


EN1640

Design of Computing Systems Spring 2015

Lecture 8, 9, & 10: FPGA Dataflow and Verilog Modeling

February 9, 11, 13, 2015
Prof. R. Iris Bahar




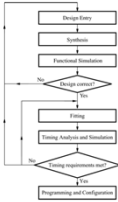
© 2015 R.I. Bahar
Portions of these slides taken from Professors S. Reda

Lab #1

- Lab #1 is posted on the webpage
www.brown.edu/Departments/Engineering/Courses/engn1640
- Note for problem #3,
 - I want the result displayed in *decimal* format.
 - Use 2's complement to represent negative numbers.
- If you are new to the Quartus II system, please be sure to start early so you have time to go through the tutorials.
- Lab assignment must be demo'ed by Friday, Feb. 13 @ 6pm

Topics

1. Programmable logic
2. Design Flow
3. Verilog --- A Hardware Description Language

```

module fulladder(output reg[3:0] sum,
                 output reg c_out,
                 input [3:0] a, b,
                 input c_in);
  ...
endmodule
  
```

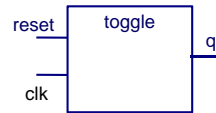
3

3. Introduction to Verilog

- Hardware description language (HDL) vs. software languages:
 - Concurrency
 - Propagation of time
 - Signal dependency or sensitivity
- Verilog:
 - Case sensitive, with syntax similar to C
 - Comments designated by // to end of line or by /* to */ across several lines
 - Textbooks:
 - [Introduction to Logic Synthesis using Verilog HDL](#)
 - [Verilog Quickstart](#)
 - [Verilog Digital System Design](#)
 - [The Verilog Hardware Description Language](#)

4

Verilog modules

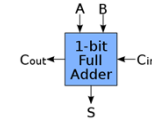


- The functionality of each module can be defined with 3 modeling levels:
 - Structural (or gate level)
 - Dataflow level
 - Behavioral (or algorithmic level)
- Verilog allows different levels of abstraction to be mixed in the same module.

5

Modules and ports

```
module FA(A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
...
endmodule
```



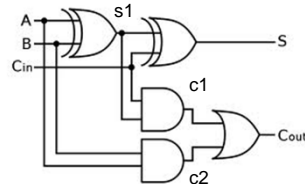
- All port declarations (input, output, inout) are implicitly declared as wire.
- If an input should hold its value, it must be declared as reg.

```
module FA(input A, input B, input Cin, output S, output Cout);
...
endmodule
```

6

Verilog Part 1. Structural modeling

- Data types: bits**
- Nets represent connections between hardware elements.
- Continuously driven by the output of connected devices.
- Nets are declared using the keyword wire.



- wire s1;
- wire c1, c2;
- wire d=0;

7

Gate level modeling

```
wire Z, Z1, OUT, OUT1, OUT2, IN1, IN2;

and a1(OUT1, IN1, IN2);
nand na1(OUT2, IN1, IN2);
xor x1(OUT, OUT1, OUT2);
not (Z, OUT);
buf final (Z1, Z);
```

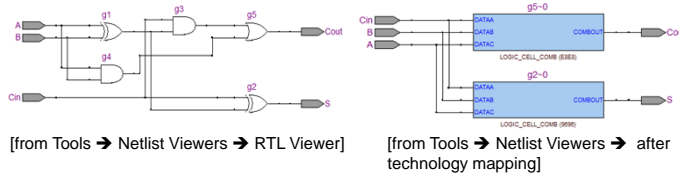
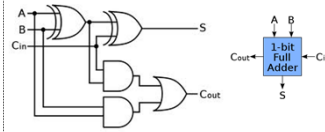
- Describes the topology of a circuit
- All instances are executed concurrently just as in hardware
- Instance name is not necessary
- The first terminal in the list is an output; others are inputs
- Not the most interesting modeling technique for this class

8

Example: 1-bit full adder

```
module FA(A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
wire s1, c1, c2;

xor g1(s1, A, B);
xor g2(S, s1, Cin);
and g3(c1, s1, Cin);
and g4(c2, A, B);
or g5(Cout, c1, c2);
endmodule
```



9

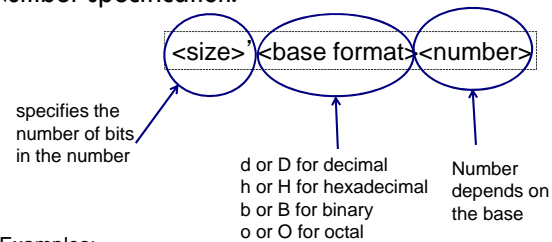
Data types: vectors

- A net or register can be declared as vectors.
- Examples:
 - wire a;
 - wire [7:0] bus;
 - wire [31:0] busA, busB, busC;
- It is possible to access bits or parts of vectors
- Examples:
 - busA[7]
 - bus[2:0]
 - virt_addr[0:2]

10

Specifying values for wires and variables

- Number specification:



Examples:

- 4'b1111
- 12'habc
- 16'd235
- 12'h13x
- 6'd3
- 12'b1111_0000_1010

X or x: don't care
Z or z: high impedance
_: used for readability

11

Instantiating an array of gates

```
wire [7:0] OUT, IN1, IN2;

// array of gates instantiations
nand n_gate [7:0] (OUT, IN1, IN2);

// which is equivalent to the following
nand n_gate0 (OUT[0], IN1[0], IN2[0]);
nand n_gate1 (OUT[1], IN1[1], IN2[1]);
nand n_gate2 (OUT[2], IN1[2], IN2[2]);
nand n_gate3 (OUT[3], IN1[3], IN2[3]);
nand n_gate4 (OUT[4], IN1[4], IN2[4]);
nand n_gate5 (OUT[5], IN1[5], IN2[5]);
nand n_gate6 (OUT[6], IN1[6], IN2[6]);
nand n_gate7 (OUT[7], IN1[7], IN2[7]);
```

12

Modules with input / output vectors

```
module fulladd4(output [3:0] sum,
               output c_out,
               input [3:0] a, b,
               input c_in);
...
endmodule
```

OR

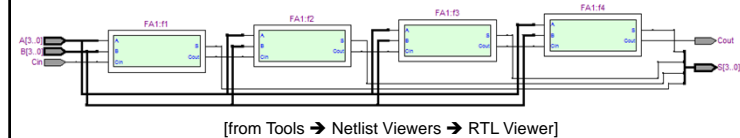
```
module fulladd4(sum, c_out, a, b, input c_in);
Output [3:0] sum;
Output c_out;
input [3:0] a, b;
input c_in;
...
endmodule
```

13

Module instantiation

```
module fulladd4(A, B, Cin, S, Cout);
input [3:0] A, B;
input Cin;
output [3:0] S;
output Cout;
wire C1, C2, C3;

FA f1(A[0], B[0], Cin, S[0], C1);
FA f2(A[1], B[1], C1, S[1], C2);
FA f3(A[2], B[2], C2, S[2], C3);
FA f4(A[3], B[3], C3, S[3], Cout);
endmodule
```



14

Alternative form of module instantiation

```
module FA(A, B, Cin, S, Cout);
input A, B, Cin;
output S, Cout;
...
endmodule
```

wire a1, a2, a3, a4, a5

```
FA f1(a1, a2, a3, a4, a5);
```

OR

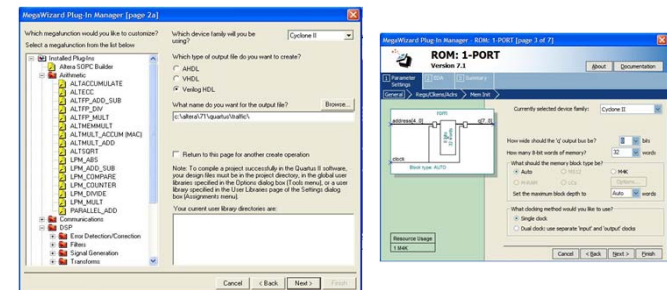
```
FA1 f1(.Cout(a5), .S(a4), .B(a2), .A(a1), .Cin(a3));
```

15

Quartus II builtin modules (megafuncions)

- Use megafuncions instead of coding your own logic to save time.
- Megafuncions include the library of parameterized modules (LPM) and Altera device-specific megafuncions

➔ Use when possible!



16

Verilog Part 2. Dataflow modeling

- Module designed by specifying data flow: Designer is aware of how data flows between registers and how it is processed in the design
- The continuous assignment is one of the main constructs used in dataflow modeling
 - assign out = i1 & i2;
 - assign addr[15:0] = addr1[15:0] ^ addr2[15:0];
 - assign {c_out, sum[3:0]}=a[3:0]+b[3:0]+c_in;
- A continuous assignment is always active and evaluated as soon as one of its right-hand-side variables change
- Assign statements describe hardware that operates concurrently
→ ordering does not matter
- Left-hand side must be scalar or vector net. Right-hand side operands can be wires, (registers, integers, and real)

17

Operator types in dataflow expressions

- Operators are similar to C except that there are no ++ or -- conventions allowed.
 - Arithmetic:** *, /, +, -, % and **
 - Logical:** !, && and ||
 - Relational:** >, <, >= and <=
 - Equality:** ==, !=, === and !==
 - Bitwise:** ~, &, |, ^ and ^~
 - Reduction:** &, ~&, |, ~|, ^ and ^~
 - Shift:** << and >>
 - Concatenation:** { }
 - Replication:** { { } }
 - Conditional:** ? :

18

Examples: 2x1 MUX and 4x1 MUX

```

module mux2to1(s, a, b, y);
output y;
input s, a, b;
assign y = (b & s) | (a & ~s);
// OR THIS WAY
assign y = s ? b : a;
endmodule

module mux4to1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
output s1, s0;
assign out = (~s1 & ~s0 & i0) |
             (~s1 & s0 & i1) |
             (s1 & ~s0 & i2) |
             (s1 & s0 & i3);
// OR THIS WAY
assign out = s1 ? (s0 ? i3:i2) : (s0 ? i1:i0);
endmodule
  
```

19

HLL vs. Verilog assignments

(a) assignment statement
ordering does matter in an HLL

```

a = 1; b = 0; s = 0;
na = 0; nb = 0;

y = na | nb;
nb = b & s;
na = a & ~s;
y = na | nb;
  
```

Final y value is 0. Final y value is 1.

(b) assign statement
ordering does not matter in Verilog

```

wire na, nb;
assign y = na | nb;
assign nb = b & s;
assign na = a & ~s;
  
```

[Example from Thornton & Reese]

20

HLL vs. Verilog assignment

(a) assignment statements in an HLL can target the same variable

```

a = 1; b = 0; s = 0;
na = 0; nb = 0;
na = b & s;
na = a & ~s;

```

The **na** variable is assigned twice; the final value of **na** is the last assignment.

(b) illegal use of **assign** statements

```

wire na;
assign na = b & s;
assign na = a & ~s;

```

Gate outputs are shorted together!
can only work with tri-state drivers

[Example from Thornton & Reese]

21

Example of a dataflow 4-bit adder

(a) Four-bit adder with no carry-in or carry-out

```

//4-bit adder
// no carry-in, carry-out
module add4bit ( a, b, s );

input [3:0] a,b;
output [3:0] s;

assign s = a + b;

endmodule

```

(b) Four-bit adder with carry-in, carry-out

```

//4-bit adder with carry-in, carry-out
module add4bit (ci, a, b, s, co);

input ci;
input [3:0] a,b;
output [3:0] s;
output co;

wire [4:0] y;

//do 5-bit sum so that we
// have access to carry out
assign y = (1'b0,a) + (1'b0,b) + (4'b0,ci);
assign s = y[3:0]; //four-bit output
assign co = y[4]; //carry-out

endmodule

```

{ } is the concatenation operator

[Example from Thornton & Reese]

22

Simulation using Quartus waveform editor

- Integrated with Quartus tool for design simulation and verification
- Enables you to create waveforms easily (in binary, decimal, or hexadecimal)
- Tutorial available on class webpage
- You can also use Mentor Graphics Model for simulation (but it is not as intuitive)

23

Verilog Part 3. Behavioral modeling

- Design is expressed in algorithmic level, which frees designers from thinking in terms of logic gates or data flow.
- All algorithmic or procedural statements in Verilog can appear only inside two statements: **always** and **initial**.
- Each **always** and **initial** statement represents a separate activity flow in Verilog. Remember that activity flows in Verilog run in parallel.
- You can have multiple **initial** and **always** statements but you can't nest them.

```

.
reg a, b, c;

initial a=1'b0;
.
.
always @*
begin
    b = a ^ 1'b1;
    c = a + b;
end
.
.

```

24

Data types: reg, parameter

- **reg**: Verilog variable type (does not necessarily imply a physical register). Think of it as a variable or place holder. Unsigned by default
 - reg clock;
 - reg[0:40] virt_addr;
- **Register arrays or memories**: Used to model register files, RAMs, ROMs. Modeled in Verilog as a 1-dimensional array of registers.
 - reg mem1bit[0:1023];
 - reg[7:0] membyte[0:1023];
 - To access an element in an memory array: membyte[511];
- **parameters**: Define constants and cannot be used as variables.
 - parameter port_id=5;

25

Data types

- **Integers**: (signed and real): They are of type **reg**.


```
real delta;
integer i;
initial
begin
    delta = 4e10;
    i=4;
end
```
- **Arrays of integers and reals**:


```
integer count[0:7];
Integer matrix[4:0][0:255];
```
- **Strings**: can be stored in **reg**. The width of the register variables must be large enough to hold the string.


```
reg [8*19:1] string_value;
initial
    string_value = "Hello Verilog World";
```

26

initial statements

- An **initial** block start at time 0, executes exactly once and then never again.
- If there are multiple **initial** blocks, each blocks starts to execute concurrently at time 0 and each blocks finish execution independently of the others.
- Multiple behavioral statements must be grouped using **begin** and **end**. If there is one statement then grouping is not necessary.

In procedural statements (*initial*, *always*) LHS must be of type registers (and its derivatives)

```
reg x, y, m;
initial m=1'b0;

initial
begin
    x = 1'b0;
    y = 1'b1;
end
```

27

always statements

- The **always** statement starts at time 0 and executes the statements in the always block when the events in its sensitivity list occur
- Powerful constructs like if, if-else, case, and looping are only allowed inside always blocks.
- **always** statements can be used to implement both combinational or sequential logic
- Multiple behavioral statements must be grouped using **begin** and **end**.
- Multiple **always** statement can appear in a module

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y;

//use boolean ops
always @(a or b or s)
begin
    y = (b & s) | (a & ~s);
end

endmodule
```

28

Sensitivity list of events

- An event is the change in the value on a register or a net. Events can be utilized to trigger the execution of a statement of a block of statements.
- The @ symbol is used to specify an event control.
- For combinational logic, any net that appears on the right side of an "=" operator in the **always** block should be included in the event list.
- [For sequential logic] Statements executed on changes in signal value or at a positive (**posedge**) or negative (**negedge**) transition of the signal.

```

module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y, na, nb;

//use intermediates
//and implicit event
//list
always @*
begin
    nb = b & s;
    na = a & ~s;
    y = na | nb;
end
endmodule

```

29

always statements

- Any net assigned within an **always** block must be declared **reg**;
 - does not imply that net is driven by a register or sequential logic.
- "=" operator in an **always** block is called a *blocking assignment*
- A latch is inferred when there is a logic path through the **always** block that does not assign a value to the output
- Synthesis tool assumes blocking assignments are evaluated sequentially.

→ The order assignments are written in **always** blocks determines how logic is synthesized.

(a) Incorrect, produces an inferred latch as no assignment is made to q if ld is '0'

```

always @(ld or d)
begin
    if (ld) q = d;
end

```

(b) Correct, produces combinational logic

```

always @(ld or d or q_old)
begin
    q = q_old;
    if (ld) q = d;
end

```

30

always statements

- Because of the sequential nature of an **always** block, the same net can be assigned multiple times in an **always** block;
- The last assignment takes precedence.

(a) **clr** takes precedence over **ld** if both are '1'

```

always @(ld or clr or d or q_old)
begin
    q = q_old;
    if (ld) q = d;
    if (clr) q = 0;
end

```

ld	clr	q	q_old
0	0	0	0
0	1	0	0
1	0	d	0
1	1	0	0

(logic is not minimal)

(b) **ld** takes precedence over **clr** if both are '1'

```

always @(ld or clr or d or q_old)
begin
    q = q_old;
    if (clr) q = 0;
    if (ld) q = d;
end

```

ld	clr	q	q_old
0	0	0	0
0	1	0	0
1	0	d	0
1	1	d	0

[Example from Thornton & Reese]

31

Conditional statements

- Very similar to C
- Can also appear inside **always** and **initial** blocks

```

if(x)
begin
    y = 1'b1;
    z = 1'b0;
end

```

conditional expression

```

if (count < 10)
    count = count+1;
else
    count = 0;

```

```

if(alu_control == 0)
    y = x + z;
else if (alu_control == 1)
    y = x - z;
else if (alu_control == 2)
    y = x * z;
else
    y = x;

```

```

reg [1:0] alu_control;
..
case (alu_control)
    2'd0 : y = x + z;
    2'd1 : y = x - z;
    2'd2 : y = x * z;
    default: y=x;
endcase

```

32

Example: Mux4x1

```

module mux4x1(out, i0, i1, i2, i3, s1, s0);
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
    case({s1, s0})
        2'd0: out = i0;
        2'd1: out = i1;
        2'd2: out = i2;
        2'd3: out = i3;
    endcase
endmodule

```

33

Level sensitive latch (D-Latch)

- The D-latch: an **always** block that makes a non-blocking assignment (" \leq ") of d to q when the q input is non-zero
- When input g is 0, the **always** block does not make any assignment to q
 - What happens when $g \neq 0$?
 - Latch is inferred on q
- Use non-blocking assignments (" \leq ") as opposed to blocking assignments (" $=$ ") in **always** blocks intended as sequential logic

(a) level-sensitive storage element (D-latch)

```

always @(g or d)
begin
    if (g) q <= d;
end

```

latch is transparent to changes on d

q follows d when g is high

[from Thornton & Reese]

34

Edge-triggered storage element (D-FF)

- The **@** symbol is used to specify event control.
- Statements can be executed on changes in signal value or at a positive (posedge) or negative (negedge) transition
- In general, edge-triggered storage elements are preferred to level-sensitive because of simpler timing requirements
- The 1-bit edge-triggered FF provided by FPGA vendors are DFF because of simplicity and speed.

(b) edge-triggered storage element (data flip-flop, or DFF)

```

always @(posedge clk)
begin
    q <= d;
end

```

capture the d input

q follows d on rising edge of clk

[Thornton & Reese]

35

Example from problem 3 from Lab #1

```

module quest3(CLOCK_50, LEDR);
input CLOCK_50;
output reg [17:0] LEDR;
integer count;
always @(posedge CLOCK_50)
begin
    if(count == 50000000)
    begin
        LEDR[0] <= !LEDR[0];
        count <= 0;
    end
    else
        count <= count + 1;
    end
endmodule

```

[from Tools -> Netlist Viewers -> RTL Viewer]

36

DFF chains

(a) All of these Verilog code fragments synthesize to the same chain of DFFs

```

reg qa, qb, qc;
always @(posedge clk)
begin
  qa <= a;
  qb <= qa;
  qc <= qb;
end

```

```

reg qa, qb, qc;
always @(posedge clk)
begin
  qc <= qb;
  qb <= qa;
  qa <= a;
end

```

```

reg qa, qb, qc;
always @(posedge clk) qa <= a;
always @(posedge clk) qb <= qa;
always @(posedge clk) qc <= qb;
end

```

(b) a wire

```

reg qa, qb, qc;
//synthesizes to a wire
always @*
begin
  qa = a;
  qb = qa;
  qc = qb;
end

```

[Thornton & Reese]

- Each non-blocking assignment synthesizes to a single DFF where its input is the output of another non-blocking assignment
- The ordering of these non-blocking assignments within an always block does not matter.

37

Blocking vs. non-blocking statements

(a) Blocking assignments - RHS values applied to LHS immediately

```

always @(posedge clk)
begin
  q1 = d;
  q2 = q1;
end

```

blocking assignments

(b) Non-blocking assignments - all RHS values applied to LHS after always b

```

always @(posedge clk)
begin
  q1 <= d;
  q2 <= q1;
end

```

non-blocking assignments

- Zero-delay blocking assignments:** assignment from right-hand-side (RHS) to left-hand-side (LHS) is completed w.o. any intervening Verilog
 - The assignment blocks the execution of the other Verilog code
- Non-blocking assignments within always block:** all RHS expressions are evaluated, and only assigned to LHS targets after the always block completes.

38

Loops in Verilog

```

for (count = 0; count < 128; count = count + 1)
begin
  .
  .
end

```

```

count = 0;
while (count < 128)
begin
  .
  .
  count = count + 1;
end

```

```

count = 0;
repeat(128)
begin
  .
  .
  count = count + 1;
end

```

Must contain a number or a signal value; only evaluated once at the beginning

- It is sometimes easier to use counts and if-then statements to create loops

39

Loop synthesis

```

module loop(CLOCK_50, A, out);
input CLOCK_50;
input [15:0] A;
output reg [15:0] out;

reg [15:0] r;
reg [4:0] count;

initial out = 16'd0;

always @(posedge CLOCK_50)
begin
  r <= A;
  for(count = 0; count <= 16'd15; count = count+1)
  begin
    if (count % 2 == 0) out[count] <= r[count];
    else out[count] <= ~r[count];
  end
end
endmodule

```

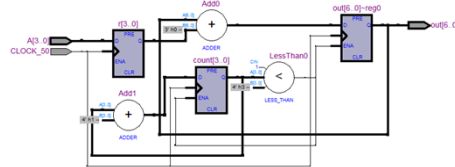
40

Loop synthesis with counts & if-then

```

module loop(CLOCK_50, A, out);
input CLOCK_50;
input [3:0] A;
output reg [6:0] out;
reg [3:0] r, count, B;
initial
begin
    out = 4'b0;
    count = 4'd0;
end
always @(posedge CLOCK_50)
begin
    always @(posedge CLOCK_50)
    begin
        if (count <= 3)
        begin
            out <= out + r;
            count <= count + 1;
        end
    end
end
endmodule

```



41

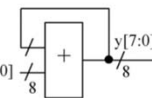
Avoid combinational loops

✗ (a) A combinational loop

```

always @*
begin
    y = y + a;
end

```



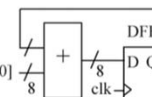
Output oscillates; period is dependent upon adder delay

✓ (b) Sequential element in feedback path

```

always @(posedge clk)
begin
    y <= y + a;
end

```



Output can only change on the active clock edge

[Thornton & Reese]

42

Guidelines (1)

- Combinational logic:
 - Use continuous assign statements to model simple combinational logic
 - Use always @(*) and blocking assignments (=) to model more complicated combinational logic
 - If an always block for combinational logic contains logic pathways due to if-else branching or other logic constructs, assign every output a default value at the beginning of the block.
 - Ensures that all outputs are assigned a value regardless of path taken through the logic, avoiding inferred latches on outputs.

[Thornton/ Reese & Harris]

43

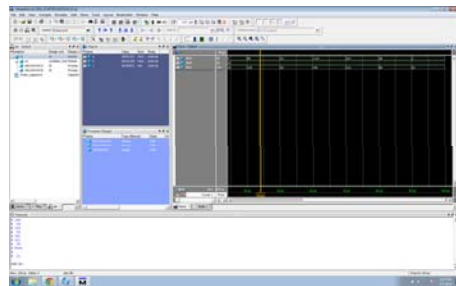
Guidelines to avoid frustration (2)

- Sequential logic:
 - Use non-blocking assignments (<=) in always blocks that are meant to represent sequential logic
 - Use posedge sensitivity to ensure DFF
- Do not make assignments to the same signal in more than one **always** statement or continuous **assign** statement
- Avoid mixing blocking and non-blocking assignments in the same **always** block.

[Thornton/ Reese & Harris]

44

Simulation using testbenches in ModelSim



```

module tb;
  reg [7:0] a;
  reg [7:0] b;
  wire [7:0] c;

  add_module add1(a, b, c);
  initial
  begin
    #5 b = 20;
    #10 b = 50;
    $monitor("%d", c);
  end

  always
    #10 a=$random;

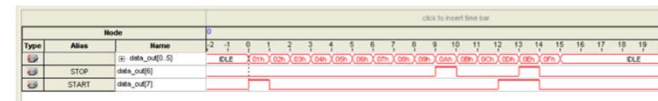
  always
    #10 if(c %2 == 0)
      $display("even\n");
endmodule

```

- Testbenches are just scripts for simulation, but are not synthesizable
- New Verilog commands enable precise timing simulation and monitoring of outputs.
- Need to learn to work with it for Mentor Graphics ModelSim
 - Optional for this class

45

SignalTap II for in-system debugging



- Enable debugging FPGA
- Insert probes and additional HW to capture internal signals and relay them over JTAG USB in the form of waveform displays
- Very valuable for identifying bugs after implementation
 - See tutorial on course website
- Make sure to disable/remove signalTap after you are done with debugging.

46