18-447 Synthesizable Verilog Tutorial

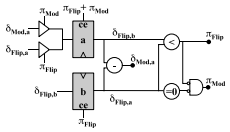
Peter Milder
Dept of ECE, CMU
January 21, 2009

Announcements: none

Handouts: Verilog Tutorial

RTL Verilog Review (The 447 Subset)

Schematic Capture



Hardware Description Language

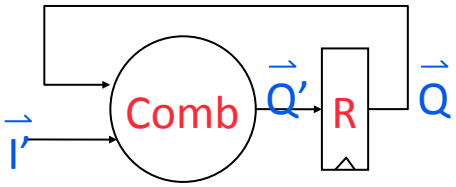
```
always @ (posedge Clk) begin
    if (a >= b) begin
        a <= a - b;
        b <= b;
    end else begin
        a <= b;
        b <= a;
    end
end
```

RT-Level Descriptions

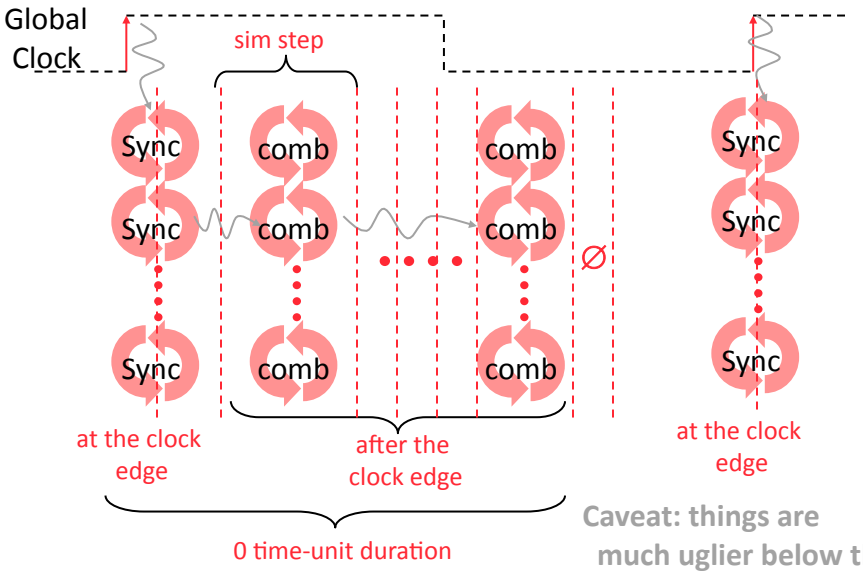
- Textual descriptions of logic circuits
  - different **representation**, but essentially the same **semantics/ abstraction** as in schematic capture

So why is it any better?

- Fine-grain behavior: synchronous **register transfer**
  - a collection of **synchronous** state elements (e.g. registers)
  - a collection of **combinational** logic that computes the next-state (NS or Q') values of state elements based current-state (CS or Q) values of state elements
  - all state elements take on new values simultaneously (implies a globally synchronous clock)



RTL Execution/Timing

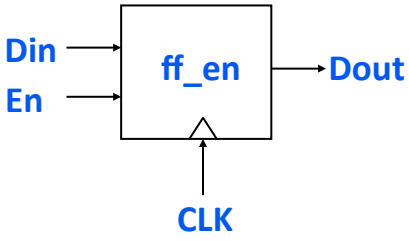


Caveat: things are much uglier below the digital/RTL abstraction

Verilog Modules

- A module with input and output ports

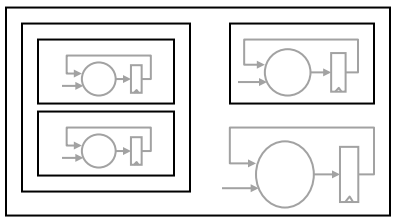
```
module ff_en( input Din, En, CLK,
              output Dout );

       ... body of module ...

endmodule
```



You are NOT writing a Verilog "program".
You are describing hardware in text

Hierarchical Descriptions

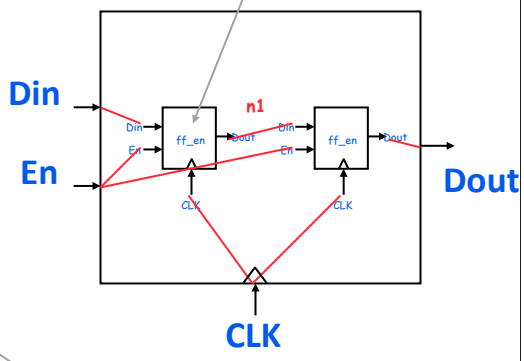
- Coarse-grain structure
  - A hierarchy of modules
  - A module contains RTL logic and/or, recursively, a network of modules interconnected by wires
  - A module is a **convenience**
    - a design encapsulation/abstraction
      partitions a design into manageable chunks
      (vertically and horizontally)
    - reusable design
- How is a hardware module different from a software function?



Module Instantiation and Hierarchy

```
module shift_reg (input Din, En, CLK,
   output Dout );

   wire n1;

   ff_en ff1 (.Din ( Din ),
              .En ( En ),
              .CLK ( CLK ),
              .Dout ( n1 ));

   ff_en ff2 (.Din ( n1 ),
              .En ( En ),
              .CLK ( CLK ),
              .Dout ( Dout ));
endmodule
```

black box: don't need to know what is in ff_en yet



Use **named notation** to wire modules:
More likely to get port hookups right!
E.g., connect **ff1's Dout** to wire **n1**, …

Parameters

- Can parameterize modules:

default parameter value

```
module pmod #(parameter width = 4)
  (input [width-1:0] in,
   output [width-1:0] out);


endmodule
```

width = 4
width = 5
width = 6

```
module topmod( ... );

   pmod               m0(.in(a), .out(b));
   pmod #(5)          m1(.in(c), .out(d));
   pmod #(.width(6))  m2(.in(e), .out(f));


endmodule
```

Arrays of Modules

- Hardware frequently has many arrays
  - Verilog distributes **some** connections across arrays for you
  - 1D arrays only
    - Multi-dimensional arrays can be emulated with "hacks"

```
module big_reg #(parameter width = 4)
  (input [width:0] Din,
   input En, CLK,
   output [width:0] Dout );

   wire [width:0]   n1;

   ff_en ff[width:0] ( .Din ( Din ),
                       .En ( En ),
                       .CLK ( CLK ),
                       .Dout ( n1 ));

endmodule
```

Arrays of modules just like wires

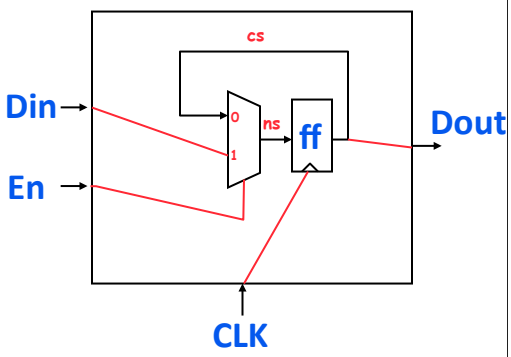
Automatically distributes multi-bit and single-bit inputs (if sizes match)

Combinational Logic

Wires, expressions and assign

```
module ff_en (input Din, En, CLK,
              output Dout);

   wire   cs, ns;

   assign Dout = cs;

   assign ns = En ? Din : cs;

   ff ff1 ( .Din ( ns ),
            .CLK ( CLK ),
            .Dout ( cs ) );

endmodule
```

continuous assignment:
bind wire to an "expression"



Synchronous Sequential Logic

```
module ff (input Din, CLK,
           output Dout);

  reg q;

  assign Dout = q ;

  always @ (posedge CLK ) begin
    q <= Din ;
  end

endmodule
```

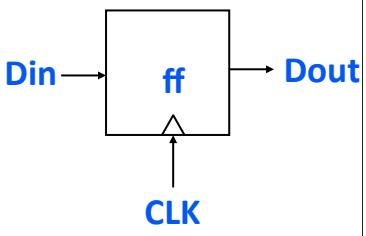
"reg" declares a "stateful" variable (as opposed to a stateless wire) ; reg does not declare a "register"

trigger list

trigger event

delayed assignment symbol

an "always" block (aka process)



A reg variable should only be assigned in *at most* one "always" block!!

Combinational Logic

Procedural Assignment

```
module ff_en (input Din, En, CLK
              output Dout);

  wire   cs;
  reg    ns;

  assign Dout = cs;

  always @ ( Din or cs or En ) begin
    if (En) begin
         ns = Din;
    end else begin
         ns = cs;
    end
  end

  ff ff1 (.Din ( ns ),
          .CLK ( CLK ),
          .Dout ( cs ));

endmodule
```

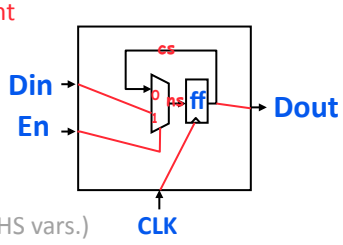
this "reg" is a combinational value

trigger list (must list **all** RHS vars.) (now, can use `always @*`)

describing combinational logic as a sequential program (i.e. how to compute ns based on Din or En).

procedure assignment: ns retains the **last assigned value** at the end of the "always"

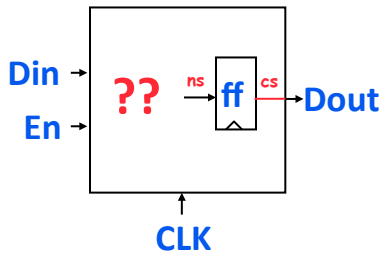
A reg variable should appear on LHS in **at most one** always block!! A LHS reg must be assigned **at least once** following **all possible** control paths in a combinational block!!



Combinational Logic

Procedural Assignments Gone Bad

```
module ff_en (input Din, En, CLK,
              output Dout);

   wire   cs;
   reg    ns;

   assign Dout = cs;

   always @ ( Din or cs or En ) begin
     if (En) begin
          ns = Din;
     end // else begin
     //   ns = cs;
     // end
   end

   ff ff1 (.Din ( ns ),
           .CLK ( CLK ),
            .Dout ( cs ));

endmodule
```



What is missing?

Is this still combinational?

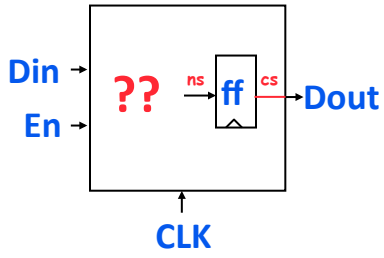
What will the synthesis tool do with this?

Combinational Logic

Procedural Assignments Gone Bad

OK??

```
module ff_en (input Din, En, CLK,
              output Dout);

   wire   cs;
   reg    ns;

   assign Dout = cs ;


   always @ ( Din or cs or En ) begin
     ns = cs;

     if ( En ) begin
          ns = Din;
     end
   end

   ff ff1 ( .Din ( ns ),
            .CLK ( CLK ),
             .Dout ( cs ));

endmodule
```

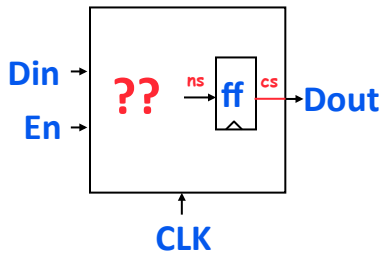


If there is control flow, always begin by providing a default assignment to each assigned LHS. Make it a habit.

Combinational Logic

Procedural Assignments Gone Bad

```
module ff_en (input Din, En, CLK,
              output Dout);

   wire   cs;
   reg    ns;

   assign Dout = cs;

   always @ ( Din or cs /* or En */ )
   begin
     ns = cs ;

     if ( En ) begin
          ns = Din;
     end
   end

   ff ff1 (.Din ( ns ),
           .CLK ( CLK ),
           .Dout ( cs ));

endmodule
```



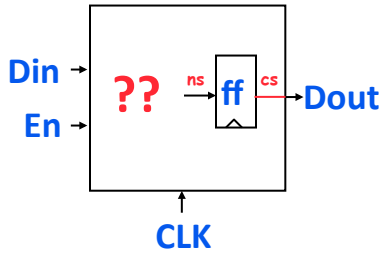
Incomplete sensitivity lists considered harmful!

What happens here?

Combinational Logic

Procedural Assignments Gone Bad

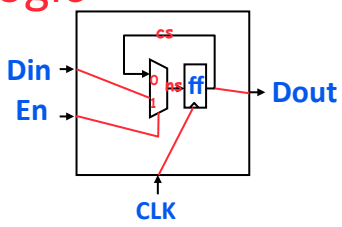
```
module ff_en (input Din, En, CLK
              output Dout);

   wire   cs;
   reg    ns;

   assign Dout = cs;

   always @ ( Din or cs or En )
   begin
     ns <= cs;

     if ( En )begin
          ns <= Din;
     end
   end

   ff ff1 (.Din ( ns ),
           .CLK ( CLK ),
           .Dout ( cs ));

endmodule
```



What is the value of ns here?

Combining Combinational and Sequential Logic

```
module ff_en (input Din, En, CLK
              output Dout);

   reg    Dout;       // sequential
   reg    Dout_next;  // combinational

   always @ ( Din or Dout or En ) begin
     if ( En ) begin
          Dout_next = Din;
     end else begin
          Dout_next = Dout;
     end
   end

   always @ ( posedge CLK ) begin
     Dout <= Dout_next;
   end

endmodule
```



clearly combinational

clearly synch. sequential

Never do
- an always block that has **both** delayed and procedural assigns!!
- a reg assigned to by **both** delayed and procedural assigns!!

Synthesis: under the hood

Inferring Synch Sequential Logic

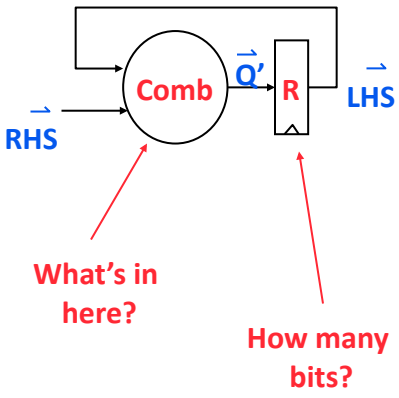
```
always @ ( posedge CLK ) begin
    LHS <= F( RHS ) ;
end
```

- Edge triggered "always" block with delayed assignments
  - LHS are reg type and cannot be assigned by another always block
  - RHS are either reg or wire type, can overlap with LHS
  - always block can contain control flow---0 or 1 assign per LHS
- A register of the appropriate width is instantiated for every LHS reg variable
- The next-state value on the next clock edge is a combinational function of the RHS variables

  Note: the RHS vars could be combinational or synchronous

Register Inference

```
reg [7:0]     count;
reg           and_bits,
              or_bits,
              xor_bits;

always @ (posedge CLK) begin
      if (reset)
          count <= 0;
      else
          count <= count + 1;

      and_bits <= &count;
      or_bits  <= |count;
      xor_bits <= ^count;
end
```



Register Inference + Retiming

```
reg [7:0]     count;
reg           and_bits,
              or_bits,
              xor_bits;

always @ (posedge CLK) begin
      if (reset)
          count <= 0;
      else
          count <= count + 1;

      and_bits <= &count;
      or_bits <= |count;
      xor_bits <= ^count;
end
```

```
reg [7:0]     count;
reg           and_bits,
               or_bits,
               xor_bits;

always @ (posedge CLK) begin
      if (reset)
         count <= 0;
      else
         count <= count + 1;
end

always @ (count) begin
      and_bits = &count;
      or_bits = |count;
      xor_bits = ^count;
end
```


Inferring Combinational Logic

- Continuous assignments: bind an expression to a wire
- Always blocks:
  - procedural assignments to reg-type LHS
  - all input signals are included in the trigger list
    ⇒ LHS recomputed whenever inputs change
  - { LHS } does not intersect { inputs }
    ⇒ no feedback cycles
  - all LHS are assigned on all possible control paths
    ⇒ LHS does not depend on itself ⇒ no storage

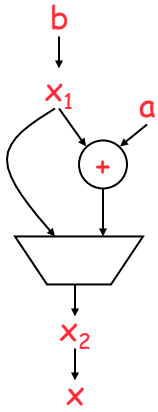
⇒ A combinational relationship between RHS and LHS

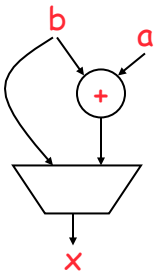
Procedural Descriptions

```
{
  x = b;
  if (y)
    x = x + a;
}
```

```
{
  x1 = b;
  if (y)
    x2 = x1 + a;
  else
    x2 = x1
  . . . .

  x = xmax
}
```

```
{
  if (y)
    x = b + a;
  else
    x = b;
}
```

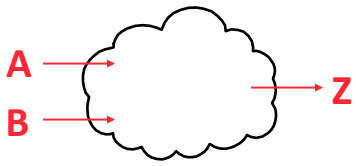




to flatten out a procedural description, convert to single-assignment

Sequentialized Description

```
output [15:0] Z;
reg [15:0]    Z;

input [15:0]  A;
input [15:0]  B;

integer    i;

always @ (A or B) begin

   Z=0;

   for(i=0;i<=15;i=i+1) begin
     if (A[i]) begin
        Z=Z+(B<<i);
     end
   end

end
```



Try writing *that* in a continuous assignment expression

Synthesizable Loops

- Static loops are allowed

  good for scanning across fixed size vectors and arrays

- A loop must be statically unrollable, i.e.,
  - loop index must be integer type
  - loop index initial value must statically resolve to a constant
  - valid loop index operations are +, -
  - the valid loop condition test must test against a static limit using relational operators: <, <=, >, >=

Some tools even support bounded-recursive functions. We do not suggest this in 18-447.

Combinational Logic

- In general
  - RTL Synthesis tools let you control the combinational logic by the structure of your expressions
  - With optimizations turned off, tools translate an expression literally and faithfully

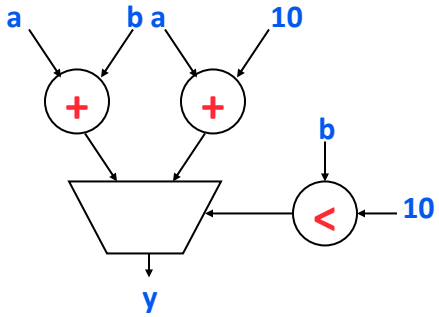
    Most optimizations are turned-off by default
  - RTL Synthesis tools can infer "custom-width" {adders, subtractors, comparators}, multipliers and multiplexers, everything else is mapped as random logic

    Simple syntax-directed translation
  - You can instantiate library modules explicitly to improve performance
  - Tool- and target-specific "pragmas" or "directives" permit fine-tuning.

Simple Example

- The statement

```
if (b < 10)
        y = a + b;
else
        y = a + 10;
```



- Maps to
  - A comparator that compares the value of `b` with `10`
  - An adder that has `a` and `b` as inputs
  - An adder that has `a` and `10` as inputs
  - A multiplexer (implied by the `if` statement) that controls the final value of `y`
- The width of the datapath depends type of `a`, `b` and `y`

Case-Switch

```
output [7:0] out;
input [2:0]  in;
reg [7:0]    out;

always @( in ) begin
    case (in)
       3'h0: out=1;
       3'h1: out=2;
       3'h2: out=4;
       3'h3: out=8;
       3'h4: out=16;
       3'h5: out=32;
       default: out=x;
    endcase
end
```

Watch out: keywords different from C/C++

Tells synthesis that everything else is **don't care!**

Verilog Compiler Directives

- Insert external file
  - `` `include "filename.v" ``
  - inserts `filename.v` at that line
- Macros
  - `` `define MACRONAME value ``
  - Replace `` `MACRONAME `` with value everywhere in code
- Macro-controlled conditionals
  - Use different code depending on which macros are defined

```
`ifdef SYNTHESIZE
`include "something_synthesizable.v"
`else
`include "something_not_synthesizable.v"
`endif
```

  - Also: `` `ifndef `` "if not defined"

Don't forget the backticks!

Default Net Type

- By default, most tools will assume that undeclared values are 1-bit wires
- Instead, you can turn this "feature" off!

Before

```
module test (input [3:0] a,
   output [3:0] b);

   wire [3:0] typoo;

   assign typo = a;
   assign b = typo;

endmodule
```

Assumes "typo" is a one-bit wire (and doesn't tell you!)

After

```
`default_nettype none

module test (input wire [3:0] a,
   output wire [3:0] b);

   wire [3:0] typoo;

   assign typo = a;
   assign b = typo;

endmodule
```

Now need to explicitly declare inputs and outputs as "wires"

Gives error: "typo" undeclared

Readings

- Highly recommended readings,
  - HDL Compiler for Verilog: Reference Manual, Synopsys Documentation
  - Guide to HDL Coding Styles for Synthesis, Synopsys Documentation