Spiral 1 / Unit 4 Verilog HDL

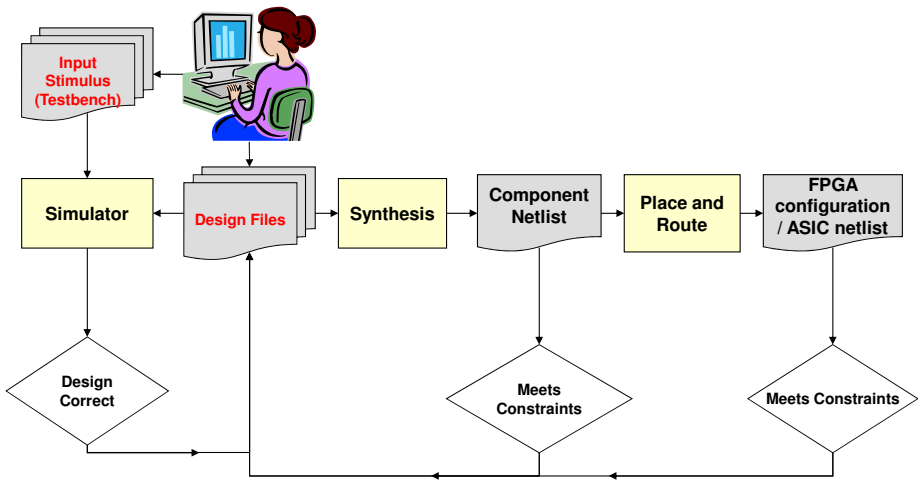
Mark Redekopp

OVERVIEW

Digital Circuit Design Steps

- Description
  - Design and computer-entry of circuit
- Verification
  - Simulate design for correctness
- Synthesis
  - Determine components/gates and their connections
- Place and Route
  - Determine the physical placement and wire connections between components on actual silicon
- ASIC Fabrication / FPGA implementation

Digital Circuit Design

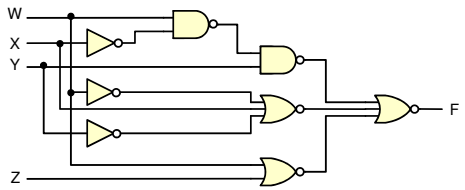


Step 1: Description

- Much of the design process is done by a computer
- Human designers must describe and capture their circuits into a format a computer can use
- 1 form for use usually only at the transistor level:
  - Schematic Entry: computerized drawing of the gates/transistors and components and their connections
- 2 forms used for large digital designs
  - HDL (Hardware Description Language): text description of circuit (similar to programming languages)
  - Behavioral descriptions (C, Matlab, etc.)

Schematic Entry

- Schematics
  - Graphically "draw" the gates, components, and connecting wires of a design
  - Requires design at the structural level (i.e. must specify design down to the exact gate interconnections)
  - Hard to manage for large designs
  - Not as commonly used in industry as HDL's



HDL's

- "Programming" languages that describe hardware components (e.g. Verilog, VHDL)
- Functional descriptions (describe function at high level) or structural descriptions of digital components
- Easier to manage large designs

Functional

```
assign F = WX + ~WY
```

or

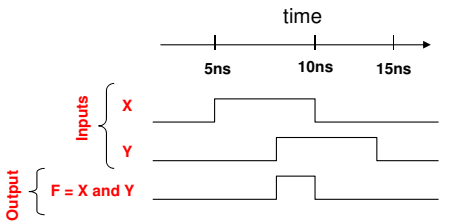
```
if (W==1 && X==1)
  F <= 1;
else if (W==0 && Y==1)
  F <= 1;
else
  F <= 0;
```

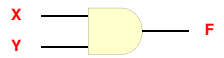
Structural

```
and mygate0(n1,w,x);
not mygate1(not_w, w);
and mygate2(n2,not_w,y)
or mygate3(f,n1,n2);
```


Step 2: Simulation

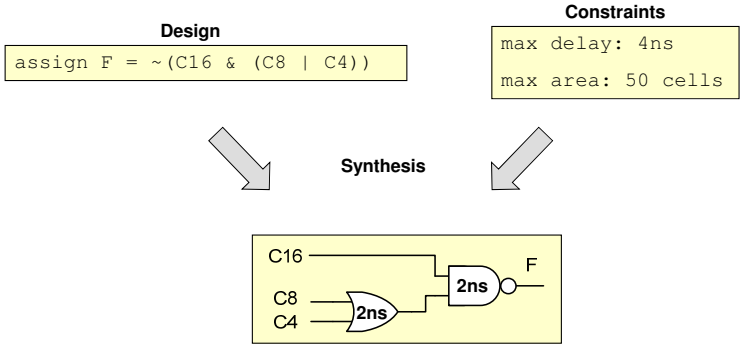
- Exercises the description of the circuit
- Designer provides input stimulus to the circuit
  - Set X=1 at 5 ns.
  - Set Y=1 at 8 ns.
- Simulator will run inputs through your proposed circuit and show the outputs it would generate
- Use waveforms (values over time to see the behavior of a circuit)
- Designer must know what to expect and check against what is produced





Step 3: Synthesis

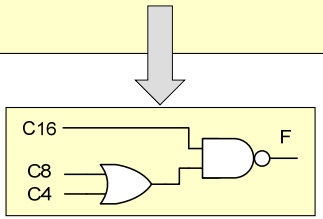
- Takes in design files along with time and area constraints to find what parts are needed and how they should be connected



Step 3: Synthesis

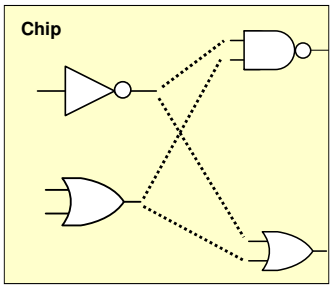
- Able to take a functional description and convert to AND/OR gate design

```
assign count[4:0] = {C16,C8,C4,C2,C1};

always @*
begin
  if(count < 20)
    F = 1;
  else
    F = 0;
end
```



Step 4: Place and Route

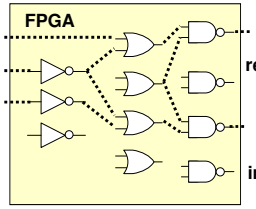
- Finds where each gate should be placed on the chip and how to route the wires that connect to it
- Affects timing and area
  - wiring takes up space and longer wires lead to longer delays



Digital Design Targets

- Two possible implementation targets
  - Custom Chips (ASIC's = Application Specific Integrated Circuits): Physical gates are created on silicon to implement 1 particular design
  - FPGA (Field Programmable Gate Array's): Prefabricated chips that we can configure and reconfigure to perform digital logic functions

In an ASIC design, a unique chip will be manufactured that implements our design and cannot be reconfigured (example: Pentium, etc.)



FPGA's have logic resources on them that we can configure to implement our specific design. We can then reconfigure it to implement another design

VERILOG AND HDLS

Purpose

- HDL's were originally used to model and simulate hardware before building it
- In the past 20 years, synthesis tools were developed that can essentially build the hardware from the same description
- Common ones:
  - Verilog and SystemVerilog
  - VHDL
  - SystemC

Differences from Software

- Software programming languages are inherently sequential
  - Operations executed in sequential order (next, next, next)
- Hardware blocks always run in parallel (at the same time)
  - Uses event-driven paradigm (change in inputs causes expression to be evaluated)
- HDL's provide constructs for both parallel & sequential operation

Software
Perform x+y and when that is done assign d-c to tmp

```
var = x+y;
tmp = d-c;
```

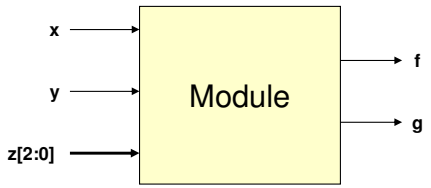
Hardware
This description models 2 gates working at the same time

```
assign f = a & b;
assign g = a | b;
```

Event Driven Paradigm: If a or b changes, f and g will be re-evaluated

Modules

- Each Verilog designs starts as a block diagram (called a "module" in Verilog)
- Start with input and output signals, then describe how to produce outputs from inputs

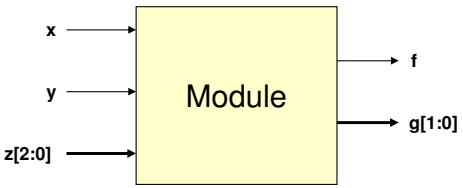



```
module m1(x,y,z,f,g);
// circuit
// description
endmodule
```

Software analogy: Modules are like functions, but also like classes in that they are objects that you can instantiate multiple times.

Ports

- Input and output signals of a module are called "ports" (similar to parameters/arguments of a software function)
- Unlike software, ports need to be declared as "input" or "output"
- Vectors declared using [MSB : LSB] notation



These are the ports

```
module m1(x,y,z,f,g);

input  x,y;
input  [2:0] z;
output f;
output [1:0] g;

endmodule
```


Signal Types

- Signals represent the inputs, outputs, and internal values
- Signals need to be typed
  - Similar to variables in software (e.g. int, char)
- 2 basic types
  - Wire: Represents a node connecting two logic elements
    - Only for modeling combinational logic
    - Used in "assign" statements
    - Use for signals connecting outputs of instantiated modules (structural modeling)
  - Reg: Used for signals that are described behaviorally
    - Used to model combinational & sequential logic
    - Used for anything produced by an "always" or "initial" block

```
module m1(x,y,z,f,g);

 input  x,y;
 input  [2:0] z
 output f;
 output reg [1:0] g;


 wire   n1, n2;
 reg    n3, n4;
 ...
endmodule
```

Inputs are always type 'wire'. Outputs are assumed 'wire' but can be redefined as 'reg'

Constants

- Multiple bit constants can be written in the form:
  - [size] `base value
    - *size* is number of bits in constant
    - *base* is o or O for octal, b or B for binary, d or D for decimal, h or H for hexadecimal
    - *value* is sequence of digits valid for specified *base*
      - Values a through f (for hexadecimal base) are case-insensitive
- Examples:
  - 4'b0000 // 4-bits **b**inary
  - 6'b101101 // 6-bits **b**inary
  - 8'hfC // 8-bits in **h**ex
  - Decimal is default
  - 17 // 17 decimal converted to appropriate # of unsigned bits

Structural vs. Behavioral Modeling

Structural

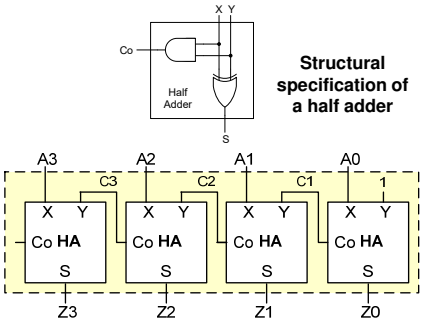
- Starting with gates, build up a hierarchy of components and specify how they should be connected

Behavioral

- Describe behavior and let synthesis tools select internal components and connections

Structural Modeling

- Starting with primitive gates, build up a hierarchy of components and specify how they should be connected



Structural specification of a half adder

Use HA's to structurally describe incrementer

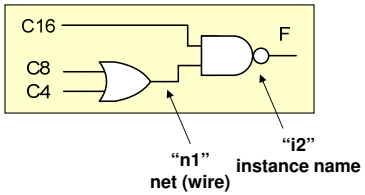
```
module ha(x,y,s,co);
  input    x,y;
  output   s,co;

  assign s = x ^ y; // xor
  assign co = x & y; // and
endmodule

module incrementer(a,z);
  input    [3:0] a;
  output   [3:0] z;
  wire     [3:1] c;

  ha ha0(a[0],1,z[0],c[1]);
  ha ha1(a[1],c[1],z[1],c[2]);
  ha ha2(a[2],c[2],z[2],c[3]);
  ha ha3(a[3],c[3],z[3], );

endmodule
```


Structural Modeling of Logic Gates

- Modules and primitive gates can be instantiated using the following format:

module_name instance_name(output, input1, input2,…)

- Input and outputs must be wire types
- Supported Gates: *and, or, not, nand, nor, xor, xnor*




```
module m1(c16,c8,c4,f);
  input    c16,c8,c4;
  output   f;
  wire     n1;

  assign n1 = c8 | c4;
  assign f = ~(c16 & n1);

endmodule
```

Verilog Description

Instantiating User-Defined Modules

- Format: module_name instance_name(port1, port2, port3, …)
- Positional mapping
  - Signals of instantiation ports are associated using the order of module's port declaration (i.e. order is everything)
- Named mapping
  - Signals of instantiation ports are explicitly associated with module's ports (i.e. order is unimportant)
  - module_name instance_name(.module_port_name(signal_name),…);

```
module ha(x,y,s,co);
  ...
endmodule

module incrementer(a,z);
  ha ha0(a[0],1,z[0],c[1]);
  ...
endmodule
```

Positional mapping

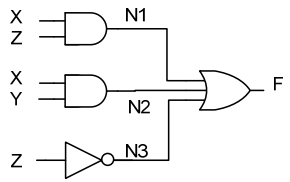
```
module ha(x,y,s,co);
  ...
endmodule

module incrementer(a,z);
  ha ha0(.x(a[0]),
         .s(z[0]),
         .y(1),
         .co(c[1]) );
  ...
endmodule
```

Named Mapping

Internal Signals

- Define signals (wire or reg) for each internal signal/wire

```
module m2(x,y,z,f);

input  x,y,z;
output f;
wire   n1,n2,n3;

assign n1 = x & z;
assign n2 = x & y;
assign n3 = ~z;

assign f = n1 | n2 | n3;

endmodule
```



Behavioral Modeling

- Describe behavior and let synthesis tools select internal components and connections
- Advantages:
  - Easier to specify
  - Synthesis tool can pick appropriate implementation (for speed / area / etc.)

```
module incrementer(a,z);
  input    [3:0] a;
  output   [3:0] z;

  assign z = a + 1'b1;

endmodule
```

Could instantiate a ripple-carry adder, a fast carry-lookahead adder, etc. as needed

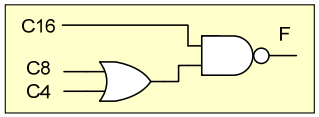
Use higher level operations and let synthesis tools infer the necessary logic

Operators

- Operator types
  - Non-blocking / Blocking assignment (<=, =)
  - Arithmetic (+, -, *, /, %)
  - Relational (<, <=, >, >=)
  - Equality (= =, !=, = = = , ! = =)
  - Logical (&&, ||, !)
  - Bitwise (~, &, |, ^, ~^)
  - Reduction (&, ~&, |, ~|, ^, ~^)
  - Shift (<<, >>)
  - Conditional (? :)
  - Concatenation and replication

Assign Statement

- Used for combinational logic expressions (must output to a 'wire' signal type)
- Can be used anywhere in the body of a module's code
- All 'assign' statements run in parallel
- Change of any signal on RHS (right-hand side) triggers re-evaluation of LHS (output)
- Format:
  - **assign *output* = *expr;***
    - '&' means AND
    - '|' means OR
    - '~' means NOT
    - '^' means XOR




```
module m1(c16,c8,c4,f);
  input    c16,c8,c4;
  output   f;
  wire     n1;

 assign f = ~(c16 & (c8 | c4));

endmodule
```


Multi-bit (Vector) Signals

- Reference individual bits or groups of bits by placing the desired index in brackets (e.g. x[3] or x[2:1])
- Form vector from individual signals by placing signals in brackets (i.e. { }) and separate with commas

```
module m1(x,f);

  input [2:0] x;
  output      f;

  // f = minterm 5
  assign f = x[2] & ~x[1] & x[0];

endmodule
```

```
module incrementer(a,x,y,z);

  input [2:0] a;
  output x,y,z;


  assign {x,y,z} = a + 1;

endmodule
```

More Assign Statement

- Can be used with other operators besides simple logic functions
  - Arithmetic (+, -, *, /, %=modulo/remainder)
  - Shifting (<<, >>)
  - Relational (<, <=, >, >=, !=, ==)
    - Produces a single bit output ('1' = true / '0' false)
  - Conditional operator (? :)
    - Syntax: condition ? statement_if_true : statement_if_false;

```
module m1(x,y,sub,s,cout,d,z,f,g);
  input    [3:0] x,y;
  input          sub;
  output   [3:0] s,d;
  output   [3:0] z;
  output         cout,f,g;

  assign {cout,s} = {0,x} + {0,y};
  assign d = x - y;
  assign f = (x == 4'h5);
  assign g = (y < 0);
  assign z = (sub==1) ? x-y : x+y;

endmodule
```

Sample "Assign" statements

Understanding Simulation Timing

- When expressing parallelism, an understanding of how time works is crucial
- Even though 'always' and 'assign' statements specify operations to be run in parallel, simulator tools run on traditional computers that can only execute sequential operations
- To maintain the appearance of parallelism, the simulator keeps track of events in a sorted event queue and updates signal values at appropriate times, triggering more statements to be executed

Explicit Time Delays

- In testbenches, explicit delays can be specified using '# delay'
  - When this is done, the RHS of the expression is evaluated at time *t* but the LHS is not updated until *t+delay*

```
module m1_tb;
  reg  a,b,c;
  wire  w,x,y,z;
  initial begin
    a = 1;
    #5  // delay 5 ns (ns=default)
    a = 0;
    b = 0;
    #2  // delay 2 more ns
    a = 1;
endmodule
```

Simulator Event Queue

Time	Event
0 ns	a = 1
5 ns	a = 0
5 ns	b = 0
7 ns	a = 1

Explicit Time Delays

- Assignments to the same signal without an intervening delay will cause only the last assignment to be seen

```
module m1_tb;
  reg  a,b,c;
  wire  w,x,y,z;
  initial begin
    a = 1;
    #5  // delay 5 ns (ns=default)
    a = 0;
    a = 1;
endmodule
```

Simulator Event Queue

Time	Event
0 ns	a = 1
5 ns	a = ~~0~~→1

Implicit Time Delays

- Normal behavioral descriptions don't model propagation delay until the code is synthesized
- To operate correctly the simulators event queue must have some notion of what happens first, second, third, etc.
- Delta (δ) time is used
  - Delta times are purely for ordering events and all occur in "0 time"
  - The first event(s) occur at time 0 ns
  - Next event(s) occur at time 0 + δ
  - Next event(s) occur at time 0 + 2δ

```
always @(a,b,c,w,x,y)
begin
   w <= a ^ b;
   x <= b | c;
   y <= w & x;
   z <= ~y;
end
```

Equivalent Implementations

```
assign w = a ^ b;
assign x = b | c;
assign y = w & x;
assign z = ~y;
```

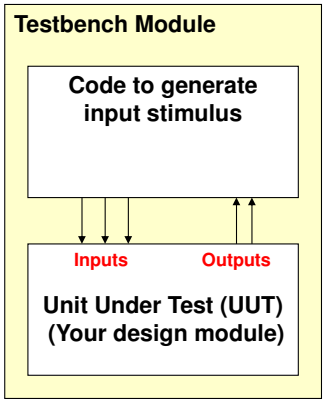
Time	Event	Triggers
0 ns	a,b,c = 0,0,1	w and x assigns
0 + δ	w=0, x=1	y assign
0 + 2δ	y = 0	z assign
0 + 3δ	z = 1	Anything sensitive to z

Simulator Event Queue

TESTBENCHES

Testbenches

- Generate input stimulus (values) to your design over time
- Simulator will run the inputs through the circuit you described and find what the output from your circuit would be
- Designer checks whether the output is as expected, given the input sequence
- Testbenches consist of code to generate the inputs as well as instantiating the design/unit under test and possibly automatically checking the results



Testbench Modules

- Declared as a module just like the design circuit
- No inputs or outputs

```
module my_tb;
  // testbench code
endmodule
```

Testbench Signals

- Declare signals in the testbench for the inputs and outputs of the design under test
  - inputs to your design should be declared type 'reg' in the testbench (since you are driving them and their value should be retained until you change them)
  - outputs from your design should be declared type 'wire' since your design is driving them

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;

...
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

endmodule
```

Testbench

UUT Instantiation

- Instantiate your design module as a ***component*** (just like you instantiate a gate in you design)
- Pass the input and output signals to the ports of the design
- For designs with more than 4 or 5 ports, use named mapping rather than positional mapping

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;
  ...
endmodule
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);
  /* m1 uut(.x(x), .y(y),
            .z(z), .f(f),
            .g(g));
  */
endmodule
```

Testbench

Generating Input Stimulus (Values)

- Now use Verilog code to generate the input values over a period of time

```
module m1(x,y,z,f,g);

  input  x,y,z;
  output f,g;
  ...
endmodule
```

Unit Under Test

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);
  /* m1 uut(.x(x), .y(y),
            .z(z), .f(f),
            .g(g));
  */
endmodule
```

Testbench

Initial Block Statement

- Tells the simulator to run this code just once (vs. always block that runs on changes in sensitivity list signals)
- Inside the "initial" block we can write code to generate values on the inputs to our design
- Use "begin…end" to bracket the code (similar to { .. } in C or Java)

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    // input stimulus
    // code

  end

endmodule
```

Testbench

Assignment Statement

- Use '=' to assign a signal a value
  - Can assign constants
    - x = 0; y = 1;
  - Can assign logical relationships
    - x = ~x // x = not x
    - x = y & z // x = y and z

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    x = 0;

  end

endmodule
```

Testbench

Aggregate Assignment Statement

- Can assign multiple signals at once
- Place signals in brackets (i.e. { }) and separate with commas
- Multiple bit constants can be written in the form:
- *num_bits* '{b,o,d,h} *value*
  - 4'b0000 // 4-bits **b**inary
  - 6'b101101 // 6-bits **b**inary
  - 8'hFF // 8-bits in **h**ex
  - Decimal is default
  - 17 // 17 decimal

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 uut(x,y,z,f,g);

  initial
  begin

    {x,y,z} = 3'b000;

  end

endmodule
```

Testbench

Time

- We must explicitly indicate when and how much time should pass between assignments
- Statement ('#' indicates a time delay):
  - # 10; // wait 10 ns;
  - # 50; // wait 50 ns;
- Default timescale is nanoseconds (ns)

```
module my_tb;

  reg   x,y,z;
  wire  f,g;

  m1 dut(x,y,z,f,g);

  initial
  begin

    {x,y,z} = 3'b000;
    #10;
    {x,y,z} = 3'b001;
    #25;

  end

endmodule
```

Testbench

Integer Signal Type

- To model a collection of bits representing a number, declare signals as type 'integer'
- Assigning an integer to a bit or group of bits will cause them to get the binary equivalent
- Assigning an integer value too large for the number of bits will cause just the LSB's of the number to be assigned
  - Assigning $8_{10}=1000_2$ to a 3-bit value will cause the 3-bit value to be 000 (i.e. the 3 LSB's of 1000)

```
module my_tb;

  reg     w,x,y,z;
  integer num;

  initial
    begin
      num = 15;
      {w,x,y,z} = num;
      // assigns
      // w,x,y,z = 1111
      #10;
      num = num+1;
      // num = 16
      {w,x,y,z} = num;
      // w,x,y,z = 0000
  end
endmodule
```

Testbench

For loop

- Integers can also be used as program control variables
- Verilog supports 'for' loops to repeatedly execute a statement
- Format:
  - for(initial_condition; end_condition; increment statement)

```
module my_tb;

  reg     a,b;
  integer i;

  initial
    begin
      for(i=0;i<4;i=i+1)
      begin
        {a,b} = i;
      end
    end
endmodule
```

You can't do "i++" as in C/C++ or Java

a,b = 00, then 01, then 10, then 11

Here, 'i' acts as a counter for a loop. Each time through the loop, i is incremented and then the decimal value is converted to binary and assigned to a and b

For loop

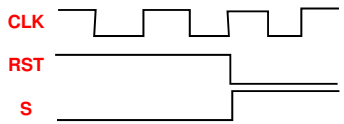
- **Question**: How much time passes between assignments to {a,b}
- **Answer**: 0 time…in fact if you look at a waveform, {a,b} will just be equal to 1,1…you'll never see any other combinations
- We must explicitly insert time delays!

```
module my_tb;

  reg     a,b;
  integer i;

  initial
    begin
      for(i=0;i<4;i=i+1)
      begin
        {a,b} = i;
        #10;
      end
    end
endmodule
```

Now, 10 nanoseconds will pass before we start the next iteration of the loop

Generating Sequential Stimulus

- Clock Generation
  - Initialize in an initial block
  - Continue toggling via an always process
- Reset generation
  - Activate in initial block
  - Deactivate after some period of time
  - Can wait for each clock edge via @(posedge clk)



Generated stimulus

```
module my_tb;
  reg     clk, rst, s;

  always #5 clk = ~clk;

  initial begin
   clk = 1; rst = 1; s=0;
   // wait 2 clocks
   @(posedge clk);
   @(posedge clk);
   rst = 0;
   s=1;
   @(posedge clk);
   s=0;
  end
endmodule
```