laalcham@ucsc.edu for CSE 125, UCSC

Verilog Synthesis

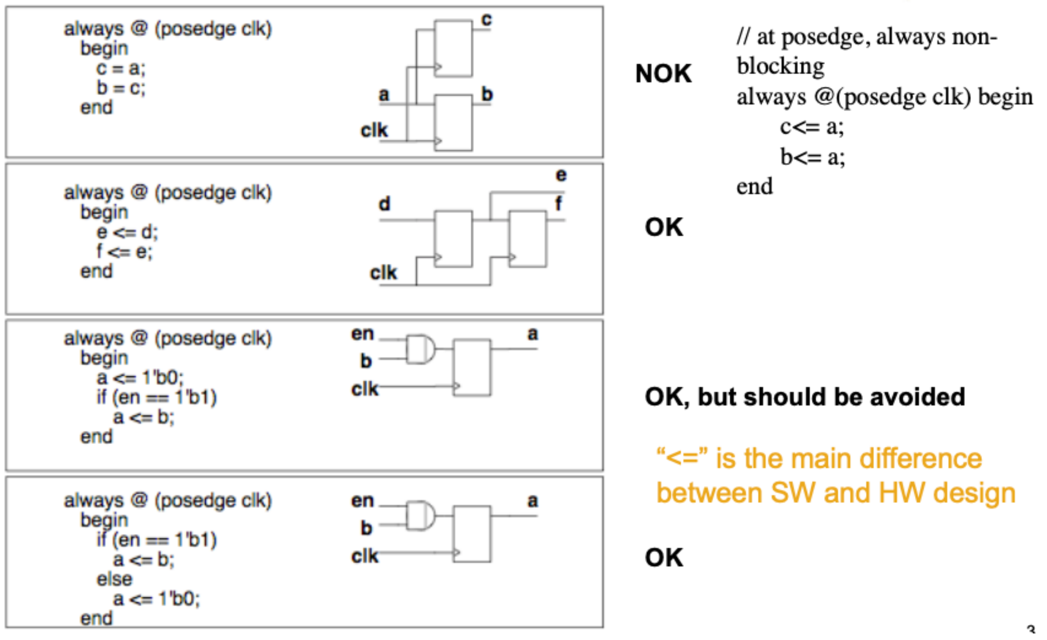
Recall: only subset of Verilog is synthesizable.

Blocking Versus Non-Blocking assignments in synchronous blocks

The choice affects both simulation and hardware synthesis.

blocking: evaluated one after another

non-blocking: evaluated simultaneously



```
always @ (posedge clk)
  begin
    c = a;
    b = c;
  end
```

NOK

```
// at posedge, always non-blocking
always @(posedge clk) begin
    c<= a;
    b<= a;
end
```

```
always @ (posedge clk)
  begin
    e <= d;
    f <= e;
  end
```

OK

```
always @ (posedge clk)
  begin
    a <= 1'b0;
    if (en == 1'b1)
      a <= b;
  end
```

OK, but should be avoided

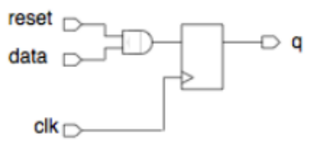
"<=" is the main difference between SW and HW design

```
always @ (posedge clk)
  begin
    if (en == 1'b1)
      a <= b;
    else
      a <= 1'b0;
  end
```

OK

D-flop w/ Synchronous Reset

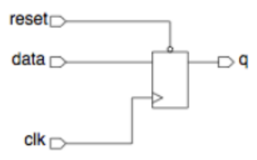
- assume that reset is active low. res_n another common name



```
always @(posedge clk) begin
   if (!res_n)
      q <= 0;
   else
      q <= data;
end
```

D-flop w/ Asynchronous Reset

- assume that reset is active low. res_n another common name
- most basic FPGA libraries provide asynchronous FFs which have additional reset signals. they are preferable



```
always @(posedge clk or negedge res_n) begin
   if (!res_n)
      q <= 0;
   else
      q <= data;
end
```

If/else in Verilog

- must have a begin/end keyword set if it surrounds multiple lines
- must be in an always or initial block

[If/else example: Flip-Flop, reset used as asynchronous signal]

```
always @(posedge clk or negedge res_n) begin
   if (!res_n)
   begin
      q1 <= 0;
      q2 <= 0;
   end
   else
   begin
      q1 <= data1;
      q2 <= data1;
   end
end
```

Multiplexer: Implemented with if/else

- multiplexors are a combinatorial logic unit, so they are not driven by a clock
- multiplexors are synthesized into custom FPGA/ASIC hardware
- multiplexors use blocking assignments because their output is continuously updated
- most efficiently implemented with transmission gates

```
always @(*) begin
    if (sel)
        z = in1;
    else
        z = in0;
end
```

Larger Multiplexers

- larger multiplexors are used to "decode" select signals in the if statement
- sometimes hierarchies of multiplexers are used b/c there is hardware for 2-1, 4-1, 8-1
  - synthesis tools will come up with the minimal logic to implement large if/else statements as a mux

[Example: if/else synthesized into a 4-1 mux]

```
always @(*) begin
   if (sel==2'b00)
      z = in0;
   else if (sel==2'b01)
      z = in1;
   else if (sel==2'b10)
      z = in2;
   else
      z = in3;
end
```

[Example: if/else synthesized into ?]

- may synthesize into not a mux.

```
always @(*) begin
   if (sel==2'b00)
      z = in0;
   else if (sel==2'b01)
      z = in1;
   else
      z = in2;
end
```

For larger muxes, a case statement looks better!

Multiplexer: implemented with Case

- often synthesized into hardware just like the if/else
- default case is a useful way to handle unknown cases

[Example: equivalent Case and If/Else statements]

the default case body cannot be synthesized, so it would be ignored in hardware implementation

```
wire [15:0] in0, in1, in2, in3;
wire [1:0] sel;
always @(*) begin
       case ( sel )
         2'b00:   Y = in0;
         2'b01:   Y = in1;
         2'b10:   Y = in2;
         2'b11:   Y = in3;
         default: Y = 16'hxxxx;
       endcase
end
```

```
always @(*) begin
   if (sel==2'b00)
      z = in0;
   else if (sel==2'b01)
      z = in1;
   else if (sel==2'b10)
      z = in2;
   else if (sel==2'b11)
      z = in3;
   else
      z = 1'bx;
end
```

Design pattern:
Always have a else/default case to catch bugs. Assign x'es

Casex in Verilog

- casex statements use the equivalent of === to do comparisons (z and x are well-defined inputs)
  - casex treats x, z, or ? as don't cares
  - casez treats z or ? as don't cares

[Example]

default case would have been sufficient for the last case listed.

```
always @ (*)
casex (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  2'bxx,2'bx0,2'bx1,2'b0x,2'b1x,
  2'bzz,2'bz0,2'bz1,2'b0z,2'b1z : $display("Error in SEL");
endcase
```

For Loop in Verilog: not what it seems

In software, for loops are used as a dynamic construct to perform an action multiple times. **In hardware design, for loops generate multiple components that are defined within the body.**

- dynamic enabling/disabling of hardware is not possible
- wires can be used to interconnect the repeated hardware
- to synthesize, the loop boundary must be a compile time constant

[Example: For Loop]

- generates 4 multiplexers with an XOR going into each, 1 for each bit of b and c

```
integer i;

for(i=0 ; i <4 ; i++ ) begin
   if(d[i])
      a[i] = b[i] ^ c[i];
   else
      a[i] = 0;
end
```

i needs to be a compile time constant

is this synthesizable?

Local Parameters in Verilog

- local parameters are something from the old Verilog-95 style
- used for inputs
- increase flexibility of modules

[Example: local parameters]

- A_size can be changed during instantiation, but B_size cannot

```
module first_module
#(parameter A_size = 4)(A, B);
 localparam B_size = 1<<A_size;
 input [A_size-1:0]A;
 input [B_size-1:0]B;
 output C;
 ...
endmodule
```

Compiler Directives

- used to let the compiler know what to do

```
`include "file_name" // defines, paremeters definitions

`ifdef macro_name
     // verilog_source_code
`else
     /* verilog_source_code */
`endif

`define VAL 1'b1
`define NAND(dval) nand #(dval)

`NAND(3) i1 (y,a,`VAL);
`NAND(5) i2 (o,c,d);
```

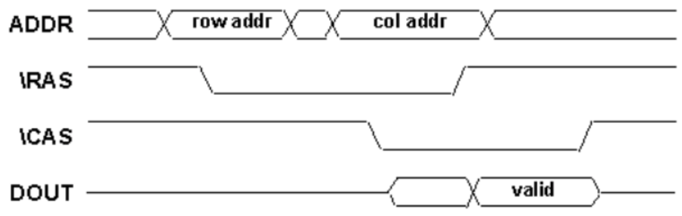
Interfaces vs. Protocols

- Verilog modules define interface but not protocol on how to use them ☹
    - this is a weakness of Verilog!
- C/C++ has similar issues (lack of protocol) but protocol is more critical in HDLs
- Challenge: how should we use the interface?
    - Solution: Module interface + written specification + timing diagram

[Example: DRAM]
Written Specification for the Module:

1. each access starts with a RAS
2. wait for ras2cas
3. assert CAS

- the only thing to clearly describe how the module works is to see the timing diagram.
- protocol missin!



Hardware versus Software Design

- hardware design is always about performance: if you don't need performance, write software. x)
- a significant part of RTL design is about performance tuning

Goal of Hardware Design:

- optimize throughput (operations/second)
- optimize Fmax (clock frequency)

2 options to Optimize Hardware:	
Play with the tools (synthesis constraints, place and route P&R)	Modify hardware design (more powerful technique)
• constrain the design to a target frequency • synthesis will try to meet timing • synthesis will also optimize for area/power **constrain tighter to push timing.**	**pipelining:** split combinatorial logic blocks and insert FFs in between them so that the design can be clocked higher • 2 cycles but 2x max frequency means double throughput

Many hardware designs have 100s-1,000s of pipeline stages.

Quick Frequency Clock

- if design does not meet your targets, you need to redesigns

Chicken-egg problem: no design, no timing. No timing, no design
Solution: design/implement the pipeline stage that seems the most frequency constrained, or or the pipeline stage that "if changed" would significantly modify other sections designs.

→ get a sense of what hardware can go into an x ns propagation delay.
get a feeling for computational complexity of particular operators.

Synthesis

Synthesis can take a loooong time for larger designs.
Problem:
You do not know the frequency you can clock at until you synthesize.

Solution:
synthesize frequently. Learn to love the warnings that tools give – it will simplify your testing! You should have 0 warnings, or understand the reason for a warning.

Clock Signal (P&R)

Recall there are 3 types of signals in hardware design:

1. data
2. clock
3. reset

clock and reset are driven across the whole chip to drive each and every flip flop.

when doing FPGAs

- specify the specialty of clk and reset
- drive clock signal from a BUFG (buff global clock)
- any synchronous block must be driven by clock buffer BUFG

Driving the clock signal from a BUFG is needed to minimize skew

- clock signal is fanned out and distributed, leading to clock skew
- low skew is desirable because of setup/hold time of FFs.
  - design has to be clocked at rate that accommodates clock skew and logic latency
  - positive and negative skew possible
  - low skew helps phase match off-chip synchronous signals.
  - some signals from previous FFs might clock inappropriately
  - high skew means clock period must be increased

Generating a Clock for Test Benches

Clock signals can be generated in an initial block in testbenches. not synthesizable. Below are some examples of how to do for a clock period of 10 time units.

```
reg clk;
```

```
// method 1
initial begin
clk= 0;
end
always
  #5 clk <= ~clk;
```

```
// method 2
always
begin
#5 clk = 0;
#5 clk = 1;
end
```

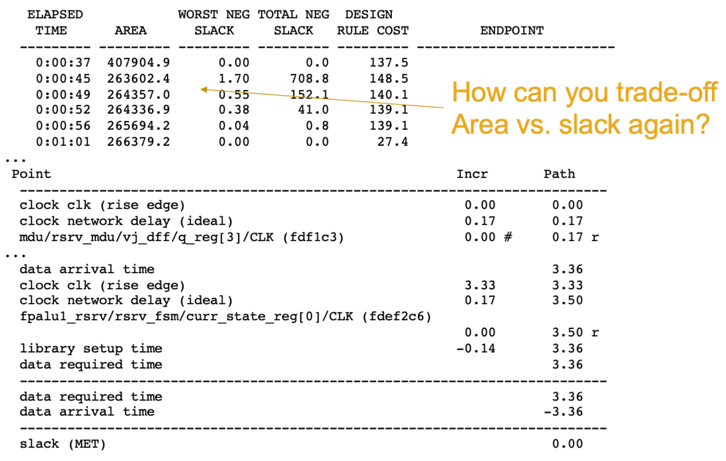
```
// method 3
initial begin
clk = 0;
forever begin
#5 clk = ~clk;
end
end
```

Reading Timing Reports

Timing Report: Synopsys

Pictured belong is a timing report from Synopsys, a big company that built tools people use. Synthesis tools will make multiple designs and list the following for each:

area
worst negative slack
total negative slack
design rule cost



```
 ELAPSED              WORST NEG TOTAL NEG  DESIGN
  TIME      AREA        SLACK     SLACK   RULE COST         ENDPOINT
--------- --------- --------- --------- --------- -------------------------
  0:00:37  407904.9      0.00       0.0     137.5
  0:00:45  263602.4      1.70     708.8     148.5
  0:00:49  264357.0      0.55     152.1     140.1
  0:00:52  264336.9      0.38      41.0     139.1
  0:00:56  265694.2      0.04       0.8     139.1
  0:01:01  266379.2      0.00       0.0      27.4
...
 Point                                                   Incr       Path
  -----------------------------------------------------------------------
  clock clk (rise edge)                                   0.00       0.00
  clock network delay (ideal)                             0.17       0.17
  mdu/rsrv_mdu/vj_dff/q_reg[3]/CLK (fdf1c3)               0.00 #     0.17 r
...
  data arrival time                                                  3.36
  clock clk (rise edge)                                   3.33       3.33
  clock network delay (ideal)                             0.17       3.50
  fpalu1_rsrv/rsrv_fsm/curr_state_reg[0]/CLK (fdef2c6)
                                                          0.00       3.50 r
  library setup time                                     -0.14       3.36
  data required time                                                 3.36
  -----------------------------------------------------------------------
  data required time                                                 3.36
  data arrival time                                                 -3.36
  -----------------------------------------------------------------------
  slack (MET)                                                        0.00
```

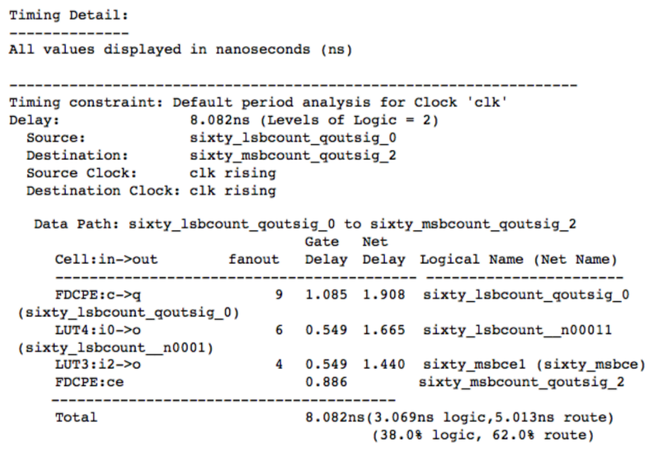
slack

- slack == clock period – propagation delay
- 0 slack is good. worst negative slack is positive and means design could be clocked faster.

How can you trade-off area and slack?

Timing Report: Xilinx

Xilinx is another company that has synthesis tools that will have a report. Reports most critical paths in design, source, clock, etc.

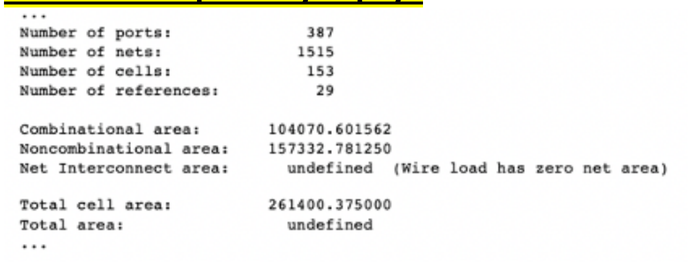


```
Timing Detail:
--------------
All values displayed in nanoseconds (ns)

-----------------------------------------------------------------
Timing constraint: Default period analysis for Clock 'clk'
Delay:               8.082ns (Levels of Logic = 2)
  Source:            sixty_lsbcount_qoutsig_0
  Destination:       sixty_msbcount_qoutsig_2
  Source Clock:      clk rising
  Destination Clock: clk rising

   Data Path: sixty_lsbcount_qoutsig_0 to sixty_msbcount_qoutsig_2
                                   Gate   Net
     Cell:in->out        fanout   Delay  Delay  Logical Name (Net Name)
     ----------------------------------------  -----------------------
     FDCPE:c->q               9   1.085  1.908  sixty_lsbcount_qoutsig_0
 (sixty_lsbcount_qoutsig_0)
     LUT4:i0->o               6   0.549  1.665  sixty_lsbcount__n00011
 (sixty_lsbcount__n0001)
     LUT3:i2->o               4   0.549  1.440  sixty_msbce1 (sixty_msbce)
     FDCPE:ce                     0.886        sixty_msbcount_qoutsig_2
    ----------------------------------------
     Total                        8.082ns(3.069ns logic,5.013ns route)
                                         (38.0% logic, 62.0% route)
```

Resource Report: Synopsys

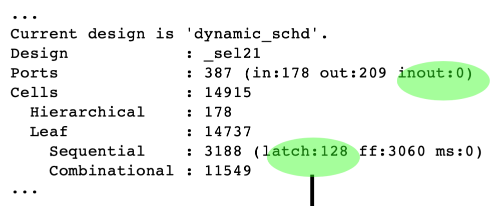


```
 ...
 Number of ports:                387
 Number of nets:                1515
 Number of cells:                153
 Number of references:            29

 Combinational area:       104070.601562
 Noncombinational area:    157332.781250
 Net Interconnect area:      undefined  (Wire load has zero net area)

 Total cell area:          261400.375000
 Total area:                 undefined
 ...
```

Other Statistics: Synopsys



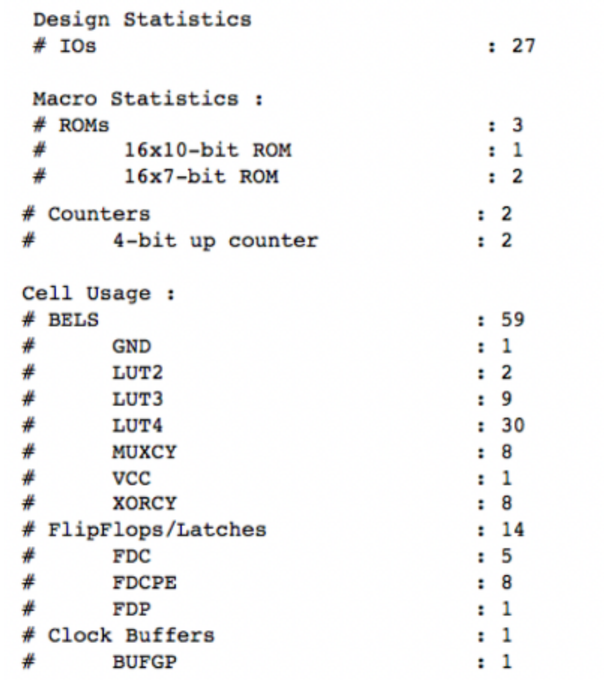
```
...
Current design is 'dynamic_schd'.
Design             : _sel21
Ports              : 387 (in:178 out:209 inout:0)
Cells              : 14915
  Hierarchical     : 178
  Leaf             : 14737
    Sequential     : 3188 (latch:128 ff:3060 ms:0)
    Combinational  : 11549
...
```

class projects should not have latches

Why?

Latches probably mean you forgot to flop a signal
E.g. Include all the branches of an if or case statement.

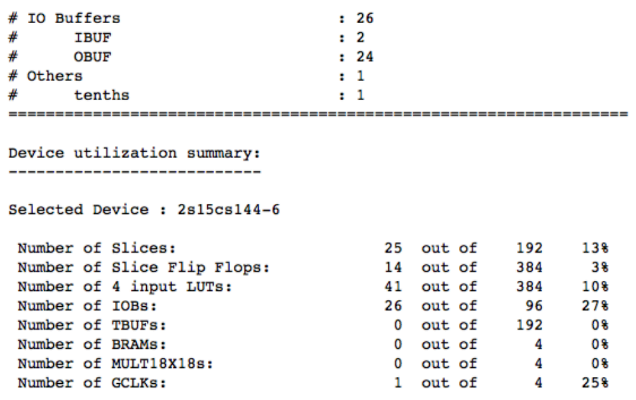
Resource Report: Xilinx



```
  Design Statistics
  # IOs                              : 27

  Macro Statistics :
  # ROMs                             : 3
  #      16x10-bit ROM               : 1
  #      16x7-bit ROM                : 2
 # Counters                          : 2
 #      4-bit up counter             : 2

 Cell Usage :
 # BELS                              : 59
 #      GND                          : 1
 #      LUT2                         : 2
 #      LUT3                         : 9
 #      LUT4                         : 30
 #      MUXCY                        : 8
 #      VCC                          : 1
 #      XORCY                        : 8
 # FlipFlops/Latches                 : 14
 #      FDC                          : 5
 #      FDCPE                        : 8
 #      FDP                          : 1
 # Clock Buffers                     : 1
 #      BUFGP                        : 1
```



```
# IO Buffers                         : 26
#      IBUF                          : 2
#      OBUF                          : 24
# Others                             : 1
#      tenths                        : 1
=================================================================

Device utilization summary:
---------------------------

Selected Device : 2s15cs144-6

 Number of Slices:                      25  out of    192    13%
 Number of Slice Flip Flops:            14  out of    384     3%
 Number of 4 input LUTs:                41  out of    384    10%
 Number of IOBs:                        26  out of     96    27%
 Number of TBUFs:                        0  out of    192     0%
 Number of BRAMs:                        0  out of      4     0%
 Number of MULT18X18s:                   0  out of      4     0%
 Number of GCLKs:                        1  out of      4    25%
```