
Introduction to Design With Verilog

Course “Mantras”

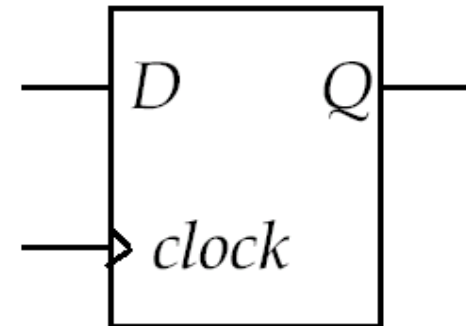
- One clock, one edge, Flip-flops only
- Design BEFORE coding
- Behavior implies function
- Clearly separate control and datapath

Purpose of HDLs

- Purpose of Hardware Description Languages:
 - Capture design in Register Transfer Language form
 - i.e. All registers specified
 - Use to simulate design so as to verify correctness
 - Pass through Synthesis tool to obtain reasonably optimal gate-level design that meets timing
 - Design productivity
 - Automatic synthesis
 - Capture design as RTL instead of schematic
 - Reduces time to create gate level design by an order of magnitude
- Synthesis
 - Basically, a Boolean Combinational Logic optimizer that is timing aware

Basic Verilog Constructs

- Flip-Flop
 - Behavior:
 - For every positive edge of the clock Q changes to become equal to D
- Write behavior as “code”
- **always@()**
 - Triggers execution of following code block
 - () called “sensitivity list”
 - Describes when execution triggered



Mantra #3

- Behavior implies function
 - Determine the behavior described by the Verilog code
 - Choose the hardware with the matching behavior

always@(posedge clock)

Q <= D;

- Code behavior:
 - Q re-evaluated every time there is a rising edge of the clock
 - Q remains unchanged between rising edges
- This behavior describes the behavior of an edge-triggered Flip-flop



Verilog example

- What is the behavior and matching logic for this code fragment?

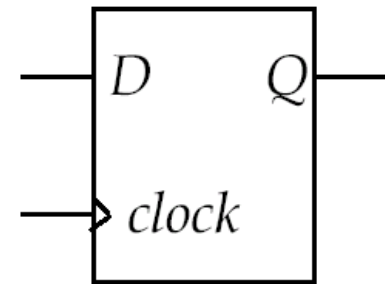
```
always@(clock or D)  
    if (clock) Q <=D;
```

- Hint : always@(foo or bar) triggers execution whenever foo or bar changes

Flip-Flops

- Every variable assigned in a block starting with

`always@ (posedge clock)` or
`always@ (negedge clock)`



becomes the output of an edge-triggered flip-flop

- This is the only way to build flip-flops

Verilog Module for Flip-flop

```
module flipflop (D, clock, Q);  
input D, clock;  
output Q;  
reg Q;  
always@(posedge clock)  
begin  
    Q <= D;  
end  
endmodule
```

Module Name
Connected Ports
Port Declarations
Local Variable
Declarations
Code Segments
endmodule

VHDL model for Flip-flop

```
entity flipflop is
    port (clock, D:in bit; Q: out bit);
end flipflop;
architecture test of flipflop is
begin process
    begin
        wait until clock' event and clock = `1';
        Q <= D;
    end process;
end test;
```

Verilog vs. VHDL

- Verilog
 - Based on C, originally Cadence proprietary, now an IEEE Standard
 - Quicker to learn, read and design in than VHDL
 - Has more tools supporting its use than VHDL

Verilog vs. VHDL

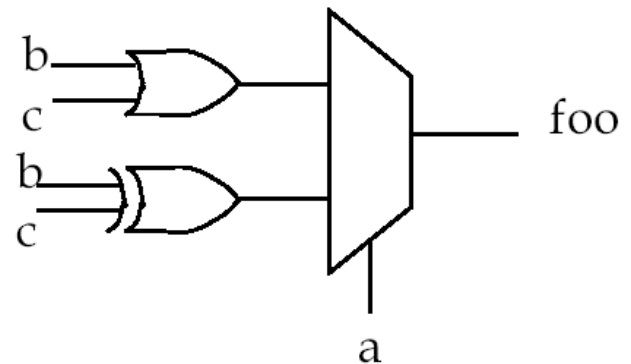
- VHDL
 - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
 - Developed by the Department of Defense, based on ADA
 - An IEEE Standard
 - More formal than Verilog, e.g. Strong typing
 - Has more features than Verilog

Verilog vs. VHDL (cont'd)

- In practice, there is little difference
 - How you design in an HDL is more important than how you code
 - Can shift from one to another in a few days

Verilog Combinational Logic

- Combinational Logic Example
- How would you describe the behavior of this function in words?



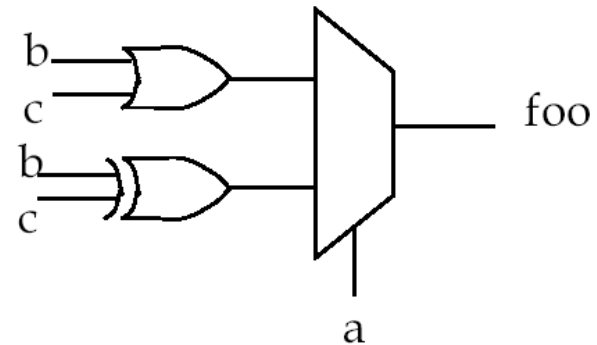
- And in Code?

Behavior → Function

always@(a or b or c)

if (a) foo = b^c;

else foo = b | c;



- All logical inputs in sensitivity list
- If; else → Multiplexor
- Behavior = whenever input changes, foo = mux of XOR or OR
- Same behavior as combinational logic

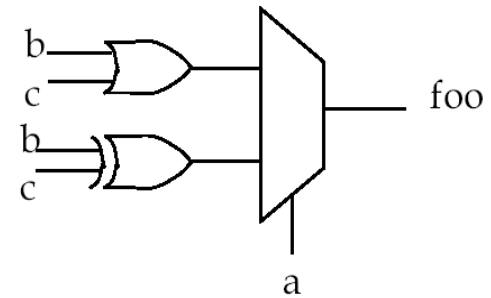
Procedural Blocks

- Statement block starting with an “always@” statement is called a procedural block
- Why?
 - Statements in block are generally executed in sequence (i.e. procedurally)

Alternative Coding Style for CL

- Verilog has a short hand way to capture combinational logic
- Called “continuous assignment”

`assign foo = a ? b^c : b | c;`



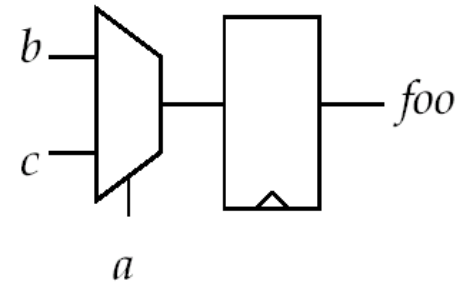
LHS re-evaluated whenever anything in RHS changes

`f = a ? d : e;` same as “if (a) f=d else f=e;

Input Logic to Flip-Flops

- Can include some combinational logic in FF procedural block

```
always@(posedge clock)
  if (a) foo <= c;
  else foo <= b;
```



- Behavior → function
 - foo is re-evaluated on every clock edge → output of FF
 - If;else → MUX

RTL Coding Styles

- That's it!
- Three coding styles
 - `always @ (???edge clock)` → FFs and input logic
 - `always @ (*)` → Combinational logic (CL)
 - `assign a =` → Continuous Assignment CL
- The hard part is NOT coding but DESIGN

Mantra #2

- The most important one for this course

ALWAYS DESIGN BEFORE CODING



- Why?
 - Must code at Register Transfer Level
 - → Registers and “transfer” (combinational) logic must be worked out before coding can start

Design Before Coding

- Automatic synthesis does NOT relieve you of logic design
- It does relieve you of:
 - Logic optimization
 - Timing calculations and control
 - In many cases, detailed logic design
- If you don't DESIGN BEFORE CODING, you are likely to end up with the following:
 - A very slow clock (long critical path)
 - Poor performance and large area
 - Non-synthesizable Verilog
 - Many HDL lint errors



Avoid Temptation!

- Temptation #1:
 - “Verilog looks like C, so I’ll write the algorithm in C and turn it into Verilog with a few always@ statements”
- Usual results:
 - Synthesis problems, unknown clock level timing, too many registers

Avoid Temptation! (cont'd)

- Temptation #2
 - “I can’t work out how to design it, so I’ll code up something that looks right and let Synthesis fix it”
- Usual result
 - Synthesis DOES NOT fix it

Avoid Temptation! (cont'd)

- Temptation #3
 - “Look at these neat coding structures available in Verilog, I’ll write more elegant code and get better results”
- Usual result of temptation #3
 - Neophytes : Synthesis problems
 - Experts: Works fine but does not usually give a smaller or faster design + makes code harder to read and maintain
- Better logic, not better code gives a better design

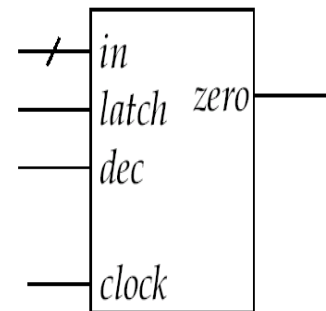
Design Before Coding

Steps in Design

1. Work out the hardware algorithm and overall strategy
2. Identify and name all the registers (flip-flops)
 - Determine system timing while doing this
3. Identify the behavior of each “cloud” of combinational logic
4. TRANSLATE design to RTL
5. Verify Design
6. Synthesize Design

Design Example: Count Down Timer

- Specification:
 - 4-bit counter
 - count value loaded from `in` on a positive clock edge when `latch` is high
 - count value decremented by 1 on a positive clock edge when `dec` is high
 - decrement stops at 0
 - `zero` flag active high whenever count value is 0



What NOT To Do

- Coding before design:

```
always@(posedge clock)
for (value=in; value>=0;value--)
  if (value==0) zero = 1
  else zero = 0;
```

- OR:

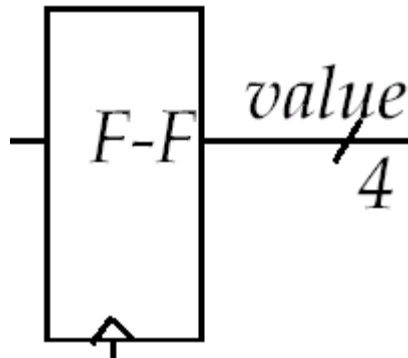
```
always@(posedge clock)
for (value=in; value>=0;value--)
  @(posedge clock)
  if (value==0) zero = 1
  else zero = 0;
```

Strategy

1. Work out the hardware algorithm and overall strategy
 - Strategy:
 - Load 'in' into a register
 - Decrement value of register while 'dec' high
 - Monitor register values to determine when zero

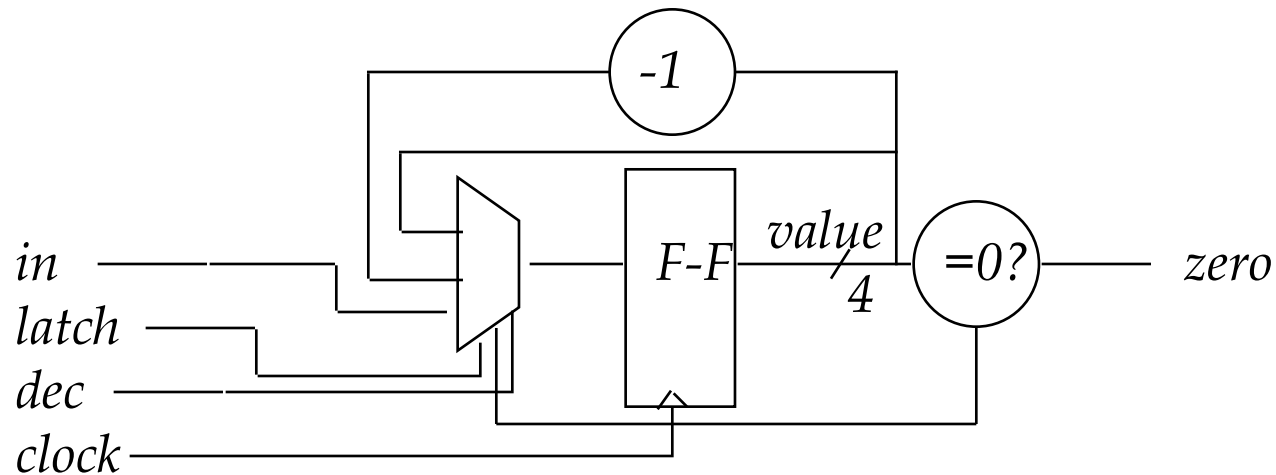
Design

2. Identify and name all the registers (flip-flops)



Design (cont'd)

3. Identify the behavior of each “cloud” of combinational logic



4. TRANSLATE design to RTL

```
module counter (clock, in, latch, dec, zero);

input      clock;      /* clock */
input [3:0] in;        /* starting count */
input      latch;     /* latch `in' when high */
input      dec;       /* decrement count when dec high */
output     zero;      /* high when count down to zero */

reg [3:0] value;      /* current count value */
wire      zero;

always@(posedge clock)
begin
    if (latch) value <= in;
    else if (dec && !zero) value <= value - 1'b1;
end

assign zero = (value == 4'b0);
endmodule /* counter */
```

Features in Verilog Code

Note that it follows the hardware design, not the 'C' specification

Multibit variables:

reg [3:0] value;
4-bit `signal' [MSB:LSB] i.e. value[3] value[2] ... value[0] **BIG ENDIAN**

Specifying constant values:

1'b1; 4'b0;
size 'base value ; size = # bits, HERE: base = binary
NOTE: zero filled to left

Procedural Block:

always@() Executes whenever variables in **sensitivity list ()**
begin change value change as indicated
 ↓
end Usually statements execute in sequence, i.e. procedurally
 begin ... end only needed if more than one statement in block

Design Example ... Verilog

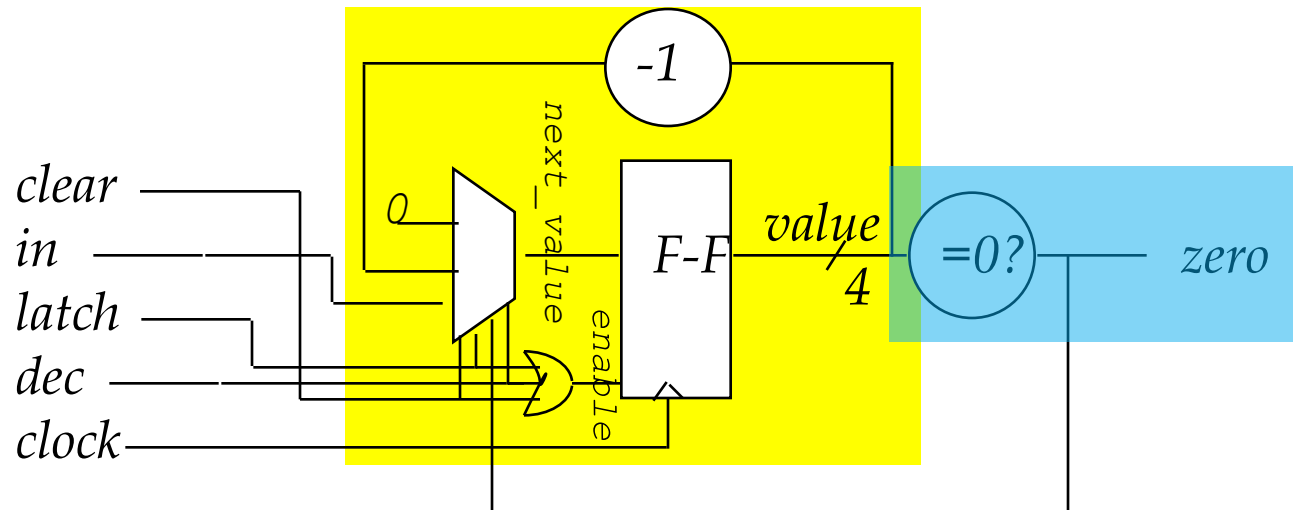
- Continuous Assignment:

`Assign` is used to implement combinational logic directly

Questions

1. When is the procedural block following the `always@ (posedge clock)` executed?
2. When is 'zero' evaluated?
3. How is a comment done?
4. What does `1'b1` mean?
5. What does `reg [3:0] value;` declare?

Behavior → Function



```
always@(posedge clock)
begin
  if (latch) value = in;
  else if (dec && !zero) value = value - 1'b1;
end
```

```
assign zero = ~|value;
```

Misc. Alternative Coding

```
module counter ( clock, in, latch, dec, zero);
  // Simple down counter with zero flag
  input  clock; /* clock */
  input [3:0] in; /* starting count */
  input  latch; /* latch `in' when high */
  input  dec; /* decrement count when dec high */
  output zero; /* high when count down to zero */

  reg [3:0] value; /* current count value */
  reg zero;
  wire [3:0] value_minus1;
  reg [3:0] mux_out;

  // Count Flip-flops with input multiplexor
  always@(posedge clock)begin
    value <= mux_out;
  end

  always @(*) begin
    if(latch) begin
      mux_out <= in;
    end
    else if(dec && !zero) begin
      mux_out <= value_minus1;
    end
    else begin
      mux_out <= value;
    end
  end
  assign value_minus1 = value - 1'b1;
  // combinational logic for zero flag
  assign zero = ~|value;
endmodule /* counter */
```

Alternative Coding

```
module counter ( clock, in, latch, dec, zero);
  // Simple down counter with zero flag
  input      clock; /* clock */
  input [3:0] in;   /* starting count */
  input      latch; /* latch `in' when high */
  input      dec;  /* decrement count when dec high */
  output     zero; /* high when count down to zero */

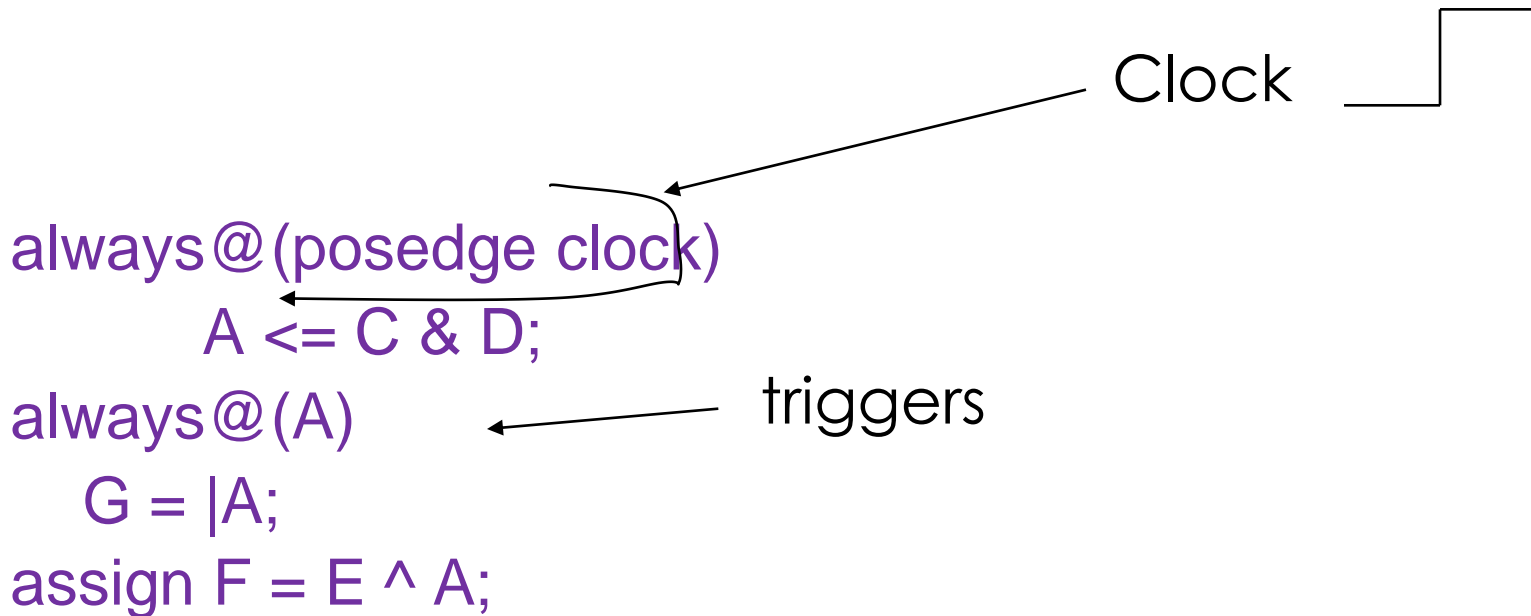
  reg [3:0]      value; /* current count value */
  reg           zero;
  // register 'value' and associated input logic
  always@(posedge clock) begin
    if (latch) value <= in;
    else if (dec && !zero) value <= value - 1'b1;
  end
  // combinational logic to produce 'zero' flag
  always@(value) begin
    if(value == 4'b0)
      zero = 1'b1;
    else
      zero = 1'b0;
  end
endmodule /* counter */
```

Verilog 2001 Version

```
module counter (input clock, input [3:0] in, input latch, input dec, output reg zero);  
  /* current count value */  
  reg [3:0] value;  
  
  always@(posedge clock) begin  
    if (latch)  
      value <= in;  
    else if (dec && !zero)  
      value <= value - 1'b1;  
  end  
  
  always@(*) begin  
    if(value == 4'b0)  
      zero = 1;  
    else  
      zero = 0;  
  end  
endmodule /* counter */
```

Intrinsic Parallelism

- How Verilog models the intrinsic parallelism of hardware



Intrinsic Parallelism

- Algorithm for
 always@(A) G = |A;
 assign F = E ^ A;

when A changes:

 In same time “step”:

 nextG = |A;

 nextF = E ^ A;

 At end of time step:

 G = nextG; F=nextF;

Review

- How do you build a flip-flop in Verilog?
- How does Verilog handle the intrinsic parallelism of hardware?
- What is a procedural block?
- What is continuous assignment?

5. Verify Design

- Achieved by designing a “test fixture” to exercise design
- Verilog in test fixture is not highly constrained
 - See more Verilog features in test fixture than in RTL

Test Fixture

```
`include "count.v" //Not needed for Modelsim simulation
module test_fixture;
    reg                clock100;
    reg                latch, dec;
    reg                [3:0] in;
    wire               zero;
    initial //following block executed only once
    begin
// below commands save waves as vcd files. These are not needed if Modelsim used as the simulator

        $dumpfile("count.vcd"); // waveforms in this file
        $dumpvars; // saves all waveforms
        clock100 = 0;
        latch = 0;
        dec = 0;
        in = 4'b0010;
        #16 latch = 1; // wait 16 ns
        #10 latch = 0; // wait 10 ns
        #10 dec = 1;
        #100 $finish; //finished with simulation
    end
    always #5 clock100 = ~clock100; // 10ns clock

// instantiate modules -- call this counter u1
    counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec), .zero(zero));
endmodule /*test_fixture*/
```

Simple Test Fixture (cont'd)

```
always #5 clock100 = ~clock100; // 10ns clock
```

```
counter u1(.clock(clock100), .in(in), .latch(latch), .dec(dec),  
.zero(zero));
```

```
endmodule /*test_fixture*/
```

Features

- 'zero' is type wire because its an output of the module instance u1

Features in test fixture

``include "count.v"`

- Includes DUT design file

`initial`

- Procedural Block
- Executed ONCE on simulation startup
- Not synthesizable

`#16`

- Wait 16 units (here ns – defined by 'timescale command)

`$dumpfile ; $finish`

- Verilog commands

Features in test fixture (cont'd)

```
counter u1(.clock(clock100), .in(in), .latch(latch),  
.dec(dec), .zero(zero));
```

- Builds one instance (called u1) of the module 'counter' in the test fixture

```
.clock(clock100)
```

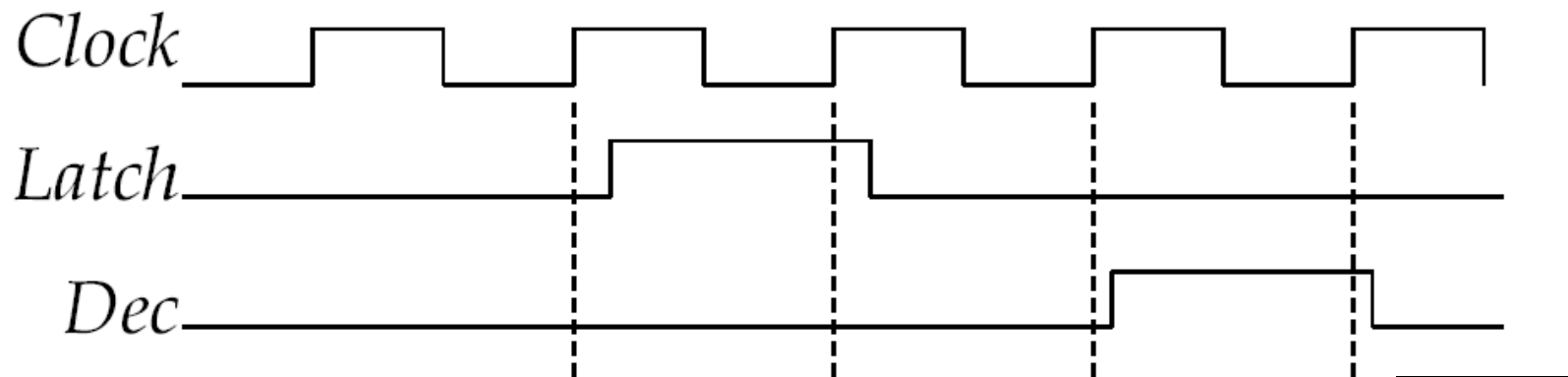
- Variable clock100 in test fixture connected to port clock in counter module

Features in test fixture (cont'd)

always #5 clock = ~clock;

- Inverts clock every 5 ns

Waveforms:



Verilog 2001 Test Fixture

```
`include "count.v"
module test_fixture;
    reg                clock100 = 0 ;
    reg                latch = 0;
    reg                dec = 0;
    reg                [3:0] in = 4'b0010;
    wire zero;
    initial //following block executed only once
    begin
        $dumpfile("count.vcd"); // waveforms in this file..
                                   // Note Comments from previous example
        $dumpvars; // saves all waveforms
        #16 latch = 1;                // wait 16 ns
        #10 latch = 0;                // wait 10 ns
        #10 dec = 1;
        #100 $finish;                //finished with simulation
    end

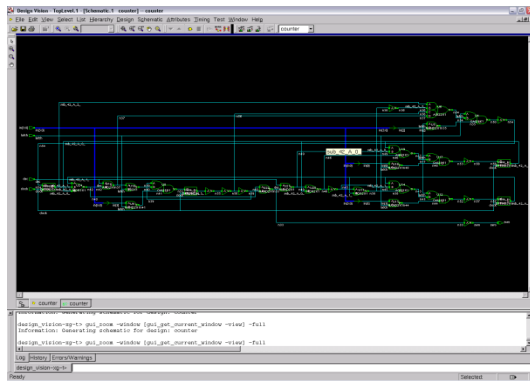
    always #5 clock100 = ~clock100; // 10ns clock

    // instantiate modules -- call this counter u1
    counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec), .zero(zero));
endmodule /*test_fixture*/
```

Synthesis

Step 5. After verifying correctness, the design can be synthesized to optimized logic with the Synopsys tool

Synthesis Script run in Synopsys (test_fixture is NOT synthesized):(See attached script file). The result is a gate level design (netlist):



```
INVX1 U7 ( .A(n38), .Y(n36) );  
OAI21X1 U8 ( .A(n39), .B(n40), .C(n41), .Y(n51) );  
NAND2X1 U9 ( .A(in[3]), .B(latch), .Y(n41) );  
OR2X1 U10 ( .A(n37), .B(latch), .Y(n40) );  
AND2X1 U11 ( .A(dec), .B(n42), .Y(n37) );
```


‘n39’, etc. are nets, i.e. wires that connect the gates together.

Synthesis Script

```
# setup name of the clock in your design.  
set clkname clock  
  
# set variable "modname" to the name of topmost module in design  
set modname counter  
  
# set variable "RTL_DIR" to the HDL directory w.r.t synthesis directory  
set RTL_DIR ./  
  
# set variable "type" to a name that distinguishes this synthesis run  
set type lecture  
  
#set the number of digits to be used for delay result display  
set report_default_significant_digits 4  
  
#-----  
# Read in Verilog file and map (synthesize)  
# onto a generic library.  
# MAKE SURE THAT YOU CORRECT ALL WARNINGS THAT APPEAR  
# during the execution of the read command are fixed  
# or understood to have no impact.  
# ALSO CHECK your latch/flip-flop list for unintended  
# latches  
#-----  
read_verilog $RTL_DIR/counter.v
```

Set all the different variables required for a given design synthesis run

Always stop at this point and look at reports generated



Synthesis Script

```
#-----  
# Our first Optimization 'compile' is intended to produce a design  
# that will meet hold-time  
# under worst-case conditions:  
#   - slowest process corner  
#   - highest operating temperature and lowest Vcc  
#   - expected worst case clock skew  
#-----  
# Set the current design to the top level instance name  
# to make sure that you are working on the right design  
# at the time of constraint setting and compilation  
#-----  
current_design $modname  
#-----  
# Set the synthetic library variable to enable use of DesignWare blocks  
#-----  
set synthetic_library [list dw_foundation.sldb]  
#-----  
# Specify the worst case (slowest) libraries and slowest temperature/Vcc  
# conditions. This would involve setting up the slow library as the target  
# and setting the link library to the concatenation of the target and the  
# synthetic library
```

Set current design for analysis

Point to DesignWare library for compilation

Use worst case delays to focus on setup timing

You can change the clock period but not uncertainty

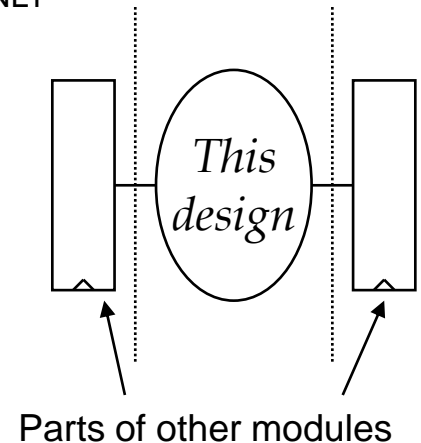
Synthesis Script

```
#-----  
set target_library osu018_stdcells_slow.db  
set link_library [concat $target_library $synthetic_library]  
#-----  
# Specify a 5000ps clock period with 50% duty cycle and a skew of 300ps  
#-----  
set CLK_PER 5  
set CLK_SKEW 0.3  
create_clock -name $clkname -period $CLK_PER -waveform "0 [expr $CLK_PER / 2]" $clkname  
set_clock_uncertainty $CLK_SKEW $clkname  
#-----  
# Now set up the 'CONSTRAINTS' on the design:  
# 1. How much of the clock period is lost in the modules connected to it?  
# 2. What type of cells are driving the inputs?  
# 3. What type of cells and how many (fanout) must it be able to drive?  
#-----  
# ASSUME being driven by a slowest D-flip-flop. The DFF cell has a clock-Q  
# delay of 353 ps. Allow another 100 ps for wiring delay at the input to design  
# NOTE: THESE ARE INITIAL ASSUMPTIONS ONLY  
#-----  
set DFF_CKQ 0.353  
set IP_DELAY [expr 0.1 + $DFF_CKQ]  
set_input_delay $IP_DELAY -clock $clkname [remove_from_collection [all_inputs] $clkname]
```

Logic must be connected to something else, which will affect its timing. Here we specify what the synthesizer design is connected to, and its timing.

Synthesis Script

```
#-----  
# ASSUME this module is driving a D-flip-flop. The DFF cell has a set-up time of 919 ps  
# Allow another 100 ps for wiring delay. NOTE: THESE ARE INITIAL ASSUMPTIONS ONLY  
#-----  
set DFF_SETUP 0.919  
set OP_DELAY [expr 0.1 + $DFF_SETUP]  
set_output_delay $OP_DELAY -clock $clkname [all_outputs]  
#-----  
# ASSUME being driven by a D-flip-flop  
#-----  
set DR_CELL_NAME DFFPOSX1  
set DR_CELL_PIN Q  
set_driving_cell -lib_cell "$DR_CELL_NAME" -pin "$DR_CELL_PIN"  
[remove_from_collection [all_inputs] $clkname]  
#-----  
# ASSUME the worst case output load is 4 D-FFs (D-inputs) and 0.2 units of wiring capacitance  
#-----  
set PORT_LOAD_CELL osu018_stdcells_slow/DFFPOSX1/D  
set WIRE_LOAD_EST 0.2  
set FANOUT 4  
set PORT_LOAD [expr $WIRE_LOAD_EST + $FANOUT * [load_of $PORT_LOAD_CELL]]  
set_load $PORT_LOAD [all_outputs]
```



Synthesis Script

```
#-----  
# Now set the GOALS for the compile. In most cases you want minimum area, so set the  
# goal for maximum area to be 0  
#-----  
set_max_area 0  
#-----  
# This command prevents feedthroughs from input to output and avoids assign statements  
#-----  
set_fix_multiple_port_nets -all [get_designs]  
#-----  
# During the initial map (synthesis), Synopsys might have built parts (such as adders)  
# using its DesignWare(TM) library. In order to remap the design to our TSMC025 library  
# AND to create scope for logic reduction, I want to 'flatten out' the DesignWare  
# components. i.e. Make one flat design 'replace_synthetic' is the cleanest way of  
# doing this  
#-----  
replace_synthetic  
#-----  
# check_design checks for consistency of design and issues # warnings and errors. An  
# error would imply the design is not compilable. Do "man check_design" for more info.  
#-----
```

This leads up to the first “compile” which does the actual logic optimization.

Synthesis Script

```
check_design
```

```
#-----
```

```
# link performs check for presence of the design components instantiated within the design.  
# It makes sure that all the components (either library unit or other designs within the  
# hierarchy) are present in the search path and connects all of the disparate components  
# logically to the present design. Do "man link" or more information.
```

```
#-----
```

```
link
```

```
#-----
```

```
# Now resynthesize the design to meet constraints, and try to best achieve the goal, and  
# using the CMOSX parts. In large designs, compile can take a long time!  
# -map_effort specifies how much optimization effort there is, i.e. low, medium, or high.  
# Use high to squeeze out those last picoseconds.  
# -verify_effort specifies how much effort to spend making sure that the input and output  
# designs are equivalent logically. This argument is generally avoided.
```

```
#-----
```

```
compile -map_effort medium
```

```
#-----
```

```
# Now trace the critical (slowest) path and see if  
# the timing works.  
# If the slack is NOT met, you HAVE A PROBLEM and  
# need to redesign or try some other minimization  
# tricks that Synopsys can do
```

```
#-----
```

We need to run checks to make sure errors (both in design and in setup) are absent before we compile.

Compile can take a while to run on a large (or poor) design.

Synthesis Script

```
report_timing > timing_max_slow_${type}.rpt
#report_timing -delay min -nworst 30 > timing_report_min_slow_30.rpt
#report_timing -delay max -nworst 30 > timing_report_max_slow_30.rpt
```

Always look at this report.

You can ask for timing of the next slowest paths as well (see commented code). This can be used to decide if you want to try retiming and analyzing other paths as well. Run `man report_timing` to see other useful options like `–from` `–to` `–through`

```
#-----
# This is your section to do different things to
# improve timing or area - RTFM (Read The Manual) :)
#-----
# Specify the fastest process corner and lowest temp and highest (fastest) Vcc
#-----
set target_library osu018_stdcells_fast.db
set link_library osu018_stdcells_slow.db
translate
#-----
# Since we have a 'new' library, we need to do this again
#-----
```

Synthesis Script

```
#-----  
# Set the design rule to 'fix hold time violations'  
# Then compile the design again, telling Synopsys to  
# only change the design if there are hold time  
# violations.  
#-----  
set_fix_hold clock compile -only_design_rule -incremental  
#-----  
# Report the fastest path. Make sure the hold  
# is actually met.  
#-----  
report_timing > timing_max_fast_${type}.rpt  
report_timing -delay min > timing_min_fast_holdcheck_${type}.rpt  
#-----  
# Write out the 'fastest' (minimum) timing file  
# in Standard Delay Format. We might use this in  
# later verification.  
#-----  
write_sdf counter_min.sdf  
#-----  
# Since Synopsys has to insert logic to meet hold violations, we might find that we have setup  
# violations now. So lets recheck with the slowest corner, etc.  
# YOU have problems if the slack is NOT MET. 'translate' means 'translate to new library'  
#-----
```

Use best case delays
to focus on hold timing

Use compile `-incremental`
after first compile

Synthesis Script

```
set target_library osu018_stdcells_slow.db
set link_library osu018_stdcells_slow.db
translate
report_timing > timing_max_slow_holdfixed_${type}.rpt
report_timing -delay min > timing_min_slow_holdfixed_${type}.rpt
#-----
# Sanity checks
#-----
set target_library osu018_stdcells_fast.db
set link_library osu018_stdcells_fast.db
translate
report_timing > timing_max_fast_holdfixed_${type}.rpt
report_timing -delay min > timing_min_fast_holdfixed_${type}.rpt
#-----
# Write out area distribution for the final design
#-----
report_cell > cell_report_final.rpt
#-----
# Write out the resulting netlist in Verilog format for use
# by other tools in Encounter for Place and Route of the design
#-----
change_names -rules verilog -hierarchy > fixed_names_init
write -hierarchy -f verilog -o counter_final.v
#-----
# Write out the 'slowest' (maximum) timing file in Standard
# Delay Format. We could use this in later verification.
#-----
write_sdf counter_max.sdf
```

Though it happens rarely, the extra logic inserted to fix hold problems, might have affected the critical path.

Here we check for that by re-doing the maximum delay analysis for the slowest process corner

Write out final netlist, area distribution reports and timing information in sdf format

Detail of Design PostSynthesis

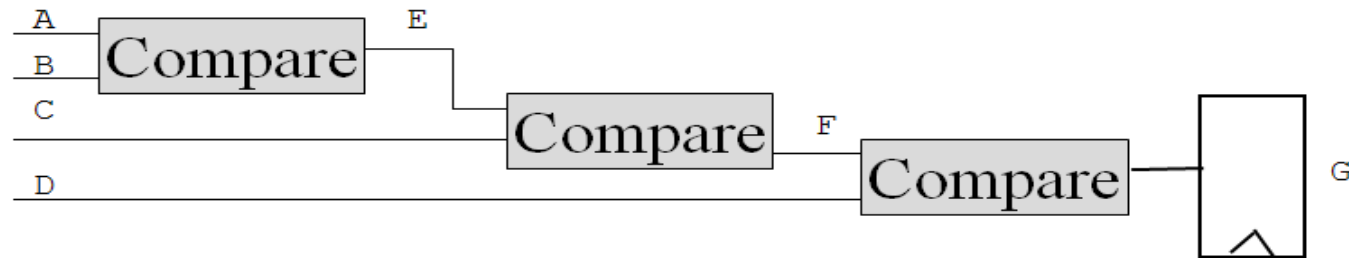
```
module counter ( clock, in, latch, dec, zero );  
input [3:0] in;  
input clock, latch, dec;  
output zero;  
wire sub_42_A_0_, sub_42_A_1_, sub_42_A_2_, sub_42_A_3_, n33, n34, n35,  
      n36, n37, n38, n39, n40, n41, n42, n43, n44, n45, n46, n47, n48, n49,  
      n50, n51, n52, n53, n54, n55, n56, n57, n58;  
  
DFFPOSX1 value_reg_0_ ( .D(n58), .CLK(clock), .Q(sub_42_A_0_) );  
DFFPOSX1 value_reg_1_ ( .D(n57), .CLK(clock), .Q(sub_42_A_1_) );  
DFFPOSX1 value_reg_3_ ( .D(n51), .CLK(clock), .Q(sub_42_A_3_) );  
DFFPOSX1 value_reg_2_ ( .D(n54), .CLK(clock), .Q(sub_42_A_2_) );  
INVX1 U3 ( .A(n33), .Y(zero) );  
OAI21X1 U4 ( .A(latch), .B(n34), .C(n35), .Y(n50) );  
NAND2X1 U5 ( .A(latch), .B(in[2]), .Y(n35) );  
AOI22X1 U6 ( .A(sub_42_A_2_), .B(n36), .C(n56), .D(n37), .Y(n34) );  
INVX1 U7 ( .A(n38), .Y(n36) );  
OAI21X1 U8 ( .A(n39), .B(n40), .C(n41), .Y(n51) );
```

Detail of Design PostSynthesis

```
NAND2X1 U9 ( .A(in[3]), .B(latch), .Y(n41) );
OR2X1 U10 ( .A(n37), .B(latch), .Y(n40) );
AND2X1 U11 ( .A(dec), .B(n42), .Y(n37) );
OAI21X1 U12 ( .A(latch), .B(n43), .C(n44), .Y(n52) );
NAND2X1 U13 ( .A(in[1]), .B(latch), .Y(n44) );
AOI21X1 U14 ( .A(sub_42_A_1_), .B(n45), .C(n38), .Y(n43) );
NOR2X1 U15 ( .A(n45), .B(sub_42_A_1_), .Y(n38) );
INVX1 U16 ( .A(n46), .Y(n45) );
OAI21X1 U17 ( .A(latch), .B(n47), .C(n48), .Y(n53) );
NAND2X1 U18 ( .A(in[0]), .B(latch), .Y(n48) );
AOI21X1 U19 ( .A(sub_42_A_0_), .B(n49), .C(n46), .Y(n47) );
NOR2X1 U20 ( .A(n49), .B(sub_42_A_0_), .Y(n46) );
NAND2X1 U21 ( .A(dec), .B(n33), .Y(n49) );
NAND2X1 U22 ( .A(n42), .B(n39), .Y(n33) );
INVX1 U23 ( .A(n56), .Y(n39) );
NOR3X1 U24 ( .A(sub_42_A_1_), .B(sub_42_A_2_), .C(sub_42_A_0_), .Y(n42) );
BUFX4 U25 ( .A(n50), .Y(n54) );
INVX8 U26 ( .A(sub_42_A_3_), .Y(n55) );
INVX1 U27 ( .A(n55), .Y(n56) );
BUFX2 U28 ( .A(n52), .Y(n57) );
BUFX2 U29 ( .A(n53), .Y(n58) );
endmodule
```

Exercise: Three Timing Examples (from Timing Notes)

- What do these look like in Verilog?



```
always@(A or B or C)
```

```
begin
```

```
  if (A>B) then E = A; else E = B;
```

```
  if (C>E) then F = E; else F = C;
```

```
end
```

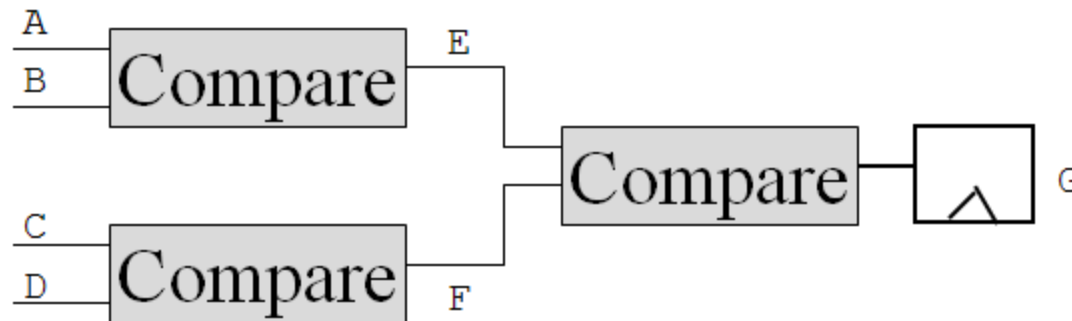
```
always@(posedge clock)
```

```
  if (D>F) then G <= D; else G <= F;
```

Why not move E, F assignments down to here?

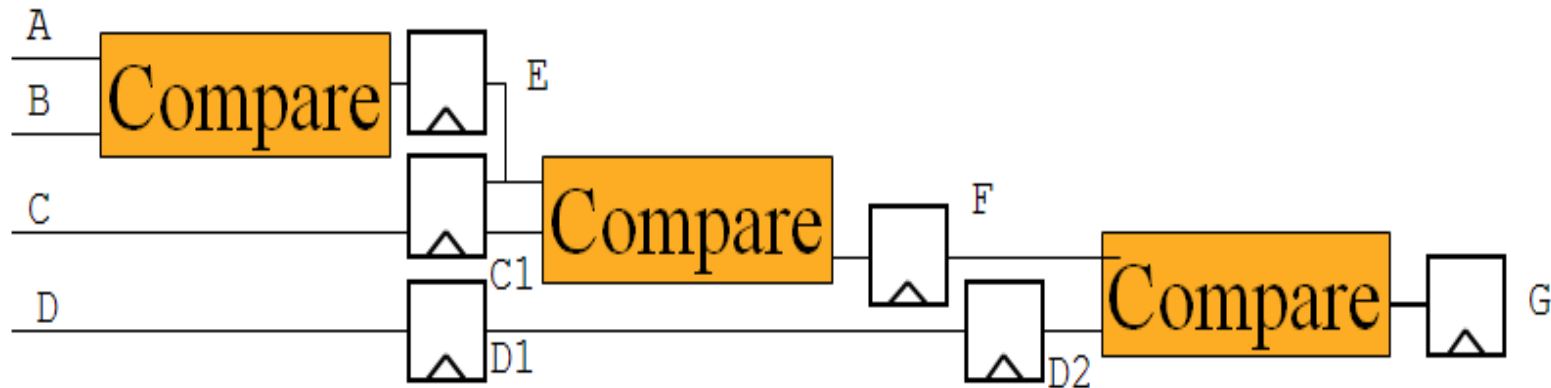
... Three timing examples

- Produce a Verilog code fragment for ...
 - Use continuous assignment



... Three Timing Examples

- And for this...



Note: Outputs of all flip-flops have to be named

Sample Problem

- Accumulator:
 - Design an 8-bit adder accumulator with the following properties:
 - While 'accumulate' is high, adds the input, 'in1' to the current accumulated total and add the result to the contents of register with output 'accum_out'.
 - use absolute (not 2's complement) numbers
 - When 'clear' is high ('accumulate' will be low) clear the contents of the register with output 'accum_out'
 - The 'overflow' flag is high is the adder overflows

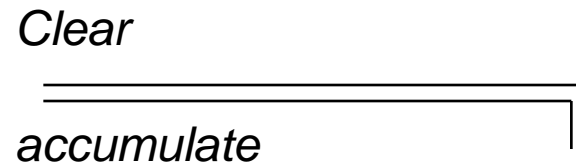
Hint:

8-bit adder produces a 9-bit result:

$$\{\text{carry_out, sum}\} = A+B;$$

Sketch Design

1. Determine and name registers.
2. Determine combinational logic



Summary

- What are our two “mantras” used here?
- What is built for all assignments after always@(posedge clock)?
- What is built after always@(A or B)
- What is built with assign C =

Summary

- In Synthesis with Synopsys
 - What is important after the “read” statement?
 - Which timing library do we use for the first compile?
 - What does “compile” do?
 - What is important to do after every incremental compile?