which eases testing equipment requirements but requires lots of design effort

Lecture Overview

- A peek into libraries
- Verilog overview
  - Many of you have seen this before, but a review is helpful for CAD8

Traditional Flow



Purpose of the Library

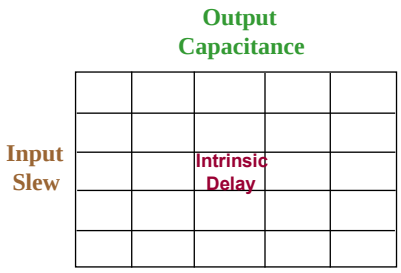
- The Library contains the cells of the technology (.13um)
  - Cells are “Building Blocks” for the circuit
  - Must use technology library to estimate physical properties
- Synthesis tools considers properties and function of cells
  - The key properties:
    - Cell delay
    - Rise/fall transitions
    - Capacitive load
    - Drive strength
    - Area and power
- $Delay_{Total} = Delay_{Cell} + Delay_{Wire}$

Contents of a Library

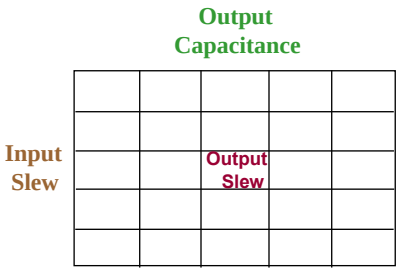
- Units (V, A, pW, KOhm, nS, etc)
- Default parameters
  - Max transition
  - Input pin cap
  - Wireload mode
  - Operating condition
  - Max fanout
- Nominal Parameters (PVT)
- Operating Conditions
  - Worst Case /Best Case
- Scaling factors
  - K Factors
- Wireload Models
  - Estimate for fan-in, fan-out
- Look-up table templates
- Cells: all properties & attributes, Delay Tables, Rise/Fall Transition Tables, Power Tables

Non-linear effects reflected in tables

- $D_G = f(C_L, S_{in})$ and $S_{out} = f(C_L, S_{in})$
  - Non-linear
- Interpolate between table entries
- Interpolation error is usually below 10% of SPICE



Delay at the gate output



10-90% slew rate

Timing Library Example (.lib)

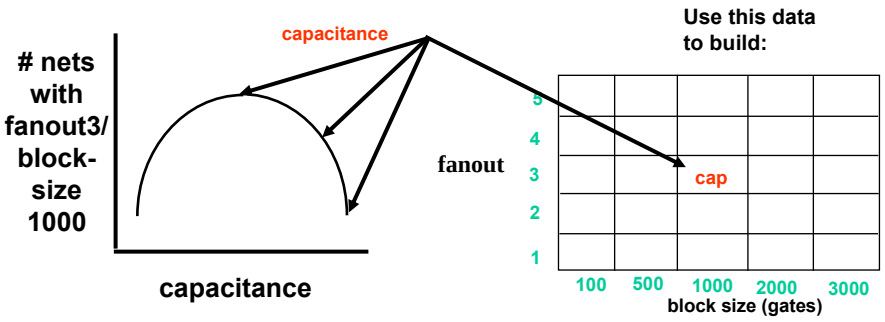
```
library(my_lib) {
  delay_model : table_lookup;
  library_features (report_delay_calculation);
  time_unit : "1ns";
  voltage_unit : "1V";
  current_unit : "1mA";
  leakage_power_unit : 1uW;
  capacitive_load_unit(1,pf);
  pulling_resistance_unit : "1kohm";
  default_fanout_load : 1.0;
  default_inout_pin_cap : 1.0;
  default_input_pin_cap : 1.0;
  default_output_pin_cap : 0.0;
  default_cell_leakage_power : 0.0;

  nom_voltage : 1.08;
  nom_temperature : 125.0;
  nom_process : 1.0;
  slew_derate_from_library : 0.500000;

operating_conditions("slow_125_1.08") {
  process     : 1.0 ;
  temperature : 125 ;
  voltage     : 1.08 ;
  tree_type   : "worst_case_tree" ;
}
default_operating_conditions : slow_125_1.08 ;

lu_table_template("load") {
  variable_1 : input_net_transition;
  variable_2 : total_output_net_capacitance;
  index_1( "1, 2, 3, 4" );
  index_2( "1, 2, 3, 4" );
}
```

```
cell("INV") {
pin(A) {
    max_transition : 1.500000;
    direction : input;
    rise_capacitance : 0.0739000;
    fall_capacitance : 0.0703340;
    capacitance : 0.07278646;
  }
  pin(Z) {
    direction : output;
    function : "!A";
    max_transition : 1.500000;
    max_capacitance : 5.1139;
    timing() {
      related_pin : "A";
      cell_rise(load) {
        index_1( "0.0375, 0.2329, 0.6904, 1.5008" );
        index_2( "0.0010, 0.9788, 2.2820, 5.1139" );
        values ( \
          "0.013211, 0.071051, 0.297500, 0.642340", \
          "0.028657, 0.110849, 0.362620, 0.707070", \
          "0.053289, 0.165930, 0.496550, 0.860400", \
          "0.091041, 0.234440, 0.661840, 1.091700" );
      }
      cell_fall(load) {
        index_1( "0.0326, 0.1614, 0.5432, 1.5017" );
        index_2( "0.0010, 0.4249, 3.6538, 8.1881" );
        values ( \
          "0.009472, 0.072284, 0.317370, 0.688390", \
          "0.009992, 0.095862, 0.360530,  0.731610", \
          "0.009994, 0.126620, 0.477260,  0.867670", \
          "0.009996, 0.144150, 0.644140,  1.127700" );
      }
```

```
fall_transition(load) {
      index_1( "0.0326, 0.1614, 0.4192, 1.5017" );
      index_2( "0.0010, 0.4249, 2.1491, 8.1881" );
      values ( \
        "0.011974, 0.071668, 0.317800, 1.189560", \
        "0.033212, 0.101182, 0.328540,  1.189562", \
        "0.059282, 0.155052, 0.389900, 1.202360", \
        "0.162830, 0.317380, 0.628160, 1.441260" );
    }
    rise_transition(load) {
      index_1( "0.0375, 0.1650, 0.5455, 1.5078" );
      index_2( "0.0010, 0.4449, 1.7753, 5.1139" );
      values ( \
        "0.016690, 0.115702, 0.418200, 1.189060", \
        "0.038256, 0.139336, 0.422960,  1.189081", \
        "0.076248, 0.213280, 0.491820, 1.203700", \
        "0.170992, 0.353120, 0.694740, 1.384760" );
    }
  }
```

Modeling Interconnect -Wire Load Models

- Synthesis and logic optimization rely on gross estimates of interconnect capacitance gathered from empirical data



Wireload model

```
wire_load("45Kto75K") {
        capacitance : 0.000070;
        resistance : 0.000042;
        area : 0.28;
        slope : 40.258665;
        fanout_length(1, 40.258865);
        fanout_length(2, 80.517750);
        fanout_length(3, 120.776600);
        fanout_length(4, 161.045450);
        fanout_length(5, 241.543200);
        fanout_length(6, 322.070900);
        fanout_length(7, 402.587600);
    }
```

Online Verilog Resources

- ASICs the book, Ch. 11:
  - http://www.ge.infn.it/~pratolo/verilog/VerilogTutorial.pdf
- Verilog Quick Reference Guide:
  - http://www.sutherland-hdl.com/online_verilog_ref_guide/vlog_ref_top.html
- Alternate Verilog FAQ:
  - http://www.angelfire.com/in/verilogfaq/index.html
- Verilog Introduction
  - http://www.see.ed.ac.uk/~gerard/Teach/Verilog/index.html
- Newsgroup:
  - http://groups.google.com/groups?group=comp.lang.verilog

Topic Outline

- Introduction
- Verilog Background
- Connections
- Modules
- Procedures
- Structural
- Behavioral
- Testbenches
- Simulation

Most slides courtesy of Andrew Kahng, UCSD

Verilog Overview

- Learn Verilog basics
  - Hardware Description Language semantics
  - Verilog Syntax
  - Features
- Behavioral vs. structural Verilog
- Synthesizable Verilog (subset of Verilog itself)
- A few small examples
  - Larger examples to be shown in discussion

High-level view of Verilog

- Verilog descriptions look like programs:

C/C++	Verilog
Function	Module
Procedure Parameters	Ports
Variables	Wires/Regs

- Modules resemble subroutines in that you can write one description and use (instantiate) it in multiple places
- Block structure is a key principle
  - Use hierarchy/modularity to manage complexity
- But they aren't 'normal' programs
  - Module evaluation is concurrent (every block has its own "program counter")

Introduction - Motivation

- Generic HDL uses:
  - Simulation
    - Test without build
    - ModelSim
  - Synthesis (build)
    - Real hardware (gates)
    - Design Compiler

Quick Verilog History

- Verilog HDL (Hardware Description Language) was developed by Gateway Design Automation in 83-84
- Put in the public domain by Cadence Design Systems in 1990 to promote the language as a standard
  - Became an IEEE standard in 1995

Hardware Description Languages

- Need a description one level up from logic gates
- Work at the level of functional blocks, not logic gates
  - Complexity of the functional blocks is up to the designer
  - A functional unit could be an adder, or even a microprocessor
- The description consists of functional blocks and their interconnections
  - Describe functional block (not predefined)
  - Support hierarchical description (function block nesting)
- To make sure the specification is correct, create a testbench and run it through ModelSim (or similar)

Slide courtesy of Ken Yang, UCLA

Verilog Naming Conventions

- The following is used in all code:
  - Two slashes "//" are used to begin single line comments
    - However "// synopsys" is a directive to Design Compiler to do something (we'll show the most common example later)
  - A slash and asterisk "/*" are used to begin a multiple line comment and an asterisk and slash "*/" are used to end a multiple line comment.
  - Names can use alphanumeric characters, the underscore "_" character, and the dollar "$" character
  - Names must begin with an alphabetic letter or the underscore.
  - Spaces are not allowed within names
- Parameter naming; use compiler directives
  - `` `define word_size 16 ``
  - Whenever you see `` `word_size `` → this will be interpreted as 16

Reserved Keywords

- The following is a list of the Verilog reserved keywords

```
always      endmodule     medium      reg         tranif0
and         endprimitive  module      release     tranif1
assign      endspecify    nand        repeat      tri
attribute   endtable      negedge     rnmos       tri0
begin       endtask       nmos        rpmos       tri1
buf         event         nor         rtran       triand
bufif0      for           not         rtranif0    trior
bufif1      force         notif0      rtranif1    trireg
case        forever       notif1      scalared    unsigned
casex       fork          or          signed      vectored
casez       function      output      small       wait
cmos        highz0        parameter   specify     wand
```


Reserved Keywords (continued)

```
deassign      highz1pmos  param     spec      weak0
default       if          posedge   strength  weak1
defparam      ifnone      primitive strong0   while
disable       initial     pull0     strong1   wire
edge          inout       pull1     supply0   wor
else          input       pulldown  supply1   xnor
end           integer     pullup    table     xor
endattribute  join        remos     task
endcase       large       real      time
endfunction   macromodule realtime  tran
```

Numbers

- Number notation:
  - <size> ‘ <base format><number>
- Examples:
  - `4’b1111 // 4 bit binary number`
  - `12’habc //12 bit hexadecimal number`
  - `16’d255 //16 bit decimal number`
- Z is high impedance, X is don’t care, ? = 0 or 1 or X

Operators

- Arithmetic
  - `* multiply`
  - `/ divide`
  - `+ add`
  - `- subtract`
  - `% modulus`
- Logical
  - `! Not`
  - `&& and`
  - `|| or`
- Relational
  - `> greater`
  - `< less`
  - `>= greater-equal`
  - `<= less-equal` (also used for non-blocking assignments, later)
- Equality
  - `== equal`
  - `!= not equal`
  - `===  (case equality)`
- Bitwise
  - `~ negation`
  - `& and`
  - `\ or`
  - `^ xor`
  - `^~ xnor`

Connections: Ports

- Keywords:
  - `input` - input
  - `output` - output
  - `inout` - bi-directional
- Ports do not store information
- Example

```
module ex (a, b, c, out)
  output out;
  input a, b, c;
endmodule
```


Wires

- Wires
  - Connection between hardware elements (visualize as a node in the circuit)
  - Module connections
  - Used to connect signals from sensitivity list
  - Memoryless
    - Must be continuously driven by an assignment statement (**assign)**
  - Assigned outside of always blocks
- Example:

```
wire a; // declared wire net
wire b = 1'b0 // tied to zero at declaration
```

(alternatively:

```
wire b;
assign b = 1'b0;
```

Memory Elements

- Register
  - Keyword = reg
  - Represents storage in that its value is whatever was most recently (procedurally) assigned to it
    - But it does NOT necessarily instantiate an actual register
  - Assigned within always blocks
- Examples:

```
reg clock; // clock
reg [0:4] vec_reg // 5 bit register vector
```


Modules

- Primary unit in Verilog
  - Functional block (can be big; ex: ALU)
  - Keywords = module/ endmodule
  - Used for all dataflow types

Procedural Statements

- Control statements
  - Keyword `always` provides functionality of a tiny program that executes repeatedly (usually on some trigger condition, more later)
  - Don't assign a value to a specific `reg` in two different always blocks as it will generate 2 FFs and combine outputs
- Inside an `always` block, can use standard control flow statements:
  - `if (<conditional>) then <statements> else <statements>;`
  - `case (<var>) <value>: <statements>; … default: <statements>`
  - Case statements are prioritized
    - The second case entry can't happen unless the first does not match
    - May not be what the actual hardware implies – especially when cases are mutually exclusive
    - Need additional directives (parallel-case, shown later) to indicate this

```
Example:
    always @ (Activation List)
        begin
                        if (x==y) then
                        out= in1
                        else
                        out = in2;
        end
```


`Initial` Block

- A type of procedural block
  - Does not need an activation list
  - It runs just once, when the simulation starts
- Used at the very start of simulation
  - Initialize simulation environment
  - Initialize design
    - This is usually only used in the first pass of writing a design
    - NOT synthesizable, real hardware does not have initial blocks
  - Allows testing of a design (outside of the design module)

Blocking vs Non-blocking

- Relates to scheduling of events
- Blocking
  - Ex:

```
begin
  A = B;
  B = A;
end
```

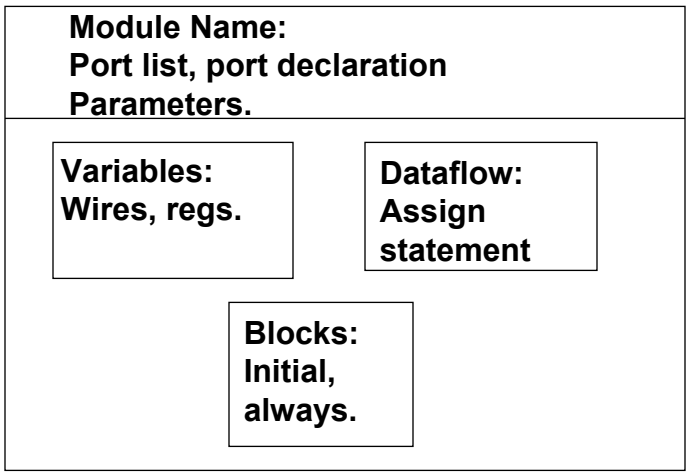
  - Each assignment is completed before moving to the next line
  - In this case, value held in B is assigned to A, and then the value assigned in A (same as in B) is then assigned back to B.
- Non-blocking (preferable in sequential elements)
  - Ex:

```
begin
  A <= B;
  B <= A;
end
```

  - Values on RHS of both expressions are held in temp locations, all assignments are done concurrently → A and B are swapped

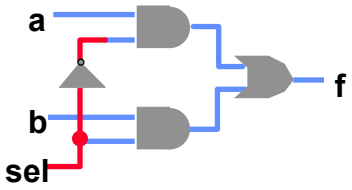
Modules

- Structure



Structural Verilog

- Structural models
  - Are built from gate primitives and/or other modules
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit
  - Basically you are clearly specifying the structure of the circuit
- Identify:
  - Gate instances, wire names, delay from *a* or *b* to *f*.




```
module mux (f, a, b, sel);
   output   f;
   input a, b, sel;

   and #5   g1 (f1, a, nsel),
            g2 (f2, b, sel);
   or    #5 g3 (f, f1, f2);
   not   g4 (nsel, sel);
endmodule
```


Behavioral Modeling

- More abstract, no direct description of how a module is implemented using primitives
- Mux using behavioral: **`assign f = (sel) ? a: b;`**
- *Procedural* statements are used
  - Statements using “always” Verilog construct
  - Can specify both combinational and sequential circuits
- Normally don’t think of procedural stuff as “logic”
  - They look like C: mix of ifs, case statements, assignments …
  - … but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)

Slide courtesy of Don Thomas, Carnegie Mellon

Behavioral Statements

- if-then-else
  - What you would expect, except that it's doing 4-valued logic. 1 is interpreted as True; 0, x, and z are interpreted as False

```
if (select == 1)
     f = in1;
else f = in0;
```

- case
  - What you would expect, except for 4-valued logic
  - There is no *break* statement — it is assumed
  - Casex statement treats Z/X as don't cares

```
case (select)
  2'b00: a = b + c;
  2'b01: q = r + s;
  2'bx1: r = 5;
  default: r = 0;
endcase
```

Slide courtesy of Don Thomas, Carnegie Mellon

Activation Lists

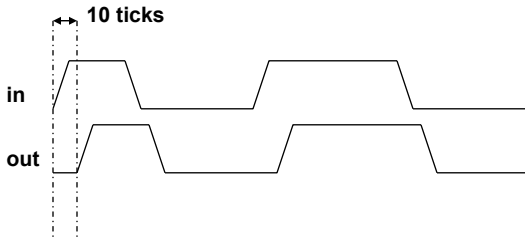
- Contained in `always` block
- Definition: Activation List
  - Tells the simulator when to run this block
    - NOTE!! If not all inputs are sensitized, a latch is created to hold state in those undefined cases
- Activation lists in Verilog:
  - `@(signalName or signalName or …)`
    - Evaluate this block when any of the named signals change (either positive or negative change)
  - `@(posedge signalName);` or `@(negedge signalName);`
    - Makes an edge triggered flip-flop
    - Evaluates only on one edge of a signal
    - Can have `@(posedge signal1 or negedge signal2)`
    - Only allow "or" not "and" because edges are singular events

Testbenches: Delay Models

- Verilog simulation time
  - Execution time of the verilog model
    - When the computer completes with all the “events” that occur at the current simulated time
    - The computer increases time until another signal is scheduled to change values
- Behavioral delay assignments within the blocks
  - `# delayAmount`
    - Simulator sees this symbol, and stops evaluation
    - Pause `delayAmount` of simulated time (# of ticks)
    - Delays are often used to model the delay in functional units
    - Can be tricky to use properly
  - Synthesis does not deal with delays (it computes delays itself)
    - Use only in testbench
    - Synthesizer will ignore

Declarative Delay Control

- A way to specifying delay of a signal
- Make out a delayed version of the input (by 10 ticks)
  - `assign #10 out = in;`
  - Delayed assignment
- Anywhere else to put delay is not allowed
  - `assign out = #10 in;` // is not allowed



Slide courtesy of Ken Yang, UCLA

Building and testing a module

- Construct a "testbench" for your design
  - Develop your hierarchical system within a module that has input and output ports (called "design" here)
  - Develop a separate module to generate tests for the module ("test")
  - Connect these together within another module ("testbench")

```
module testbench ();
   wire   l, m, n;

   design d (l, m, n);
   test   t (l, m);

   initial begin
      //monitor and display
      …
```

```
module design (a, b, c);
   input  a, b;
   output c;
   …
```

```
module test (q, r);
   output    q, r;

   initial begin
     //drive the outputs with signals
      …
```

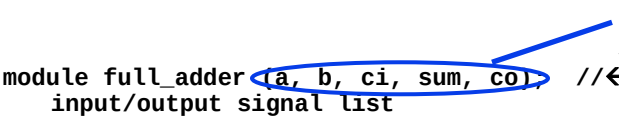

Slide courtesy of Don Thomas, Carnegie Mellon

Examples: Creating Code

- Example:
  - Given a specification – "build full adder"
  - Name signals:
    - Inputs: carry_in, A, B
    - Outputs: carry_out, sum
  - Math:
    - Sum = (A xor B xor carry_in)
    - Carry_out = (A · B) + carry_in · (A · B)
  - Need:
    - Module name
    - Algorithm (see math)

Full-adder Code

- Sample Code




```
module full_adder (a, b, ci, sum, co);  //lists full
   input/output signal list
input  a, b, ci;      //input declaration
output sum, co;       //output declaration
    assign sum = a ^ b ^ ci;
    assign co = (a & b) | (a & ci) | (b & ci);
endmodule
```


Positive edge-triggered registers with resets

```
module ff1(d,clk,reset,q)
 input d, clk, reset;
 output q;
 reg q;

always @(posedge clk)
  if (reset == 1)
      q <= 0;
  else
      q <= d;
```

OR

```
always @(posedge clk or posedge reset)
  if (reset) q <= 0;
  else
      q <= d;

endmodule
```