"Gateway" lab exercises

1. HelloWorld - Making a new project with a *module* and controlling a single LED with a button.
2. Hello lots of Worlds - making a *bus* to wire all switches to all LEDs; the UCF (User Constraints File).
3. HelloWorldSynchronous - using *registers* and *wires*, simulation with a verilog *test fixture*; the *sensitivity list* in always@ in simulation. "If within a module you have a signal that is on the left hand side of an assignment within an 'always@(...)' statement, then it needs to be defined as a register ('reg')".
4. ShiftingTheWorld - synthesizing a shift register with *fd* D-FlipFlops using gate level and behavioral level design; *register transfer level (RTL)* design; *module instantiation*; signal*concatenation*; introduction to *generate*.
5. ShiftingManyWorlds - 2d array of shift registers (memory); simulation exercise.
6. CountingWorlds - simple arithmetic, *multiplexing*

"Gateway" lab exercises

Getting to know the hardware and development environment

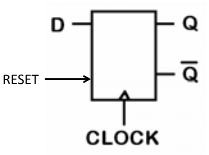
1. HelloWorld - Making a new project with a ***module*** and controlling a single LED with a button.
2. Hello lots of Worlds - making a ***bus*** to wire all switches to all LEDs; the UCF (User Constraints File).

Very important first experience with synchronous logic

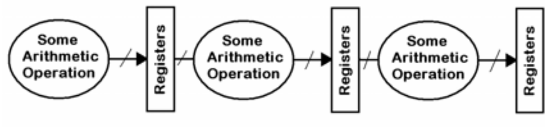
3. HelloWorldSynchronous - using ***registers*** and ***wires***, simulation with a verilog ***test fixture***; the ***sensitivity list*** in ***always@*** in simulation.

3. Hello Synchronous World

Synchronous logic – D flipflop



On the positive CLOCK edge, D is copied to Q.
Reset Q low on the positive edge of RESET

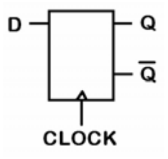

Some Arithmetic Operation
Registers
Some Arithmetic Operation
Registers
Some Arithmetic Operation
Registers

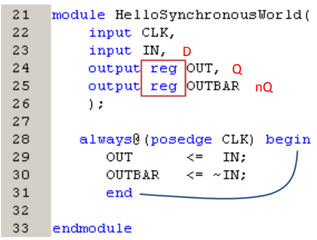
3. Hello Synchronous World

- Concepts
  - Synchronous Logic
  - The Sensitivity List
  - Registers
  - Clocks
- Syntax
  - always@(...)
  - posedge
  - negedge
  - reg
  - Revisit the use of begin and end syntax, but this time in the context of module design
  - forever

3. Hello Synchronous World

Synchronous logic with sensitivity list



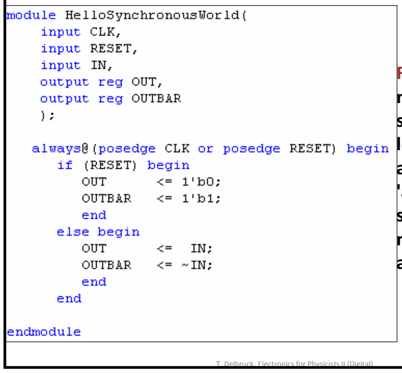


```
module HelloSynchronousWorld(
    input CLK,
    input IN,
    output reg OUT,
    output reg OUTBAR
    );

    always@(posedge CLK) begin
        OUT      <=  IN;
        OUTBAR   <= ~IN;
        end

endmodule
```

No reset here

3. Hello Synchronous World

Sensitivity list, this time with added asynchronous reset



```
module HelloSynchronousWorld(
    input CLK,
    input RESET,
    input IN,
    output reg OUT,
    output reg OUTBAR
    );

  always@(posedge CLK or posedge RESET) begin
     if (RESET) begin
        OUT      <= 1'b0;
        OUTBAR   <= 1'b1;
        end
     else begin
        OUT      <=  IN;
        OUTBAR   <= ~IN;
        end
     end

endmodule
```

Rule: If within a module you have a signal that is on the left hand side of an assignment within an 'always@(...)' statement, then it needs to be defined as a register ('reg').

Aside: Lexical Conventions

- **Comments**

  // Single line comment

  /* Begins multi-line (block) comment
  All text within is ignored
  Line below ends multi-line comment
  */
- **Number**

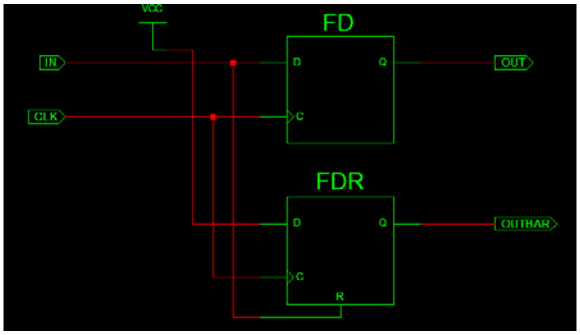
  Sized decimal, hex, octal, binary, e.g. 1'b1='1', 4'he='1110'

  Can include underlines, +,-, e.g. 16'b1111_0011_1100_1101 (for clarity)

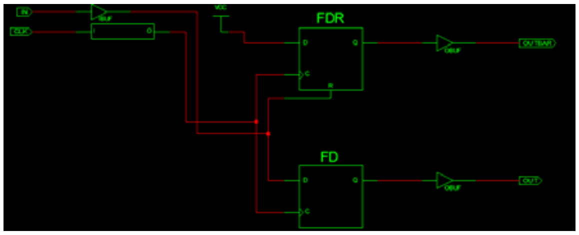
  Can use unsized when target is unambiguous
- **String**

  "   Enclose between quotes on a single line"

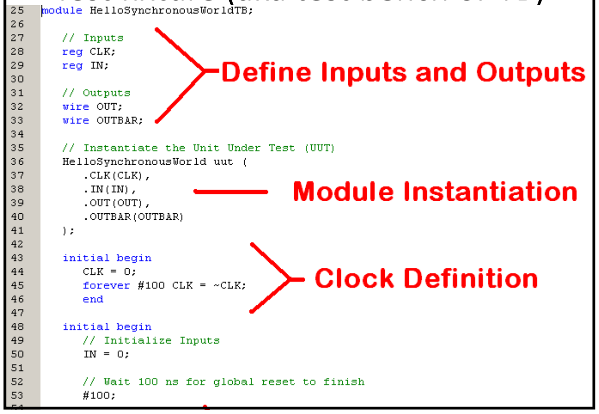
RTL (Register Transfer Level) schematic



Technology schematic



Test fixture (aka test bench or TB)

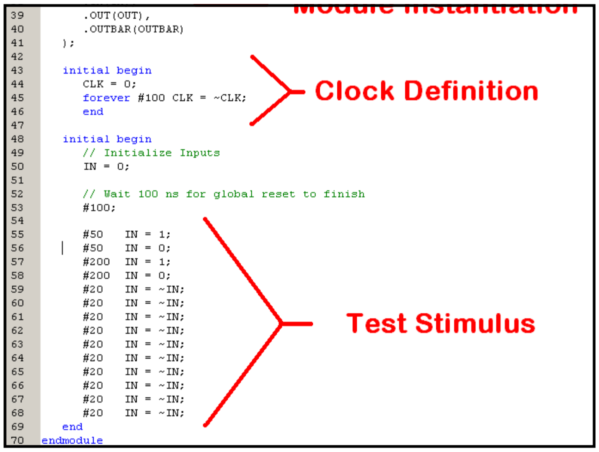


```
module HelloSynchronousWorldTB;

   // Inputs
   reg CLK;
   reg IN;

   // Outputs
   wire OUT;
   wire OUTBAR;

   // Instantiate the Unit Under Test (UUT)
   HelloSynchronousWorld uut (
      .CLK(CLK),
      .IN(IN),
      .OUT(OUT),
      .OUTBAR(OUTBAR)
   );

   initial begin
      CLK = 0;
      forever #100 CLK = ~CLK;
      end

   initial begin
      // Initialize Inputs
      IN = 0;

      // Wait 100 ns for global reset to finish
      #100;
```

Define Inputs and Outputs

Module Instantiation

Clock Definition



Module Instantiation

```
      .OUT(OUT),
      .OUTBAR(OUTBAR)
   );

   initial begin
      CLK = 0;
      forever #100 CLK = ~CLK;
      end

   initial begin
      // Initialize Inputs
      IN = 0;

      // Wait 100 ns for global reset to finish
      #100;

      #50   IN = 1;
      #50   IN = 0;
      #200  IN = 1;
      #200  IN = 0;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
      #20   IN = ~IN;
   end
endmodule
```

Clock Definition

Test Stimulus

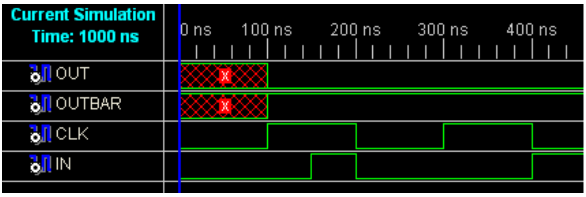
Aside: Procedural Constructs

- Two Procedural Constructs
  - initial Statement
  - always Statement
- initial Statement : Executes only once
- always Statement : Executes in a loop
- Example:

```
initial begin
   Sum = 0;
   Carry = 0;
end
...
```

```
...
always @(A or B) begin
   Sum = A ^ B;
   Carry = A & B;
end
...
```

Simulation



The UCF (user constraints file)

You will need to place one more constraint in your UCF. Place the following line below your pin assignments.

NET "CLK" CLOCK_DEDICATED_ROUTE = TRUE;

- This statement tells the implementation tool that it is ok that this clock signal originates from a non-clock input.
- Real clocks use dedicated clock driver networks on the FPGA.

Aside: Verilog directives and system tasks and functions

Directive	
`include filename	includes a file
`define NROWS 10	defines a string, use it later as e.g. `NROWS (you need prepended single backquote)
`undef NROWS	undefines
`ifdef NROWS	conditional
`else `endif	if/else
`timescale 1 ns / 10 ps	delays (e.g. #10) are in 1ns units with 2 decimal points, e.g. 10ps is .01ns)

System task/function	
$display("time=%t row=%d",$time,row)	print stuff using printf, e.g. %d %b %h, %f %s %m
$monitor("time=%t row=%d",$time,row)	same as $display but when values change
$stop	Pauses and enters debug mode
$finish	Finishes a simulation and exits the simulation process.
$time	time as 32 bit int
$realtime	time as real
$timeformat(-9, 2, " ns", 10);	Controls format of %t 10-9 is unit, 2 digits, ns, min field width
$readmemb("filename", regarray,start,end)	Loads the file into a register memory array. The file must be an ASCII file with values represented in binary ($readmemb) or hex ($readmemh). The start and end address values are optional.

Verilog primer

Two Main Components of Verilog

- Structure (Plumbing, your actual circuit)
  - Verilog program build from modules with I/O interfaces
  - Modules may contain instances of other modules
  - Modules contain local signals, etc.
  - Module configuration is static and all run concurrently
- Concurrent, event-triggered processes (behavioral simulation)
  - *Initial* and *Always* blocks
  - Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)
  - Processes run until they delay for a period of time or wait for a triggering event

Verilog's Two Main Data Types

- Nets (e.g. wire) represent connections between things
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - Cannot be assigned in an *initial* or *always* block
- Reg represents data storage
  - Behave like memory in a computer
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Never connected to something
  - Can be used to model latches, flip-flops, etc., but do not correspond exactly
  - They are shared variables with all their attendant problems

Verilog's Discrete-Event Simulation

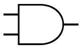
- Basic idea: only do work when something changes
- Centered around an event queue
  - Contains events labeled with the simulated time at which they are to be executed
- Basic simulation paradigm
  - Execute every event for the current simulated time
  - Doing this changes system state and may schedule events in the future
  - When there are no events left at the current time instance, advance simulated time to next soonest event in the queue

Verilog's Four-valued Data

- Verilog's nets and registers hold four-valued data
- 0, 1
  - Obvious
- Z
  - Output of a disabled tri-state driver
  - Models case where nothing is setting a wire's value
- X
  - Models when the simulator can't decide the value
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously
  - Output of a gate with Z inputs

Four-valued Logic Example

- Logical operators work on three-valued logic. Take this AND gate as example

	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

Output 0 if one input is 0

Output X if both inputs are gibberish

Structural Modeling

Nets and Registers

- Wires and registers can be bits, vectors, and arrays

```
wire a;                    // Simple wire
tri [15:0] dbus;           // 16-bit tristate bus
reg [-1:4] vec;            // Six-bit register
integer imem[0:1023];  // Array of 1024 integers
reg [31:0] dcache[0:63];       // A 32-bit memory
```

Modules and Instances

- Basic structure of a Verilog module:

```
module mymod(output1, output2, … input1,
   input2);
output output1;
output [3:0] output2;
input input1;
input [2:0] input2;
…
endmodule
```

Verilog convention lists outputs first

Instantiating a Module

Instances of

```
        module mymod(y, a, b);
```

look like

```
mymod mm1(y1, a1, b1);// Connect-by-position

mymod (y2, a1, b1),
          (y3, a2, b2);        // Instance names omitted

mymod mm2(.a(a2), .b(b2), .y(c2));  // Connect-by-name
```

Gate-level Primitives

Verilog provides the following:

and	nand	logical AND/NAND
or	nor	logical OR/NOR
xor	xnor	logical XOR/XNOR
buf	not	buffer/inverter
bufif0	notif0	Tristate with low enable
bifif1	notif1	Tristate with high enable

Delays on Primitive Instances

- Instances of primitives may include delays

```
buf             b1(a, b);            // Zero delay
buf #3                 b2(c, d);            // Delay of 3
buf #(4,5)  b3(e, f);            // Rise=4, fall=5
buf #(3:4:5)       b4(g, h);            // Min-typ-max
```

User-Defined Primitives (UDPs)

- Defines gates and sequential elements using a truth table
- Often simulate faster than using expressions, collections of primitive gates, etc.
- Gives more control over behavior with X inputs
- Most often used for specifying custom gate libraries

A Carry Primitive

```
primitive carry(out, a, b, c);
output out;
input a, b, c;
table
  00? : 0;
  0?0 : 0;
  ?00 : 0;
  11? : 1;
  1?1 : 1;
  ?11 : 1;
endtable
endprimitive
```

Always have exactly one output

Truth table may include don't-care (?) entries

A Sequential Primitive

```
Primitive dff( q, clk, data);
output q; reg q;
input clk, data;
table
// clk data q   new-q
  (01)   0  : ? :    0;            // Latch a 0
  (01)   1  : ? :    1;            // Latch a 1
  (0x)   1  : 1 :    1;            // Hold when d and q both 1
  (0x)   0  : 0 :    0;            // Hold when d and q both 0
  (?0)   ?  : ? :    -;            // Hold when clk falls
   ?   (??) : ? :    -;            // Hold when clk stable
endtable
endprimitive
```

Continuous Assignment

- Another way to describe combinational function
- Convenient for logical or datapath specifications

Define bus widths

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes

Behavioral Modeling

initial and always blocks

- Basic components for behavioral modeling

```
initial
  begin
    … imperative statements …
  end
```

Runs when simulation starts

Terminates when control reaches the end

Good for providing stimulus

```
always
  begin
    … imperative statements …
  end
```

Runs when simulation starts

Restarts when control reaches the end

Good for modeling/specifying hardware

Initial and Always

- Run until they encounter a delay

```
initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- or a wait for an event

```
always @(posedge clk) q = d;
always begin wait(i); a = 0; wait(~i); a = 1; end
```

Procedural Assignment

- Inside an initial or always block:

```
    sum = a + b + cin;
  LHS       RHS
```

- Just like in C language: RHS evaluated and assigned to LHS before next statement executes
- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg
  - Primitives or continuous assignment may set wire values

Imperative Statements

```
if (select == 1)     y = a;
else                 y = b;

case (op)
  2'b00: y = a + b;
  2'b01: y = a – b;
  2'b10: y = a ^ b;
  default: y = 'hxxxx;
endcase
```

For Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end
```

While Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

i = 0;
while (i <= 15) begin
  output = i;
  #10 i = i + 1;
end
```

Modeling A Flip-Flop With Always

- Very basic: a positive edge-sensitive flip-flop

```
reg q;
always @(posedge clk)
  q = d;
```

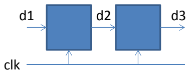
- q = d assignment runs when clock rises: exactly the behavior you expect

Verilog has two types of procedural assignment Blocking vs. Nonblocking

- Blocking (means complete assignment here)
  `a=b;`
- Non-Blocking (store RHS and assign at end of step)
  `a<=b;`
- Fundamental problem:
  - In a hardware synchronous system, all flip-flops sample (almost) simultaneously on the clock edge
  - In Verilog, always @(posedge clk) blocks run in some undefined sequence

A Flawed Shift Register

This doesn't work as you might expect:



```
reg d1, d2, d3;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
```

- These run in some order, but you don't know which. Because assignments are blocking, they run in order, and result depends on order.

Non-blocking Assignments

This version does work:

```
reg d1, d2, d3;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
```

Nonblocking rule:

RHS evaluated when assignment runs

LHS updated only after all events for the current instant have run

But Non-blocking Can Behave Oddly

- A sequence of non-blocking assignments don't communicate.

```
a = 1;
b = a;
c = b;
```

Blocking assignment:
a = b = c = 1

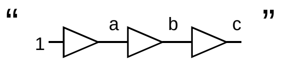
```
a <= 1;
b <= a;
c <= b;
```

Non-blocking assignment:
a = 1
b = old value of a
c = old value of b

Non-blocking Looks Like Latches

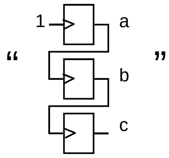
- RHS of blocking taken from wires

```
a = 1;
b = a;
c = b;
```



- RHS of non-blocking taken from latch outputs

```
a <= 1;
b <= a;
c <= b;
```



Building Behavioral Models

Modeling FSMs Behaviorally

There are many ways to do it:

1. Define the next-state logic combinationally and define the state-holding latches explicitly
2. Define the behavior in a single always @(posedge clk) block
3. Variations on these themes

FSM with Combinational Logic

```
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;

always @(a or b or state)
 case (state)
   2'b00: begin
      nextState = a ? 2'b00 : 2'b01;
      o = a & b;
   end
   2'b01: begin nextState = 2'b10; o = 0; end
 endcase
```

Output o is declared a reg because it is assigned procedurally, not because it holds state

Combinational block must be sensitive to any change on any of its inputs

(Implies state-holding elements otherwise)

FSM with Combinational Logic

```
module FSM(o, a, b, reset);
…

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;
```

Latch implied by sensitivity to the clock or reset only

FSM from Combinational Logic

```
always @(a or b or state)
 case (state)
   2'b00: begin
     nextState = a ? 2'b00 : 2'b01;
     o = a & b;
   end
   2'b01: begin nextState = 2'b10; o = 0; end
 endcase

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;
```

This is a Mealy machine because the output is directly affected by any change on the input

FSM from a Single Always Block

```
module FSM(o, a, b);
output o; reg o;
input a, b;
reg [1:0] state;

always @(posedge clk or reset)
  if (reset) state <= 2'b00;
  else case (state)
    2'b00: begin
      state <= a ? 2'b00 : 2'b01;
      o <= a & b;
    end
    2'b01: begin state <= 2'b10; o <= 0; end
 endcase
```

Expresses Moore machine behavior:

Outputs are latched

Inputs only sampled at clock edges

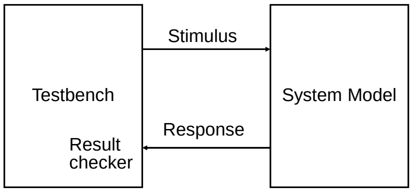
Nonblocking assignments used throughout to ensure coherency.

RHS refers to values calculated in previous clock cycle

Simulating Verilog

How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously



Writing Testbenches

```
module test;
reg a, b, sel;

mux m(y, a, b, sel);

initial begin
  $monitor($time,, "a = %b b=%b sel=%b y=%b",
           a, b, sel, y);
  a = 0; b= 0; sel = 0;
  #10 a = 1;
  #10 sel = 1;
  #10 b = 1;
end
```

Inputs to device under test

Device under test

$monitor is a built-in event driven "printf"

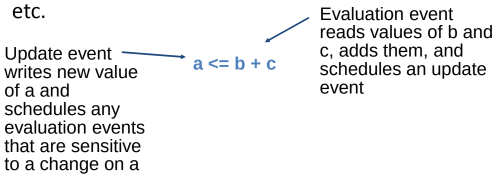
Stimulus generated by sequence of assignments and delays

Simulation Behavior

- Scheduled using an event queue
- Non-preemptive, no priorities
- A process must explicitly request a context switch
- Events at a particular time unordered
- Scheduler runs each event at the current time, possibly scheduling more as a result

Two Types of Events

- Evaluation events compute functions of inputs
- Update events change outputs
- Split necessary for delays, nonblocking assignments, etc.


Update event writes new value of a and schedules any evaluation events that are sensitive to a change on a
a <= b + c
Evaluation event reads values of b and c, adds them, and schedules an update event

Simulation Behavior

- Concurrent processes (initial, always) run until they stop at one of the following

- #42
  - Schedule process to resume 42 time units from now
- wait(cf & of)
  - Resume when expression “cf & of” becomes true
- @(a or b or y)
  - Resume when a, b, or y changes
- @(posedge clk)
  - Resume when clk changes from 0 to 1

Simulation Behavior

- Infinite loops are possible and the simulator does not check for them
- This runs forever: no context switch allowed, so ready can never change

```
while (~ready)
  count = count + 1;
```

- Instead, use

```
wait(ready);
```

Simulation Behavior

- Race conditions abound in Verilog

- These can execute in either order: final value of a undefined:

```
always @(posedge clk) a = 0;
always @(posedge clk) a = 1;
```

Simulation Behavior

- Semantics of the language closely tied to simulator implementation

- Context switching behavior convenient for simulation, not always best way to model
- Undefined execution order convenient for implementing event queue

Verilog and Logic Synthesis

Logic Synthesis

Verilog is used in two ways

- Model for discrete-event simulation
- Specification for a logic synthesis system

- Logic synthesis converts a subset of the Verilog language into an efficient netlist
- It’s one of the major breakthroughs in designing logic chips in the last 30 years
- Most chips are designed using at least some logic synthesis

Logic Synthesis

Takes place in two stages:

- Translation of Verilog (or VHDL) source to a netlist
  - Register inference
- Optimization of the resulting netlist to improve speed and area
  - Most critical part of the process
  - Algorithms very complicated

Translating Verilog into Gates

- Parts of the language easy to translate
  - Structural descriptions with primitives
    - Already a netlist
  - Continuous assignment
    - Expressions turn into little datapaths
- Behavioral statements the bigger challenge

What Can Be Synthesized

- Structural definitions
  - Everything
- Behavioral blocks
  - Depends on sensitivity list
  - Only when they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
  - Blocks sensitive to both edges of the clock, changes on unrelated signals, changing sensitivity lists, etc. cannot be synthesized
- User-defined primitives (UDP)
  - Primitives defined with truth tables
  - Some sequential UDPs can’t be translated (not latches or flip-flops)

What Isn’t Translated

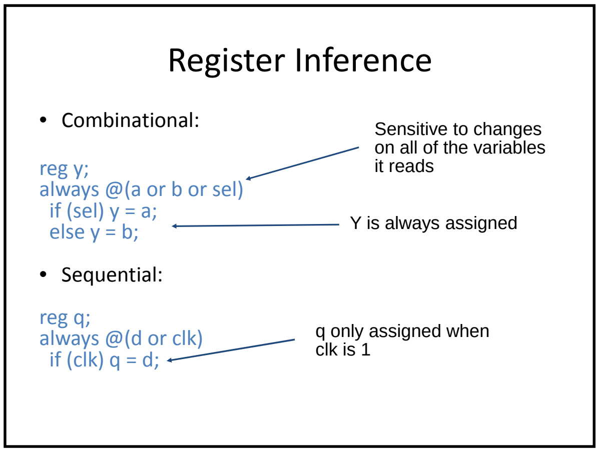
- Initial blocks
  - Used to set up initial state or describe finite testbench stimuli
  - Don’t have obvious hardware component
- Delays
  - May be in the Verilog source, but are simply ignored
- A variety of other obscure language features
  - In general, things heavily dependent on discrete-event simulation semantics
  - Certain “disable” statements
  - Pure events

Register Inference

The main trick

- reg does not always equal latch

- Rule: Combinational if outputs always depend exclusively on sensitivity list
- Sequential if outputs may also depend on previous values



Register Inference

- Combinational:

```
reg y;
always @(a or b or sel)
  if (sel) y = a;
  else y = b;
```

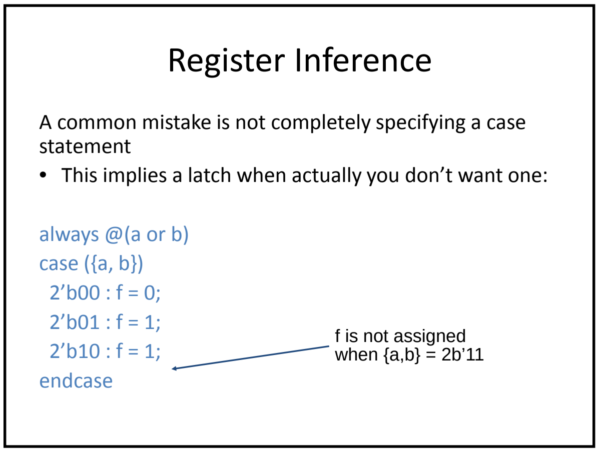
Sensitive to changes on all of the variables it reads

Y is always assigned

- Sequential:

```
reg q;
always @(d or clk)
  if (clk) q = d;
```

q only assigned when clk is 1



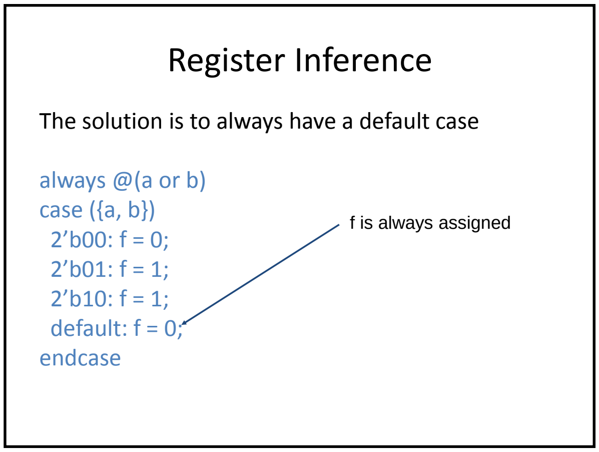
Register Inference

A common mistake is not completely specifying a case statement

- This implies a latch when actually you don't want one:

```
always @(a or b)
case ({a, b})
  2'b00 : f = 0;
  2'b01 : f = 1;
  2'b10 : f = 1;
endcase
```

f is not assigned when {a,b} = 2b'11

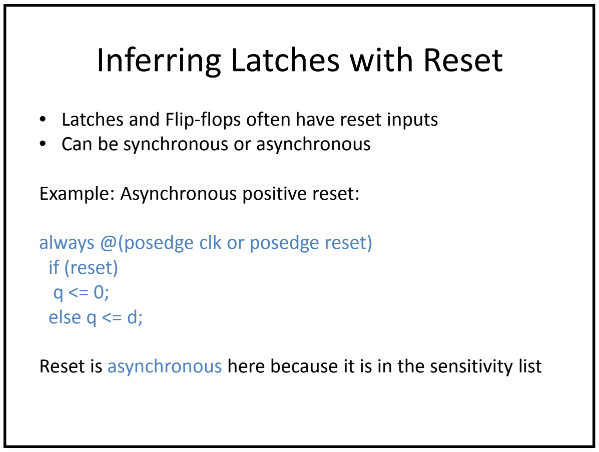


Register Inference

The solution is to always have a default case

```
always @(a or b)
case ({a, b})
  2'b00: f = 0;
  2'b01: f = 1;
  2'b10: f = 1;
  default: f = 0;
endcase
```

f is always assigned



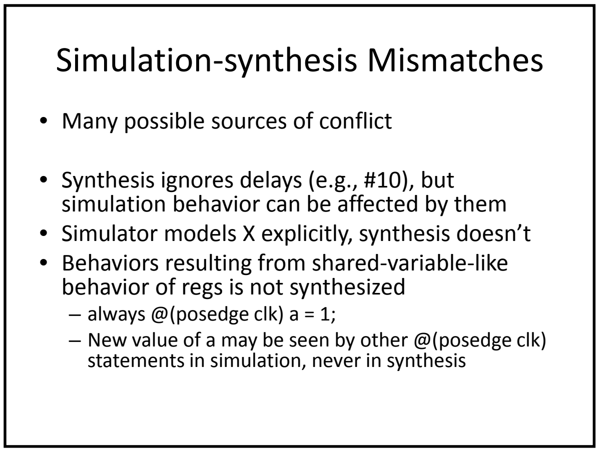
Inferring Latches with Reset

- Latches and Flip-flops often have reset inputs
- Can be synchronous or asynchronous

Example: Asynchronous positive reset:

```
always @(posedge clk or posedge reset)
  if (reset)
   q <= 0;
  else q <= d;
```

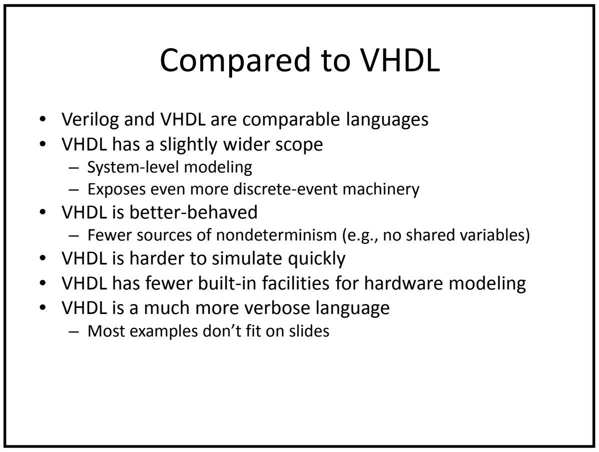
Reset is asynchronous here because it is in the sensitivity list



Simulation-synthesis Mismatches

- Many possible sources of conflict

- Synthesis ignores delays (e.g., #10), but simulation behavior can be affected by them
- Simulator models X explicitly, synthesis doesn't
- Behaviors resulting from shared-variable-like behavior of regs is not synthesized
  - always @(posedge clk) a = 1;
  - New value of a may be seen by other @(posedge clk) statements in simulation, never in synthesis



Compared to VHDL

- Verilog and VHDL are comparable languages
- VHDL has a slightly wider scope
  - System-level modeling
  - Exposes even more discrete-event machinery
- VHDL is better-behaved
  - Fewer sources of nondeterminism (e.g., no shared variables)
- VHDL is harder to simulate quickly
- VHDL has fewer built-in facilities for hardware modeling
- VHDL is a much more verbose language
  - Most examples don't fit on slides

“Gateway” lab exercises

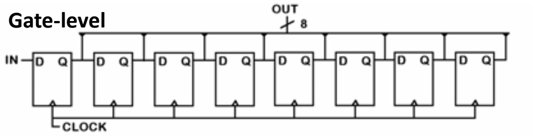
4. ShiftingTheWorld - synthesizing a shift register with *fd* D-FlipFlops using gate level and behavioral level design; *register transfer level (RTL)* design; *module instantiation*; signal *concatenation*; introduction to *generate*.
5. ShiftingManyWorlds - 2d array of shift registers (memory); simulation exercise.

4. Shifting the world

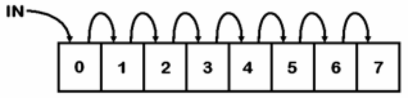
Gate-level vs. Behavioral

Sensitivity list, modules, “generate”, “{}” syntax

Gate-level

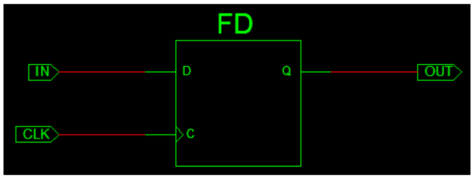


Behavioral here is more precisely referred to as "RTL" or Register Transfer Level description. In it, a circuit's behavior is defined in terms of the transfer of signals between registers and the logical operations performed on those signals.



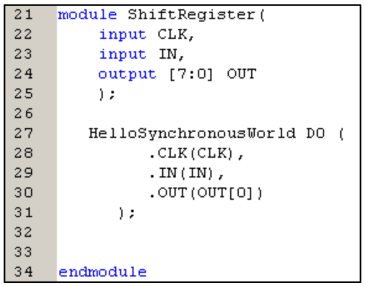
Make a D flip flop (*fd*)

Remove the qbar output from previous exercise.



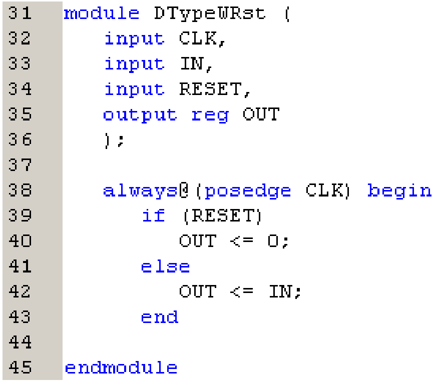
Make top level ShiftRegister

- Example of top level with one shift register



```
module ShiftRegister(
    input CLK,
    input IN,
    output [7:0] OUT
    );

   HelloSynchronousWorld D0 (
         .CLK(CLK),
         .IN(IN),
         .OUT(OUT[0])
      );


endmodule
```

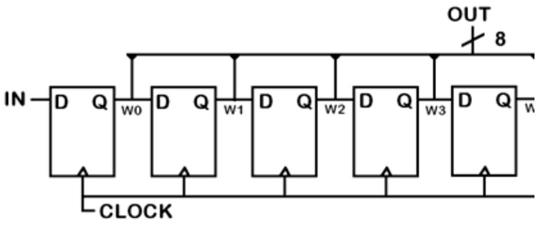
Describe D FlipFlop with Reset



```
module DTypeWRst (
    input CLK,
    input IN,
    input RESET,
    output reg OUT
    );

    always@(posedge CLK) begin
       if (RESET)
          OUT <= 0;
       else
          OUT <= IN;
       end

endmodule
```

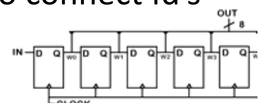
Use a wire to connect fd’s

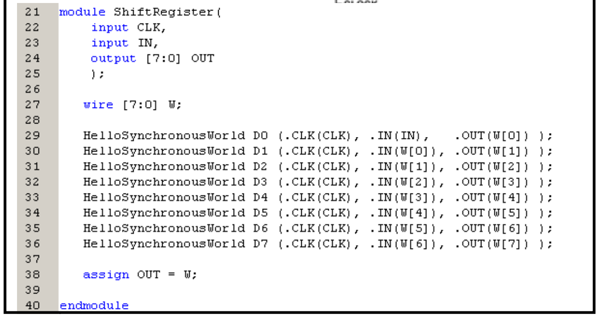
- **wire [7:0] MyWire;**



Use a wire to connect fd's

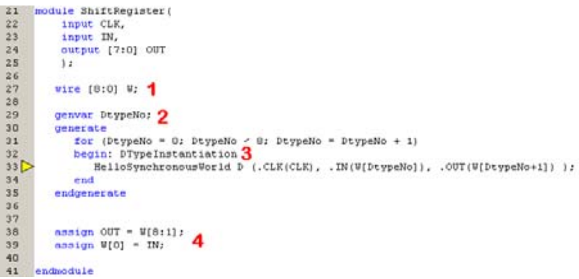
- **wire [7:0] MyWire;**





```
module ShiftRegister(
    input CLK,
    input IN,
    output [7:0] OUT
    );

   wire [7:0] W;

   HelloSynchronousWorld D0 (.CLK(CLK), .IN(IN),   .OUT(W[0]) );
   HelloSynchronousWorld D1 (.CLK(CLK), .IN(W[0]), .OUT(W[1]) );
   HelloSynchronousWorld D2 (.CLK(CLK), .IN(W[1]), .OUT(W[2]) );
   HelloSynchronousWorld D3 (.CLK(CLK), .IN(W[2]), .OUT(W[3]) );
   HelloSynchronousWorld D4 (.CLK(CLK), .IN(W[3]), .OUT(W[4]) );
   HelloSynchronousWorld D5 (.CLK(CLK), .IN(W[4]), .OUT(W[5]) );
   HelloSynchronousWorld D6 (.CLK(CLK), .IN(W[5]), .OUT(W[6]) );
   HelloSynchronousWorld D7 (.CLK(CLK), .IN(W[6]), .OUT(W[7]) );

   assign OUT = W;

endmodule
```

Now do same with generate



```
module ShiftRegister(
    input CLK,
    input IN,
    output [7:0] OUT
    );

   wire [8:0] W; 1

   genvar DtypeNo; 2
   generate
      for (DtypeNo = 0; DtypeNo < 8; DtypeNo = DtypeNo + 1)
      begin: DTypeInstantiation 3
         HelloSynchronousWorld D (.CLK(CLK), .IN(W[DtypeNo]), .OUT(W[DtypeNo+1]) );
      end
   endgenerate


   assign OUT = W[8:1];
   assign W[0] = IN;        4

endmodule
```

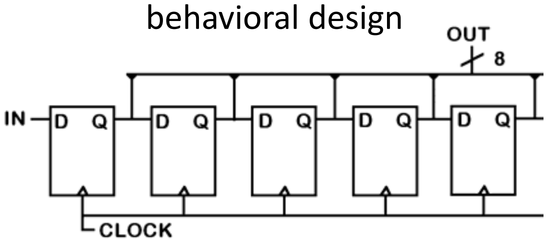
Now we will do the same with behavioral design

A synchronously resetting DType could be described as:

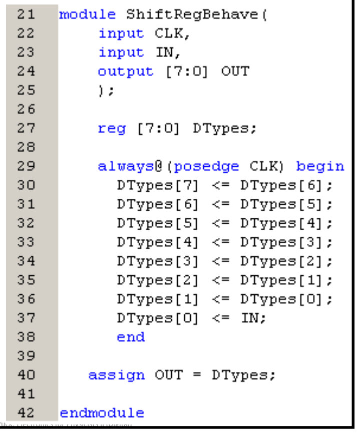
On the rising edge of the clock, if RESET is '1' then OUT takes the value of '0' else OUT takes the value of IN.

"On the rising edge of the clock" describes the sensitivity list. The "if RESET is '1' then" describes a conditional statement with two outcomes, and the remainder of the sentence describes how to deal with the two outcomes, with the "takes the value of" text denoting '<=' syntax.

Now we will do the same with behavioral design

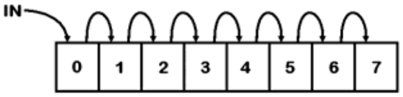


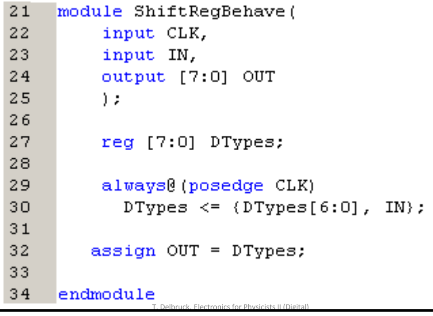
- Will synthesize to exact same as our gate level design but doesn't require us to define the D-flip flip.
- There is a shorter way to write this using *concatenation*



```
module ShiftRegBehave(
    input CLK,
    input IN,
    output [7:0] OUT
    );

    reg [7:0] DTypes;

    always@(posedge CLK) begin
      DTypes[7] <= DTypes[6];
      DTypes[6] <= DTypes[5];
      DTypes[5] <= DTypes[4];
      DTypes[4] <= DTypes[3];
      DTypes[3] <= DTypes[2];
      DTypes[2] <= DTypes[1];
      DTypes[1] <= DTypes[0];
      DTypes[0] <= IN;
      end

   assign OUT = DTypes;

endmodule
```

Concatenation example





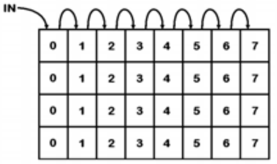
```
module ShiftRegBehave(
    input CLK,
    input IN,
    output [7:0] OUT
    );

    reg [7:0] DTypes;

    always@(posedge CLK)
      DTypes <= {DTypes[6:0], IN};

   assign OUT = DTypes;

endmodule
```

“Gateway” lab exercises

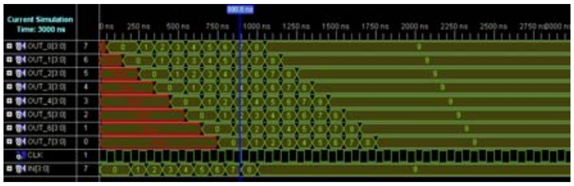
4. ShiftingTheWorld - synthesizing a shift register with *fd* D-FlipFlops using gate level and behavioral level design; *register transfer level (RTL)* design; *module instantiation*; signal*concatenation*; introduction to *generate*.
5. ShiftingManyWorlds - 2d array of shift registers (memory); simulation exercise.

5. Shifting many worlds

(simulation-only exercise)



0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7



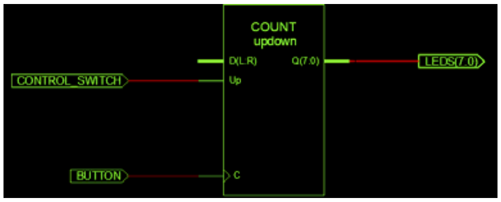
“Gateway” lab exercises

6. CountingWorlds - simple arithmetic, ***multiplexing***.
7. TimingTheWorld - a second-counter watch using two counters, one clocking the other, both up/down with enable.
8. DecodingTheWorld - Number representation; 7-segment display *decoder* (see BASYS2 manual). See 7seg for the code for this exercise.
9. TimingTheWorldInDecimel - multiple counters, using ***generics*** to instantiate modules with parameters; revisit *generate*.

6. Counting the world

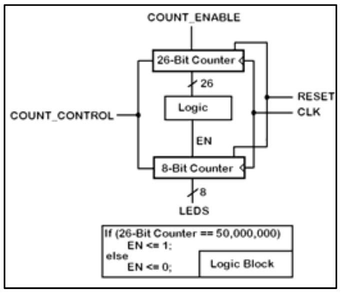
Arithmetic, multiplexing, +/-, if/else

On the rising edge of the signal from the Button, **if** the control signal from the slide switch is '1' **then** the number stored in the register takes the value of the number stored in the register **plus** one. **Else** then the number stored in the register takes the value of the number stored in the register **minus** one.

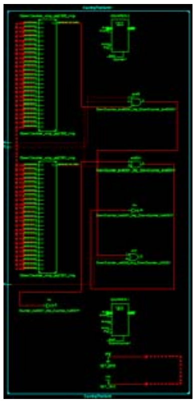


7. Timing the world

- Using on-board clock (instead of switch)
- Multiplexing

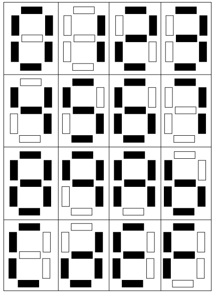


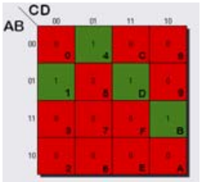
```
If (26-Bit Counter == 50,000,000)
    EN <= 1;
else
    EN <= 0;
```



8. Decoding the world

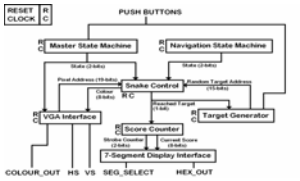
- Combinational logic, Karnaugh maps
- Module reuse with wrappers





Preview of coming exercises

7. Timing the world
8. Decoding the world (combinational logic)
9. Timing the world in decimal
10. Coloring the world (VGA interface)
11. **The world of state machines (important!)**
12. **The world of linked state machines**
13. The snake game



"Gateway" lab exercises

10. ColourTheWorld - *parameters* in generics; VGA display control to generate sync signals and RGB colors (see BASYS2 manual).
11. WorldOfStateMachines - making ***state machines*** using sequential and combinational blocks (switch/case statements) and using ROM modules (**$readmemb**).
12. WorldOfLinkedStateMachines - multiple state machines linked by a master state machine.
13. Snake - a complete snake game.