ECE 451 Verilog Tutorial

James Barnes
(jobarnes@engr.colostate.edu)

Outline

- HDL overview, design flow
- Hierarchical design and design for verification
- Numbers, data types and operators
- Builtin primitives (NAND, NOR,…)
- Control Flow (if-else,…)
- Continuous and Procedural Assignments
- Behavioral coding vs coding for “synthesis”
- Simulator behavior and avoiding problems
  - Unintended latches
  - How to avoid mismatch
- Some simple examples
- Misc – system tasks, parameters, defines
- Behavioral modelling - delays
- FSM example

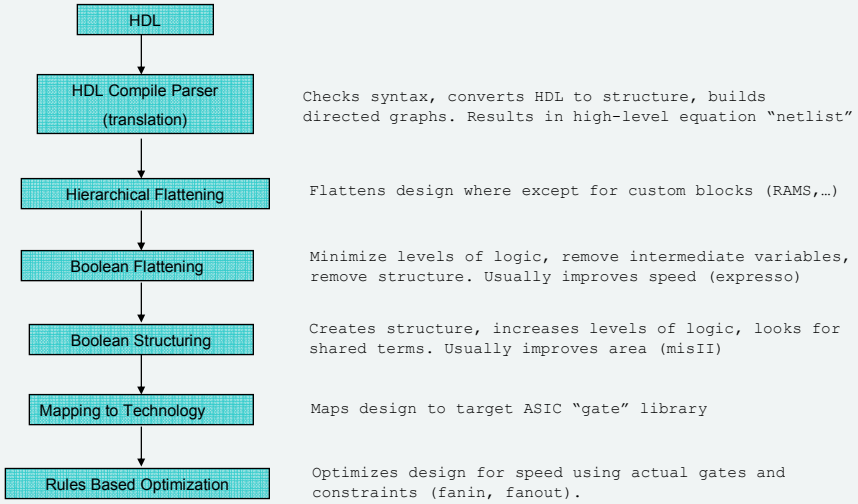
What is /is-not Covered

- We will learn verilog primarily through examples.
- Emphasis is on features used in writing synthesizable verilog.
- A few other topics will be covered, but only briefly.
- You will need to continue learning verilog to become familiar with all its features.

HDL Overview

- Two main HDLs (verilog, VHDL)
- HDLs resemble programming languages (C/C++) but model hardware concurrency (which sometimes leads to unexpected program flow).
- HDL models fall into two types
  - Behavioral models. These used for:
    - High-level models of complex systems.
    - Testbenches (but SystemC … becoming more prevalent)
    - "Blackboxing" analogish components (PLLs, CDRs, memories)
  - Behavioral models are not synthesizable.
  - Sythesizable models ("RTL" style). This is a coding style that a synthesis tool, for example synopsys, can map to circuit blocks.
    - If you put non-synthesizable constructs in your code, the synthesis tool may silently ignore them, which will lead to a behavior mismatch between the simulation and the synthesized circuit.

Synthesis tool flow (ASICs)



FPGA "synthesis"

- FPGAs contain a number of pre-assembled complex logic blocks.
  - Counters, arithmetic blocks, RAMs, muxes, look-up tables, flops
  - AND-OR gate arrays of uncommitted logic which can be programmed to produce a wide range of logic functions
- Your verilog must be written in a form such that the synthesis tool will recognize the functions
- Rules/restrictions are similar to ASIC synthesis.

Verilog supports hierarchical design

- Hierarchical advantages
  - Manage complexity
  - Promote design reuse
  - Allow parallel development
- Hierarchical features in verilog
  - modules
  - ports (connection to modules)

Structure of a module

```
module full_adder(ci,a,b,sum,cout);
// port declarations
   input a,b,ci;
   output sum,cout;
   // type declarations.
   wire  a, b, ci, sum,cout;
   // assignments
   assign sum = a ^ b ^ ci;
   assign cout = (a & b) | (a & ci) | (b & ci);
endmodule
```

Syntax notes

- Statements end with ;
- Compound statements (see later) are delimited by `begin end` (like `{ }` in C).
- Port directionality and width declared.
- Variable types must be declared, as in other programming languages.

Instantiation of a module – 4 bit adder slice

```
module adder4(xsum, xcout, xa, xb, xci);
   input [3:0] xa, xb;
   input       xci;
   output [3:0] xsum;
   output      xcout;
   wire [3:0]  xa, xb, xsum;
   wire        xci, xcout;
   wire [2:0]  cout_int; // Internal signal
   full_adder a0( .sum        (xsum[0]),
                  .cout       (cout_int[0]),
                  .a          (xa[0]),
                  .b          (xb[0]),
                  .ci         (xci));
   full_adder a1( .sum        (xsum[1]),
                  .cout       (cout_int[1]),
                  .a          (xa[1]),
                  .b          (xb[1]),
                  .ci         (cout_int[0]));
   full_adder a2( .sum        (xsum[2]),
                  .cout       (cout_int[2]),
                  .a          (xa[2]),
                  .b          (xb[2]),
                  .ci         (cout_int[1]));
   full_adder a3( .cout       (xcout),
                  .sum        (xsum[3]),
                  .a          (xa[3]),
                  .b          (xb[3]),
                  .ci         (cout_int[2]));
endmodule // adder4
```

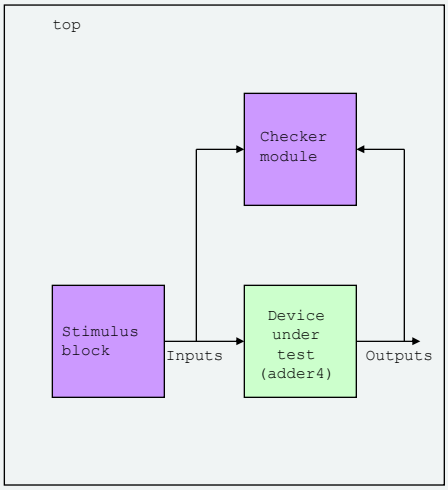
adder4 testbench – first version

```
`timescale 1ns/1ps

module tb; // No ports needed!
   reg xci;
   reg [3:0] xa, xb;

   wire [3:0] xsum;
   wire       xcout;

   // Instantiate 4 bit adder

   adder4 my_adder(// Outputs
                   .xsum(xsum[3:0]), .xcout(xcout),
                   // Inputs
                   .xa(xa[3:0]), .xb(xb[3:0]), .xci(xci));

   // Stimulus
   initial
     begin
              xa = 4'h0; xb = 4'h0; xci = 1'b0;
            #5 xa = 4'h0; xb = 4'h0; xci = 1'b1; // #5 -> wait 5ns, then execute stmt
            #5 xa = 4'h0; xb = 4'h1; xci = 1'b0;
            // yada yada yada

     end // initial begin

endmodule // tb
```

How to test?

- Looking at waves or logfiles is tedious and error-prone.
- Better to build independent verification components which can stimulate and check the results of the block under test. “Self-checking” test.
  - Must use independent means for implementing block functionality. In this case, can use simulator’s built-in arithmetic operators to check adder.
  - Other approaches:
    - System C verification library.
    - Vendor-supplied verification IP: bus functional models, 0-in components,…

A better test jig



- Verification components can be written in behavioral verilog, systemC, Verisity “E language”…
- adder4 must be written in synthesizable verilog (“RTL”)

adder4 stimulus module

```
`timescale 1ns/1ps
module stimulus(xa, xb, xci);

   output [3:0] xa, xb;
   output       xci;

   reg [3:0] xa, xb;
   reg       xci;

   integer    i, j, k;
   // Stimulus - generate all input combinations
   initial
     begin
          for (i=0; i<16; i=i+1)
            for (j=0; j<16; j=j+1)
              for (k=0; k<2; k=k+1)
                begin
                     xa = i; xb = j; xci = k;
                     #5;
                end
     end // initial begin
endmodule // stimulus
```

adder4 checker

```
`timescale 1ns/1ps
module check_adder(xci, xa, xb, xcout, xsum);

   input [3:0] xa, xb;
   input       xci;
   input [3:0] xsum;
   input       xcout;

   wire [4:0]  xa, xb, xsum;
   wire        xci, xcout;
   reg [4:0]   in_result;
   reg [4:0]   out_result;
   reg         error;

   always @(xa or xb or xci or xcout or xsum)
     begin  // Predicted xsum using Verilog's addition
            in_result = xa + xb + {3'b0,xci};
            out_result = {xcout,xsum};
     end

   always @(in_result or out_result)
     begin  // Compare predicted with actual
            error <= (in_result != out_result);
            if (error)
              $display($time, "  Error: xa=%h, xb=%h, xci=%b, xsum = %h, xcout = %b",
                           xa, xb, xci, xsum, xcout);
     end

endmodule // check_adder
```

adder4 top-level test jig

```
module top;

   wire [3:0] xa, xb, xsum;
   wire       xci, xcout;

   // Instantiate 4 bit adder
   adder4 my_adder(// Outputs
                   .xsum(xsum[3:0]), .xcout(xcout),
                   // Inputs
                   .xa(xa[3:0]), .xb(xb[3:0]), .xci(xci));

   // Instantiation checker
   check_adder my_checker(// Inputs
                          .xa(xa[3:0]), .xb(xb[3:0]), .xci(xci),
                          .xsum(xsum[3:0]), .xcout(xcout));

   // Instantiate stimulus block
   stimulus my_stim(// Outputs
                    .xa   (xa[3:0]),
                    .xb   (xb[3:0]),
                    .xci(xci));

endmodule // top
```

Number representations

- Sized numbers
  - `1'b1, 4'b1010, 4'b0x0z` – binary
    - X=unknown, z=high impedance
  - Other radixes
    - `3'o4` – octal
    - `4'hE` - hex
    - `4d'11` - decimal
  - The number in front of the ' represents the bit width of number when expressed as a binary, regardless of the radix used
  - Verilog performs arithmetic on sized numbers using 2's complement arithmetic.
  - If size parameter is omitted, defaults to max width (>-32b wide)
- Integers
  - At least 32b wide (simulator-dependent)
  - Signed, i.e. can write `-10`
- Real
  - `27.3, 4.02e+03`
  - Internal rep as 64b integer

Operators

Type	Width of Result	Symbol
Arithmetic	>Ops width	+ - * / %
Logical	1 bit	! && ||
Bitwise	Ops width	~ & | ^ ~^
Relational	1 bit	> < >= <=
Equality	1 bit	== != === !==
Reduction	1 bit	& ~& | ~| ^ ~^
Shift	Ops width	>> <<
Concatenate, replicate	> Ops width	{ , , } {{ }}
Conditional	Ops width	? :

```
4'b0110 - 4'b0111
yields 5'b11111
```

```
Note: use ? for
don't cares
```

```
Note: vacant
positions zero-
filled.
```

Some data types

- `wire`
  - Used to represent connections between blocks
  - No "memory" – value assigned in "continuous assignment" statement.
    - Right-hand side can be of `reg` (see below) or net type.
  - `wire` is most commonly used member of net class
    - Others are `wand, wor, tri, triand, trior, trireg`.
  - Used for combinational logic
  - Limited conditional assignment language features
- `reg`
  - Has "memory" but doesn't NECESSARILY imply a hardware register.
  - Assigned in "procedural assignment" block.
    - Right-hand side can be of `reg` or net type.
  - "Blocking" vs "non-blocking" assignments (more on this later)
  - Richer set of conditional assigments
- `real, realtime`
  - For floating point numbers, but represented internally as 64b integers
  - Supports scientific as well as real (XXX.XX) notation
  - Used mainly for behavioral modelling.

Scope

- Variables (`wire, reg,…`) have local scope (within module). No global variables in verilog.

Hierarchical Reference to Signals

- Signals down within a hierarchy can be referenced as:

  `a.b.c.<sig_name>`

  where `a,b,c` are module instance names and `sig_name` = signal name
- Example: at the level of module `top`, internal signal `cout_int[1]` within instance `my_adder` of module `adder4` can be referenced by `my_adder.cout_int[1]`
- Cannot assign (change) a signal thru hierarchical reference, only test. Would only be used in test.

Example wire assignments

```
wire a;
assign a = 1'b0; // a is assigned a constant value for the duration of sim
```

```
// Define a 4b wide bus
wire [3:0] a = 4'h0; // Declaration and assignment on one line.
```

```
wire a,b,c,f;
assign f = (~a) & b & c | a & (~b) & c; // Two level logic
```

```
wire in1, in0, sel;
wire f = sel ? in1 : in0; // Mux with conditional assignment
```

```
wire in1, in0, sel, f;
assign f = sel & in1 | (~sel) & in0; // Another mux
```

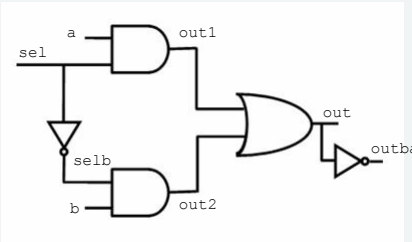
```
wire [15:0] a, b;

assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]}; // Byte swap
```

Verilog Built-in Logic Primitives

- Can also build logic functions by instantiating logic primitives
  - AND(), OR(), NAND(), NOR(), XOR(), XNOR(), BUF(), NOT()
- Port connections via an ordered list instead of by name.
  - Allows variable number of inputs for most gates (AND,…). Output is always first port
  - Some gates, such as BUF() can drive more than one output from single input. For these, input is last port.
- Tristate gates BUFIF1(), BUFIF0() tristate their outputs when a control port is 0 or 1 respectively.
- Primitives may be instantiated as named or un-named gates
- These primitives are not widely used in synthesized designs.

Verilog Primitive Example

```
module mux2_1 (a, b, out, outbar, sel);
  input a, b, sel;
  output out, outbar;
    wire out1, out2, selb;
    and a1 (out1, a, sel);
    not i1 (selb, sel);
    and a2 (out2, b , selb);
    or o1 (out, out1, out2);
    assign outbar = ~out;
endmodule
```



Operator Examples

```
wire [3:0] a = 4'b0110;
wire [3:0] b = 4'b0101;

wire [3:0] s = a & b; // Yields s = 4'b0100 (bitwise AND)
wire       t = a && b;  // Yields t = 1'b1 (logical AND)
wire       u = |a; // Reduction op | yields u = 1'b1; short for (a > 0)
wire       v = ~|a; // Reduction op ~| yields v = 1'b0; short for (a == 0)
wire       w = &a; // Reduction op & yields w = 1'b0; This tests for all ones
wire       a_gt_b = (a > b); // Yields a_gt_b = 1'b1;
```

- Operators can also be used with `reg` data type

RTL control flow constructs

- Conditional assignment (`? :`), used with continuous assignment
- Used in procedural assignments:
  - `if-else`
    - Compound statements in branches delimited by `begin end`
    - `if-else` can result in a priority encoder (slow).
  - `case, casex, casez`
    - `casex` allows use of don't cares (ex: `4'b0?0?` will be matched by `4'b0001`, `4'b0101`, `4'b0100`, `4'b000X`, `4'b000Z`, …)
    - `casez` is similar except that don't care positions (`?`) matched only by `0`,`1`,`Z` (`X` will not match).
    - Any type of case may result in a priority encoder, since the first match will cause the match to terminate. Mutually-exclusive match conditions will prevent priority encoding.
  - `for` loop. Synthesis tool will unroll.
  - Others (`while, repeat`) which I hardly ever used.
- `forever` is a simulator control construct and not synthesizable.

Procedural Assignments

- Used to assign `reg, integer, real` data types
- Two types of procedural blocks
  - `initial` block
    - Triggers (starts execution) once at Time = 0.
    - Synthesis tools ignore initial blocks. Should NOT be found in synthesizable verilog.
  - `always @()` block
    - Triggers any time a variable in sensitivity list has a value change.
    - Can be used to create combinational or sequential logic.

INITIAL block example (NOT synthesizable)

```
`timescale 1ns/1ps
module clk_gen(clk);

   // One way to create a clock with default frequency 100 MHz.
   // Not synthesizable - Only for a test component

   parameter clk_half_period = 5; // Parameterize the period so we can change it.
   output clk;

   reg      clk;

   initial
     begin
          clk = 0;
          forever
            clk <= #clk_half_period ~clk; // This loop continues until something
                                          // terminates the simulation
     end

   initial
     begin
          $monitor ($time, " clk=%b",clk);
          #1000 $finish; // OK, enough already. Stop the simulation
     end

endmodule // clk_gen
```

ALWAYS block for combinational logic

- ALWAYS blocks often used for combinational logic because of richer set of control statements which compared with continuous assignments (`wire`).

```
module mux(ina, inb, inc, ind, sel, out);

   input [3:0] ina, inb, inc, ind;
   input [1:0] sel;
   output [3:0] out;

   wire [3:0] ina, inb, inc, ind;
   wire [1:0] sel;
   reg [3:0]    out;
// Sensitivity list of the always block must include all "inputs" to always block
   always @(ina or inb or inc or ind or sel) // (…) contains the sensitivity list
     case (sel)
       2'b00: out = ina;
       2'b01: out = inb;
       2'b10: out = inc;
       2'b11: out = ind;
     endcase // case(sel)

endmodule // mux
```

- Statements within "always block" are executed sequentially. In this case, because the `case` conditions are mutually exclusive, order doesn't matter

ALWAYS block for combinational logic – accidental "inferred latch"

- Omitting a clause in a control construct (case, if-else) can lead to an unintended "inferred latch". This is almost always an error.

```
module mux(ina, inb, inc, ind, sel, out);

   input [3:0] ina, inb, inc, ind;
   input [1:0] sel;
   output [3:0] out;

   wire [3:0] ina, inb, inc, ind;
   wire [1:0] sel;
   reg [3:0]    out;

   always @(ina or inb or inc or ind or sel)
     case (sel)
       2'b00: out = ina;
       2'b01: out = inb;
       2'b10: out = inc;
       2'b10: out = ind; // TYPO. This line not executed and 2'b11 case missing.
     endcase // case(sel)

endmodule // mux
```

- No new value for sel=2'b11. Simulation will use previous value and synthesis tool will interpret that as an inferred latch. Some lint tools will catch this

Another way to accidentally infer a latch

- Forgetting an item in the sensitivity list. The always block only executes when an item in the sensitivity list changes

```
module mux(ina, inb, inc, ind, sel, out);

   input [3:0] ina, inb, inc, ind;
   input [1:0] sel;
   output [3:0] out;

   wire [3:0] ina, inb, inc, ind;
   wire [1:0] sel;
   reg [3:0]    out;

   always @(ina or inb or inc or ind or sel)
     case (sel)
       2'b00: out = ina;
       2'b01: out = inb;
       2'b10: out = inc;
       2'b11: out = ind; // This line will NOT execute when ind changes
     endcase // case(sel)

endmodule // mux
```

- Since `out` is a reg type, it will hold its previous value when `ind` changes.
- Because this is such a common error, later versions of verilog allow a shorthand notation `always @(*)` which will be filled in with the appropriate variables

A way to avoid an accidental inferred latch

- Precede all assignments with a default value (and include all items in the sensitivity list)

```
module mux(ina, inb, inc, ind, sel, out);

   input [3:0] ina, inb, inc, ind;
   input [1:0] sel;
   output [3:0] out;

   wire [3:0] ina, inb, inc, ind;
   wire [1:0] sel;
   reg [3:0]    out;

   always @(ina or inb or inc or ind or sel)
     begin
        out = 4'h0;             // Default value
           case (sel)
             2'b00: out = ina;  // This is a three input mux
             2'b01: out = inb;
             2'b10: out = inc;
           endcase // case(sel)
     end
endmodule // mux
```

- Simulator knows what value to assign to out even if no `case` condition is satisfied.

Another way to avoid an accidental inferred latch

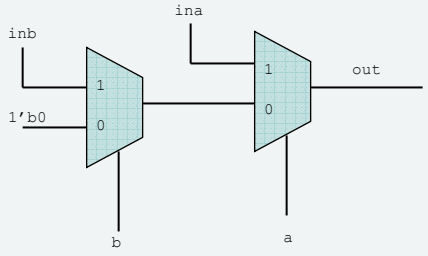
- Fully specify all conditional branches and assign all signals from branches (and include all items in sensitivity list)

```
module mux(ina, inb, inc, ind, sel, out);

   input [3:0] ina, inb, inc, ind;
   input [1:0] sel;
   output [3:0] out;

   wire [3:0] ina, inb, inc, ind;
   wire [1:0] sel;
   reg [3:0]    out;

   always @(ina or inb or inc or ind or sel)
     begin
           case (sel)
             2'b00: out = ina;  // This is a three input mux
             2'b01: out = inb;
             2'b10: out = inc;
             default: out = 4'h0; // Alternately 2'b11: out = 4'h0;
           endcase // case(sel)
     end
endmodule // mux
```

- `out` will always be assigned a value.

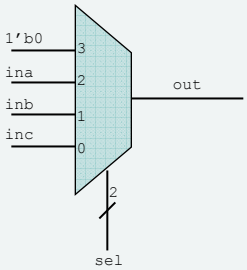
Unintended priority encoder (or maybe this is what you wanted)

```
reg  out;
wire ina, inb, a, b;

always @(ina or inb or a or b)
  begin
     // if statements are not mutually exclusive
     if (a) out = ina;
     else if (b) out = inb;
     else out = 1'b0;
  end
```



Avoid priority by mutually-exclusive if-else conditions or case construct

```
reg  out;
wire ina, inb, inc, a, b;
wire [1:0] sel = {a,b};

always @(ina or inb or inc or sel)
  begin // The if statements are mutually exclusive
     if (sel == 2'b11) out = 1'b0;
       else if (sel == 2'b10) out = ina;
         else if ((sel == 2'b01) out = inb;
           else if ((sel == 2'b00) out = inc;
  end
```



- All inputs have equal priority

Another combinational block – what do you get?

```
reg [63:0] sum;
reg cout;
wire [63:0] ina, inb;

always @(ina or inb)
          {cout,sum} = ina + inb; // + is the addition operator
```

- Synopsis will produce whatever is required to meet timing
  - If you are in a fast process and specify relaxed timing, synopsis will produce the smallest solution, probably a ripple-carry adder.
  - If timing is tight, you will get a full 64b carry-lookahead adder! Synopsis has a library of “prepackaged” logic functions – The Designware© Library – which can implement many common functions
- You can create huge amounts of logic with a few simple statements, e.g. multiply and divide. Be careful ☺.

Yet another combinational block – what do you get?

```
reg [63:0] count_ns, count_r;
wire enable_count;
wire rst, clk;
wire counter_rollover = &count_r;

always @(enable_count or count_r)
   count_ns = enable_count ? count_r + 64'h1 : 64`h0;

always @ (posedge clk)
   count_r <= rst ? 64'h0 : count_ns;
```

- This will produce a synchronous counter

ALWAYS block must be used for sequential logic (Latches and Flip-Flops)

```
module transparent_latch(in, clk, rst, out);

   input in, clk, rst;
   output out;

   wire in, clk, rst
   reg      out;

   // Normally, you would never write this as a synthesized block
   //  (except by accident) because most timing tools
   //  don't work well with transparent latches.

   always @(in or clk or rst)
     // Async reset. Normally avoided like the plague on ASIC designs.
     if (rst)
       out = 1'b0;
     else
       if (clk)
           out = in;
       // No else clause for this if => the latch
       // must hold its value when clk is low and rst is not asserted

endmodule // transparent_latch
```

Inferred D-flop

Synchronous reset

```
   // Infer an 8b wide D-flop bank w/ synchronous rst
   always @(posedge clk)
     out <= rst ? 8'h00 : din;
```

Asynchronous reset

```
   // Infer an 8b wide D-flop bank w/ asynchronous rst
   always @(posedge clk or posedge rst)
     if (rst)
         out <= 8'h00;
     else
         out <= din;
```

- These examples use non-blocking assignment <= , discussed on next slide.

Blocking vs Non-blocking assignments

- = within procedural block is a blocking assignment
  - Blocking assignments within an always block are completed in order, execute only after the previous statement is completed. If there is a delay in the previous statement (see later), the next statement will wait for the delay.
- <= within a procedural block is a non-blocking assignment. Non-blocking assignments do not wait for the previous statement to complete. They can be used to model concurrent operations.
- RTL does not use delays. A more important distinction is the order the simulator evaluates these two assignments …

Verilog Evaluation Queues

- Each Verilog simulation time step is divided into a number of queues, which are evaluated in order. The important ones are:

Time 0:

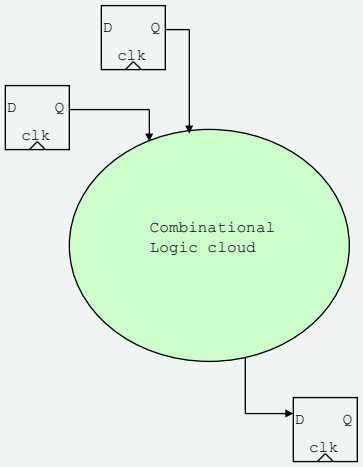
- Q1 — *(in any order)* :
  - Evaluate RHS of all non-blocking assignments
  - Evaluate RHS and change LHS of all blocking assignments
  - Evaluate RHS and change LHS of all continuous assignments
  - Evaluate inputs and change outputs of all primitives
  - Evaluate and print output from $display and $write
- Q2 — *(in any order)* :
  - Change LHS of all non-blocking assignments
- (Remaining queues)
  - Evaluate and print output from $monitor and $strobe

Time 1:

…

Ref: Sutherland, 1996 International Cadence Users Conference.

Avoiding races in simulation (the right way)



- Rule:
  - code combinational logic with continuous assignments (`wire`) or blocking procedural assignments (=)
  - code sequential logic using non-blocking procedural assignment (<=)
- If the assignment rules are followed, on the rising clk edge the old values from the upper flops will be propagated through the combo logic cloud to the input to the lower flop. Only then will the flops be evaluated. NO delays need (or should) be put into the modules.

Simple example – D flop with load and synchronous reset

```
module dff1 (d, ld, rst, clk, q);

   input d, ld, rst, clk;
   output q;
   wire d, ld, rst, clk, q_ns;
   reg      q;

   /* continuous assignment */
   assign q_ns = 1'b0; // WHAT GOES HERE??

   /* procedural assignment. "always" block "fires" on positive edge of clock */
   always @(posedge clk)?
     // <= is non-blocking assignment. Synchronous reset.
     q <= rst ? 1'b0 : q_ns;

endmodule // dff1
```

Another example –T flop

```
module tff (t, rst, clk, q);

   input t, rst, clk;
   output q;

   wire t, rst, clk, q_ns;
   reg q;

   /* continuous assignment */
   assign q_ns = (t ^ q); // ^ = XOR

   /* procedural assignment. "always" block "fires" on positive edge of clock */
   always @(posedge clk)
   // <= is non-blocking assignment. Synchronous reset.
     q <= rst ? 1'b0 : q_ns;

endmodule;
```

Mismatch between pre-synthesis model and synthesis output (gate model and silicon)

- Synthesis tools ignore some behavioral constructs, such as
  - INITIAL blocks – not realizable in hardware
  - Delays inserted into model to fix race conditions. Synthesis tools have a difficult time creating fixed delays. In general, delays are ignored.
- If your module depends on these constructs to simulate properly, there will be a mismatch between pre-synthesis and post-synthesis models (and silicon). This is a VERY BAD thing.
- How to avoid this…

Guidelines for avoiding model mismatch

- INITIAL blocks – replace with synthesizable hardware such as reset circuits, busses to set registers,…
- Don't insert delays to fix races – understand the simulator order of evaluation. At each timestep, the simulator has an evaluation queue.
  - Continuous assignments (wire) and blocking procedural assignments (=) are evaluated first. Both the RHS and LHS are evaluated.
  - The RHS of non-blocking procedural assignments (<=) are also evaluated early.
  - The LHS of non-blocking procedural assignments are updated last.
- Note that in non-synthesized modules such as verification components, you can and should use any legal construct.

More rules to avoid problems

- Do not mix blocking and non-blocking assignments within one `always` block (some synthesis tools will flag this as an error).
- Do not assign the same variable in two separate `always` blocks.
- Some suggested practices:
  - Only define one module per file and make the file name be the same as the module name with the .v extension (or whatever the convention used in the design lab).
  - An exception to this would be if you have a large collection of small blocks which will be shared among a group of designers. In that case, you can create one library file containing all the blocks.

Some system tasks

- $display, $monitor
  - $display prints one line per call. $monitor prints any time a variable in its list changes.
- $stop, $finish
  - $stop stops sim, but it can be restarted from that point.
  - $finish ends sim. In many cases, the sim will stop on its own without a $finish, but if it contains an infinite loop (such as a clock generator), $finish is needed.
- Generic IO tasks $fopen, $fwrite, … Used for saving simulation data.The development environment usually handles this behind the scenes.
- $random – generates a random number.
- $readmemb, $readmemh
  - For initializing memory arrays from a file

parameters and `define statements

- `` `define `` used for readability and compilation control
  - Ex `` `define WORD_SIZE 32 ``
- `` `defines `` are dangerous because they are just a macro substitution by preprocessor. Can be re-defined by other modules. If using, safest to re-define in each module where used (but then you may get compiler warnings).
- `parameter` is safer because
  - can be sized (Ex `parameter [3:0] maxval = 4'hF;`)
  - Scoped only within module. Can be overridden only when module instantiated.
- `parameter` recommended for FSM state mnemonics

parameter example

```
module par_reg(q, d, clk, rst);
   parameter width = 8; // Default width

   input [(width-1):0] d;
   input         clk, rst;
   output [(width-1):0] q;

   reg [(width-1):0] d;

   always @(posedge clk)
     q <= rst ? {width{1'b0}} : d;

endmodule // par_reg
```

parameters can be overridden at instantiation

```
module test_par_reg;

   reg [15:0] d;
   wire [15:0] q;
   wire        clk, rst;

   // Instantiate a 16b reg
   par_reg #(.width(16))
     my_par_reg(.q(q[(width-1):0]), .d(d[(width-1):0]), .clk(clk), .rst(rst));

   clk_gen cgen(.clk (clk));

   initial
     begin
          d=16'h0000;
          rst=1;
          #6 rst=0;
          d=16'h5555;
          #10
            d=16'hAAAA;
          #10
            d=16'hFF00;
          #10
            d=16'h00FF;
     end // initial begin

   initial
     $display($time, " d=%h, q=%h",d,q);

endmodule // test_par_reg
```

Behavioral Modeling -- Delays

- Delays are never required (or recommended) in synthesizable verilog but can be useful in behavioral modules such as stimulus blocks,…
- Any block which uses delay should have a timescale statement.
- Delays can be attached to net types (wire) or incorporated in procedural block assignments.

Some delay usage examples

- Consult a verilog reference before using delays – there are subtle differences in how delays are evaluated depending on how they are put into the code, which we can't cover here..

```
wire a, b;
wire #5 gate_out; // This is a net delay
assign gate_out = ~(a & b); // gate_out will change 5 units after a or b
```

```
wire a, b;
wire gate_out;
assign #5 gate_out = ~(a & b); // gate_out will change 5 units after a or b
```

```
wire a, b;
reg gate_out;
always @(a or b)
   gate_out <= #5 ~(a & b); // gate_out will change 5 units after a or b
```

```
wire in, clk, rst;
reg out;
always @(posedge clk)
   out <= #5 rst ? 1'b0 : in; // models clock to output delay
```

State machine example