EEM16/CSM51A: Logic Design of Digital Systems

Verilog for Combinational Logic

Ankur Mehta
mehtank@ucla.edu
Notes attributed to
Yang, Srivastava (UCLA), Mitra & Dally (Stanford)

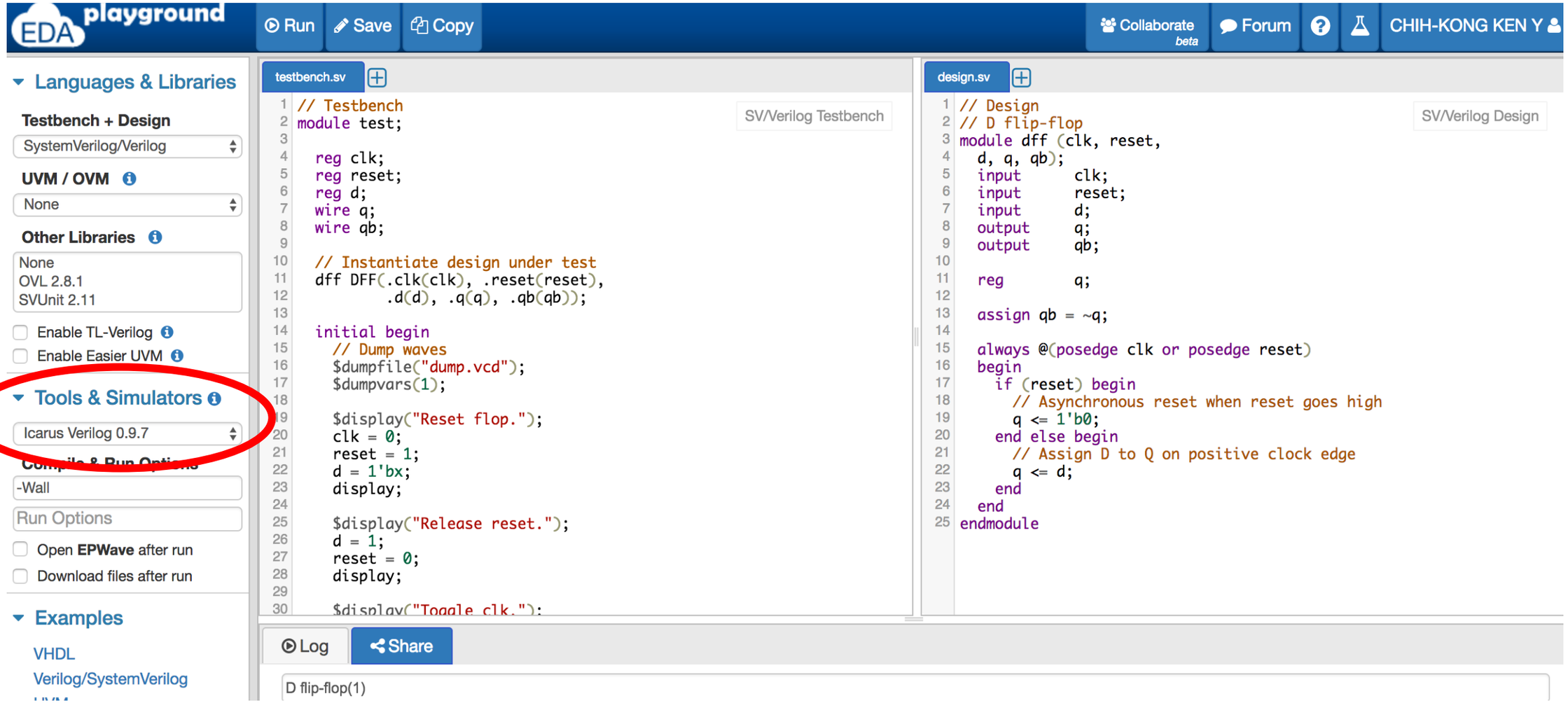
Verilog (Hardware Description Language)

- Easiest approach
  - Web interface:
    - https://www.edaplayground.com/
  - You need to create an account (in order to save your workspace)

- (Optional) Download: The version we are using is from http://iverilog.icarus.com/ v0.9.7
  - Instructions for downloading and installing for all kinds of platforms:
    - http://iverilog.wikia.com/wiki/Installation_Guide
  - For Windows, a nice executable exists:
    - http://bleyer.org/icarus/
  - Unfortunately, this packaging does not exist for Macs. I can provide a virtual machine for you to run… albeit a little slower.

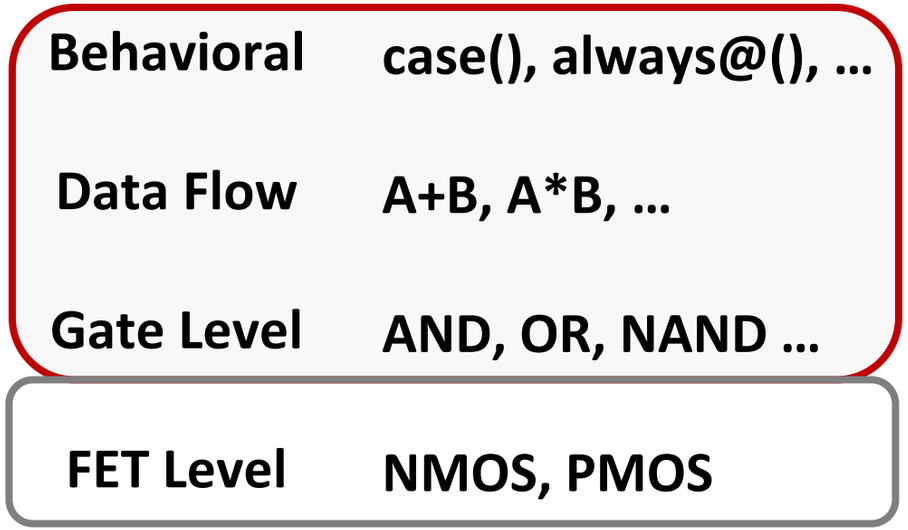
- Guides and resources: There are tons of similar guides, e.g.:
  - http://www.doulos.com/knowhow/verilog_designers_guide/

EDA Playground



What is Verilog?

- HDL: Hardware description language
  - A way to describe digital (CMOS) hardware.
- Allows abstraction
  - Describe logic in different levels of detail and expression
  - Work at a level to test out ideas
- Enables hierarchy of building blocks (nesting)
  - Complexity and implementation details of a block depends on the stage of the design
  - Can be at any abstraction level
- Execution engine to check the logical behavior of a function.
  - Checks the implementation at each level



HDL =/= high level programming language

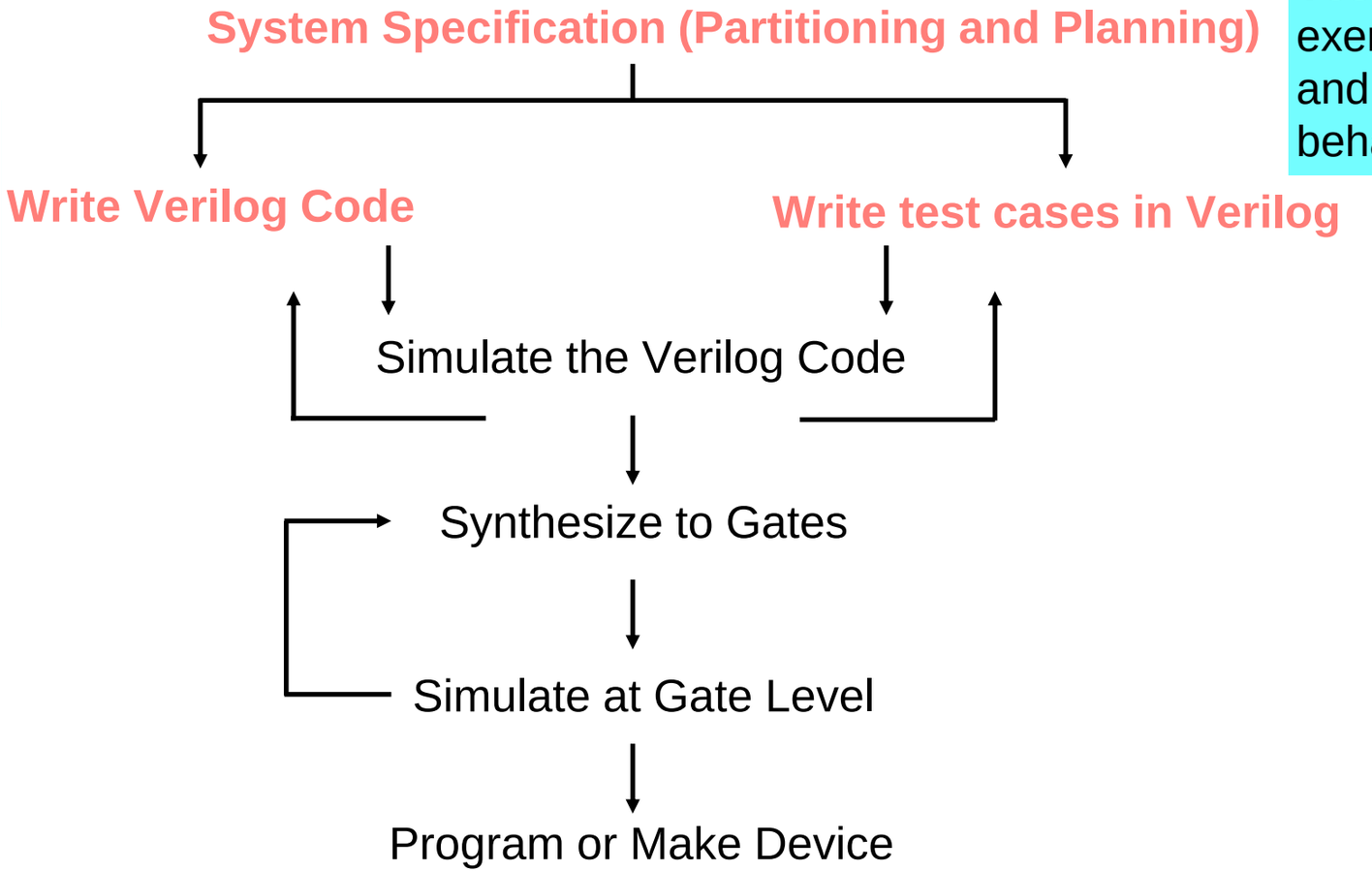
- The description of the design is physically mapped to actual hardware blocks.
  - Variables and parameters are actual wires and signals or registers that hold state
  - Only representation for variables are bits
- … with some extensions for convenience when testing / debugging
  - Loops, integers, strings, etc.
  - Only available when verifying a block.
- “Program" not sequential.
  - Modules/Blocks operate in parallel.
    - Signals communicate between modules
  - … but with support to model delays
  - … with some extensions for convenience when verifying

C / Python	Verilog
Procedures / function	Modules
Parameters	Ports
Variables	Wires / Regs

Design Flow Using Verilog

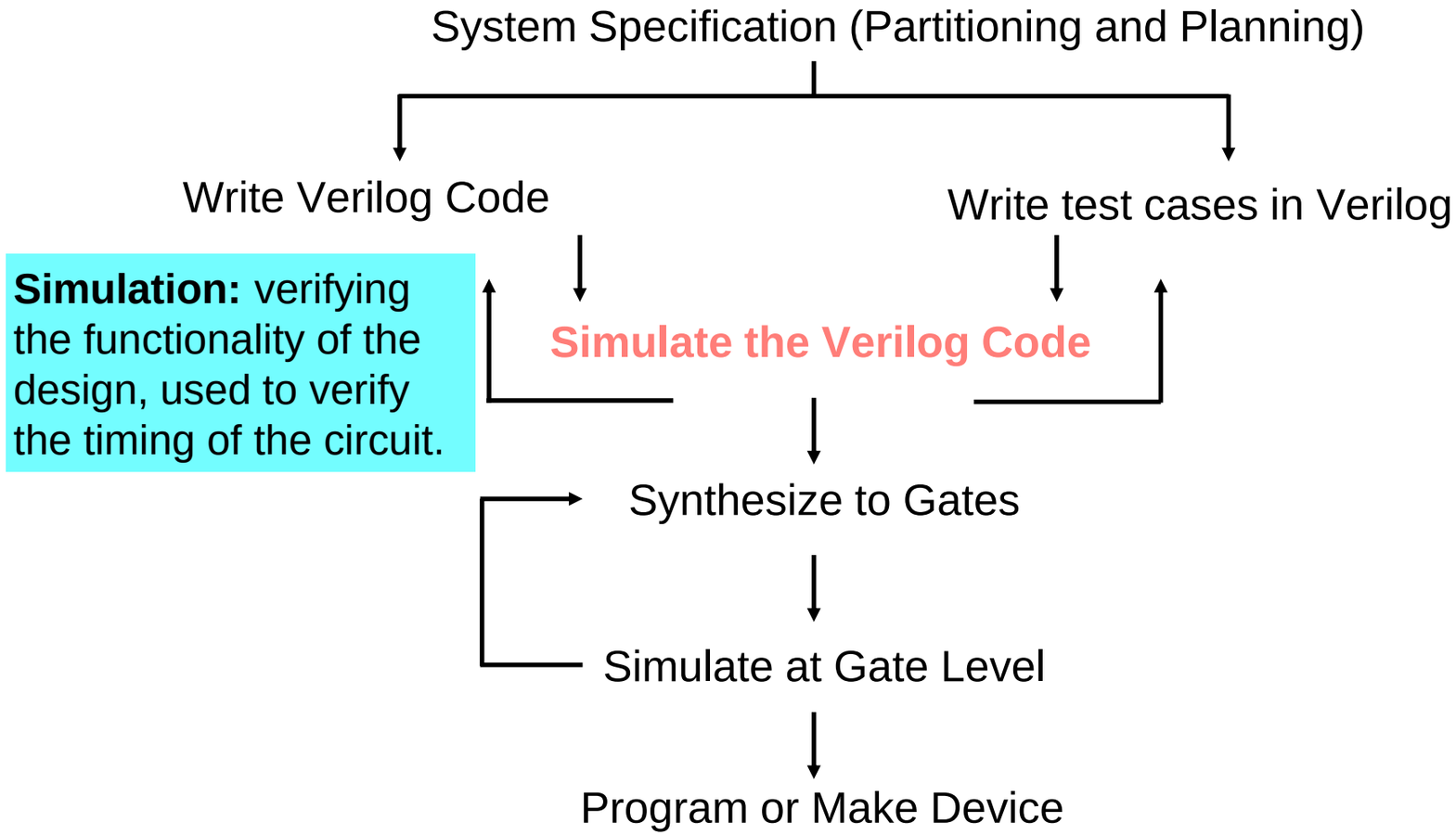
Design: Code here must be “synthesizable” (can be mapped to hardware)

TestBench: Code here exercises the Design and can be entirely behavioral.



https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

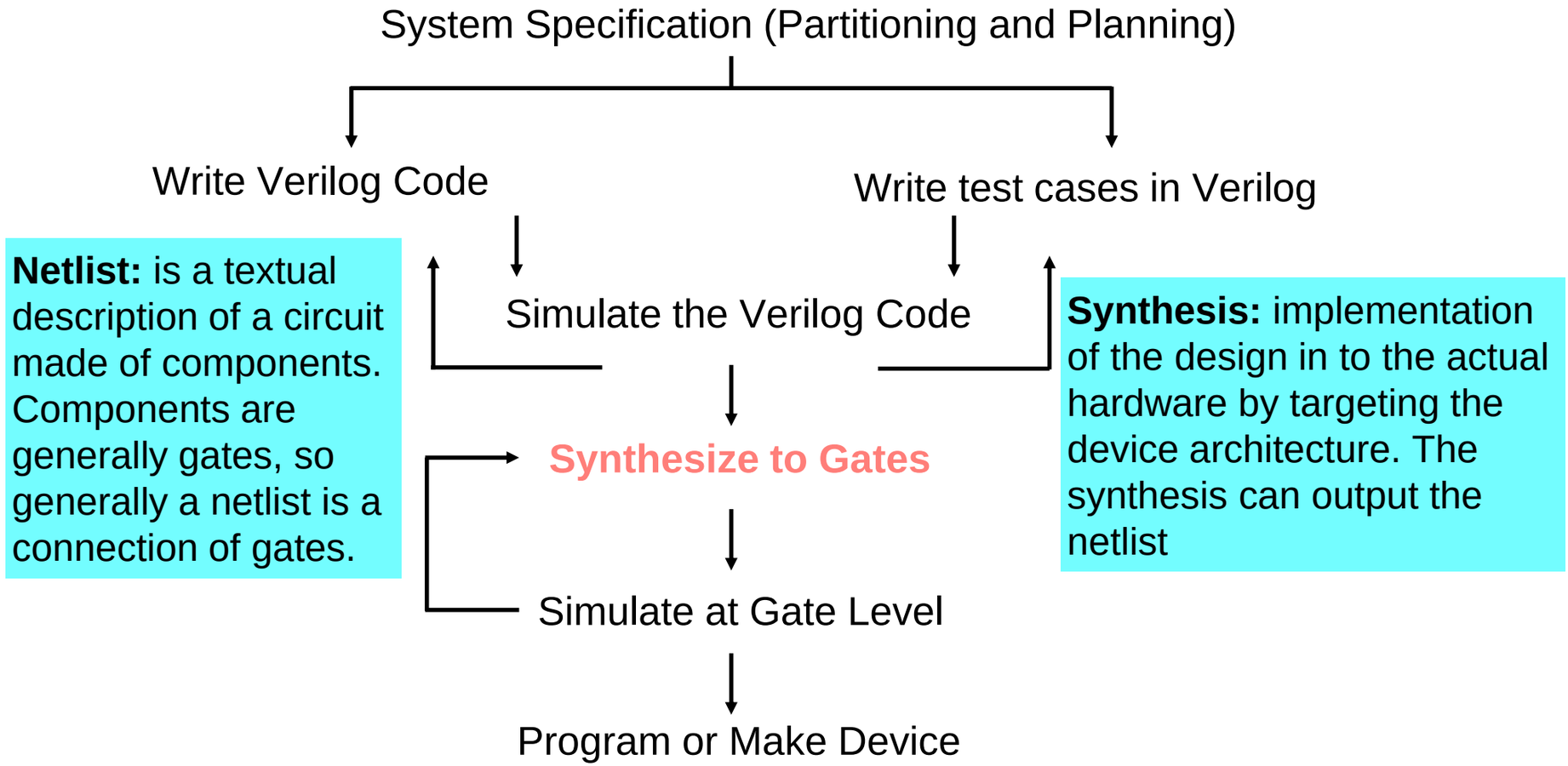
Design Flow Using Verilog



Simulation: verifying the functionality of the design, used to verify the timing of the circuit.

https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

Design Flow Using Verilog



Netlist: is a textual description of a circuit made of components. Components are generally gates, so generally a netlist is a connection of gates.

Synthesis: implementation of the design in to the actual hardware by targeting the device architecture. The synthesis can output the netlist

https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

Design Flow Using

Syster and Planning)

Write Verilog Code st cases in Verilog

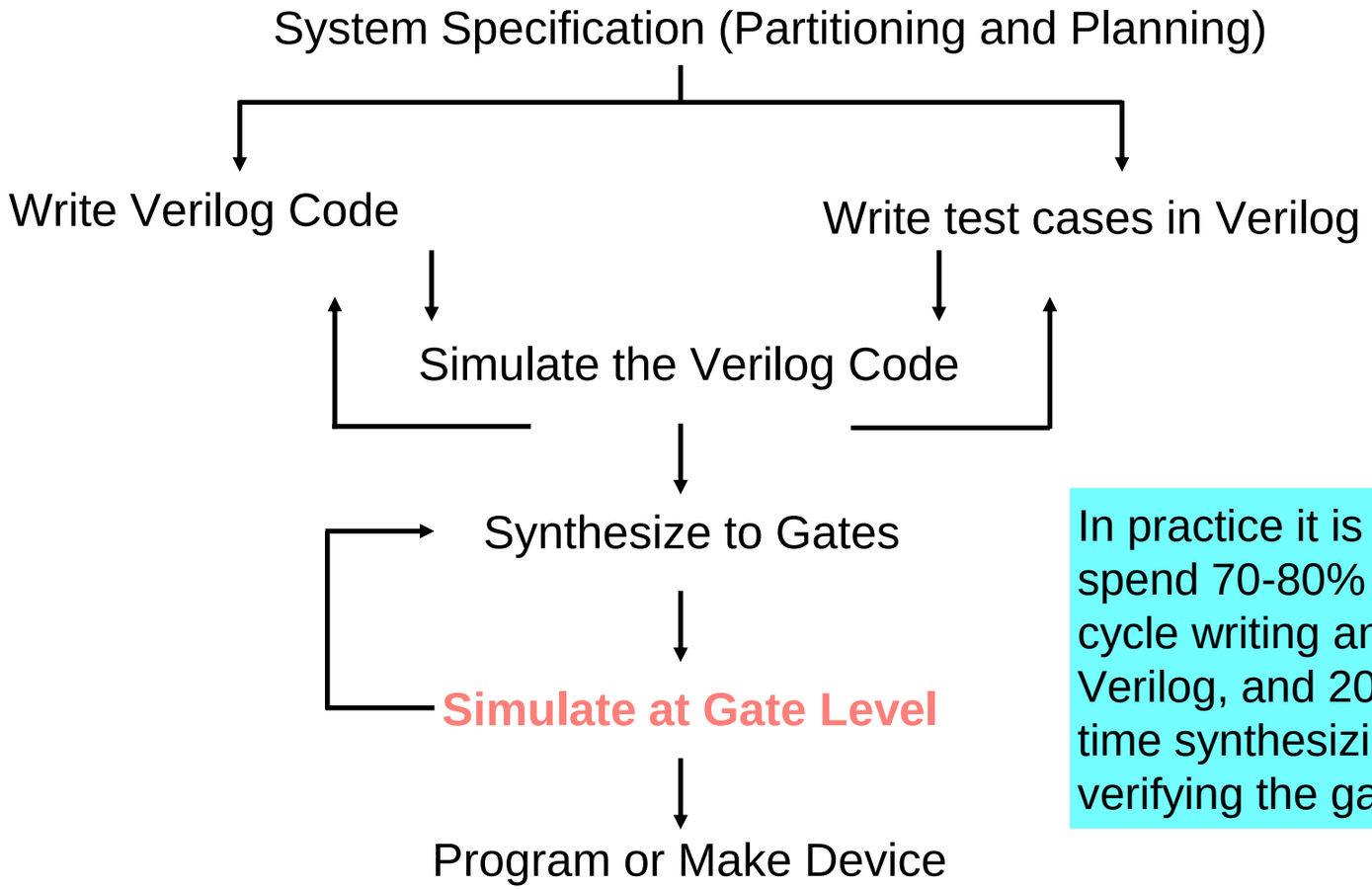
Netlist: is a textual description of a circuit made of components. Components are generally gates, so generally a netlist is a connection of gates.

Synthesis: implementation of the design in to the actual hardware by targeting the device architecture. The synthesis can output the netlist

```
module traffic_light(clk, rst, carew, lights);
 wire _000_;
 wire _001_;
 wire _002_;
 wire _003_;
 wire _004_;
 wire _005_;
 wire _006_;
 wire _007_;
 wire _008_;
 wire _009_;
 wire _010_;
 wire _011_;

 (* src = "design.sv:39" *)
 input carew;
 (* src = "design.sv:37" *)
 input clk;
 (* src = "design.sv:40" *)
 output [5:0] lights;
 (* src = "design.sv:41" *)
 wire [3:0] next;
 (* src = "design.sv:42" *)
 wire [3:0] next1;
 (* src = "design.sv:38" *)
 input rst;
 (* src = "design.sv:41" *)
 wire [3:0] state;
 NOR _047_ (
   .A(state[3]),
   .B(state[2]),
   .Y(_000_)
 );
 NOT _048_ (
   .A(state[1]),
   .Y(_001_)
 );
 NOR _049_ (
   .A(_001_),
   .B(state[0]),
   .Y(_002_)
 );
 NAND _050_ (
   .A(_002_),
   .B(_000_),
   .Y(_003_)
 );
 NOT _051_ (
   .A(state[0]),
   .Y(_004_)
 );
 NOR _052_ (
   .A(state[1]),
   .B(_004_),
   .Y(_005_)
 );
 NAND _053_ (
   .A(_005_),
   .B(_000_),
   .Y(_006_)
 );
 NAND _054_ (
   .A(_006_),
   .B(_003_),
   .Y(_007_)
 );
```

https://www.doulos.com/knowhow/verilog_designers_guide/design_flo

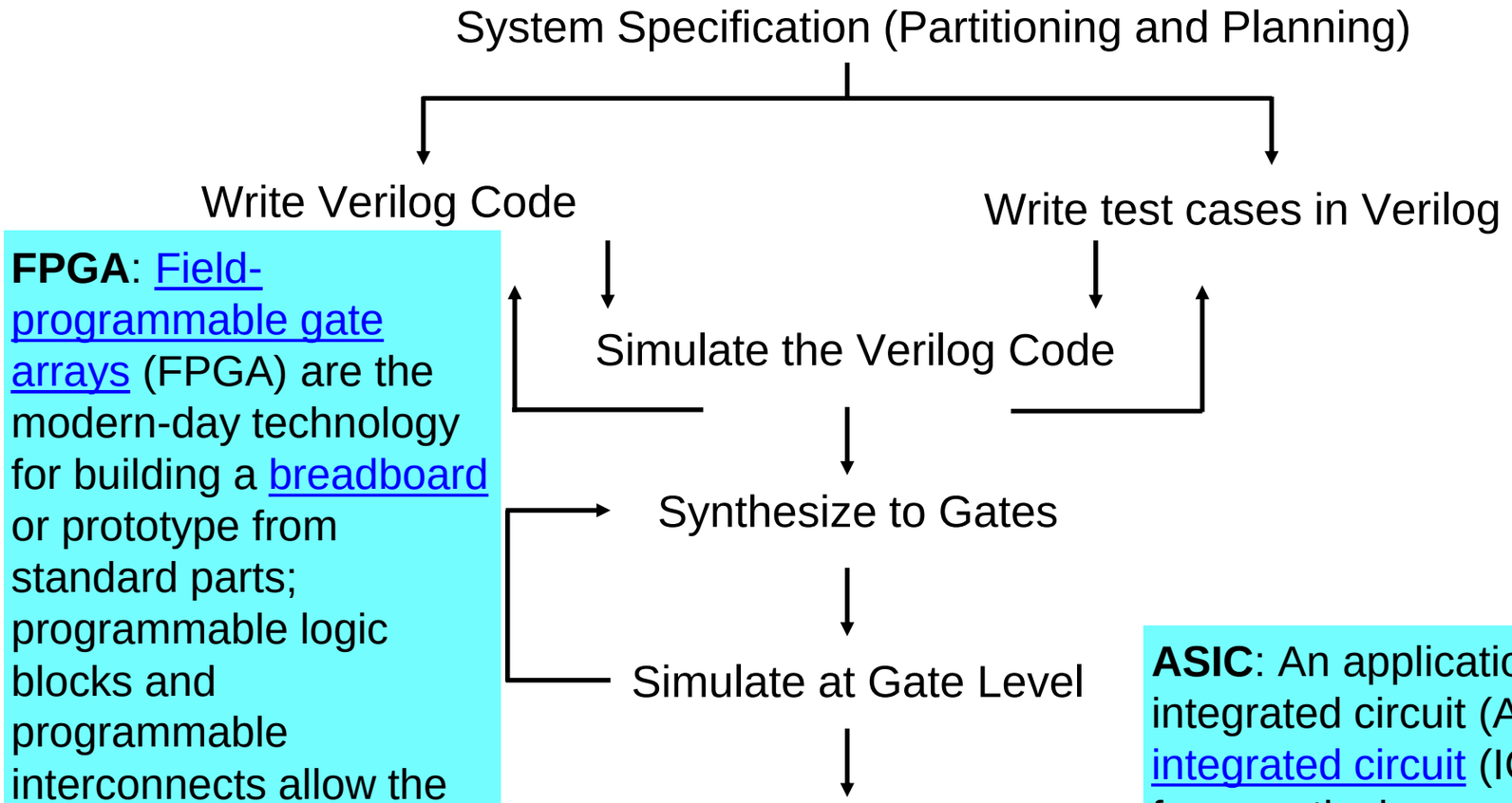
Design Flow Using Verilog



In practice it is common to spend 70-80% of the design cycle writing and simulating Verilog, and 20-30% of the time synthesizing and verifying the gates.

https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

Design Flow Using Verilog

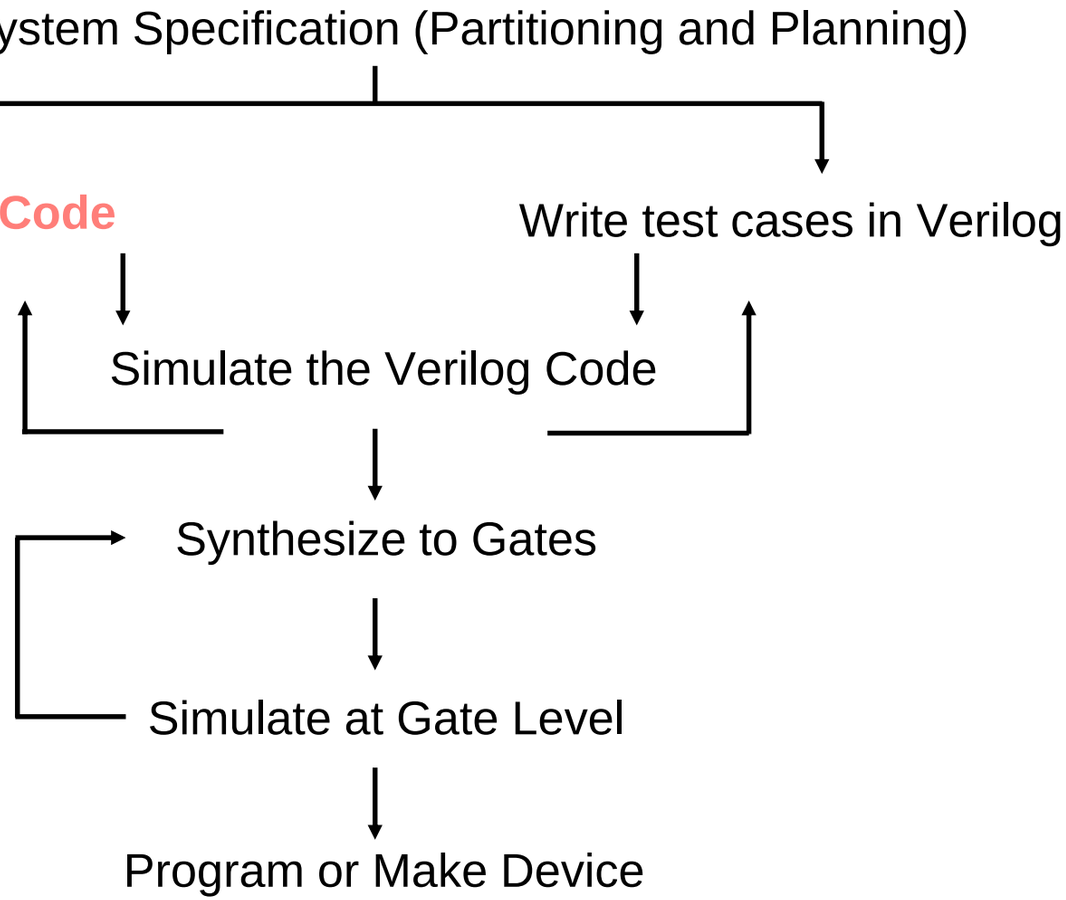


FPGA: Field-programmable gate arrays (FPGA) are the modern-day technology for building a breadboard or prototype from standard parts; programmable logic blocks and programmable interconnects allow the same FPGA to be used in many different applications.

ASIC: An application-specific integrated circuit (ASIC) is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use.

https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

Design Flow Using Verilog



RTL: **register-transfer level** is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals

https://www.doulos.com/knowhow/verilog_designers_guide/design_flow_using_verilog/

Modules – Basic Building Block

Keyword:
module
endmodule

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

endmodule
```

Modules – Basic Building Block

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class

endmodule
```

Ports

Declare the port variables
Keywords: input, output, inout

Modules – Basic Building Block

Compile time parameters Keyword: parameter

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

parameter width = 8; parameter len = 2 ;
//parameters are variable replacement during compilation

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class









endmodule
```

Modules – Basic Building Block

Two ways for comments Single/multi-lines

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

parameter width = 8; parameter len = 2 ;
//parameters are variable replacement during compilation

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class




/*
   Multi-line comment
   begin code here to describe the logic
   end code
*/
endmodule
```

Modules – Basic Building Block

Internal signals: wires and registers

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

parameter width = 8; parameter len = 2 ;
//parameters are variable replacement during compilation

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class

// instantiate any internal signals
wire [width-1:0] int_data;  //vector of wires for communicating
reg [width-1:0] int_reg [0:7]; //an array of vectors – only for reg
/*
   Multi-line comment
   begin code here to describe the logic
   end code
*/
endmodule
```

- Keyword “**wire**”
  - Connecting element between modules.
  - input/outputs are basically wires
  - Can assign logic (no state)
- Keyword “**reg**”
  - Stateful variable (for storage)
  - Can be used for pure combinational logic or sequential logic

Modules – Basic Building Block

Vectors or busses [high:low] bit-position

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

parameter width = 8; parameter len = 2 ;
//parameters are variable replacement during compilation

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class

// instantiate any internal signals
wire [width-1:0] int_data;  //vector of wires for communicatin
reg [width-1:0] int_reg [0:7]; //an array of vectors – only fo
/*
   Multi-line comment
   begin code here to describe the logic
   end code
*/
endmodule
```

Array of 8 registers [low:high] Only for reg

- Keyword “**wire**”
  - Connecting element between modules.
  - input/outputs are basically wires
  - Can assign logic (no state)
- Keyword “**reg**”
  - Stateful variable (for storage)
  - Can be used for pure combinational logic or sequential logic

Modules – Basic Building Block

```
module BuildingBlock (clk, enable, rx_data, tx_data, io_data);

parameter width = 8; parameter len = 2 ;
//parameters are variable replacement during compilation

input clk, enable ; //in-line comments – use liberally
input [width-1:0] rx_data;
output [width-1:0] tx_data ;
inout [width-1:0] io_data ; //not used much in this class

// instantiate any internal signals
wire [width-1:0] int_data;  //vector of wires for communicating
reg [width-1:0] int_reg [0:7]; //an array of vectors – only for reg
/*
   Multi-line comment
   begin code here to describe the logic
   end code
*/
endmodule

module test;
//Instantiate
reg [7:0] rx1, BB8_data, tx, io;
reg en; //either stateful register or logic signal

//longer form and ordering doesn't matter
BuildingBlock BB8(.clk(gclk), .enable(en),
                  .rx_data(rx1), .tx_data(BB8_data),
                  .io_data(io));
//shorter but ordering matters
BuildingBlock BB10( gclk, en, BB8_data, tx, io),
endmodule
```

Two ways to instantiate
1) passes in overriding parameters (in order)
2) port ordering matters

Signals, Numbers and Variables

- Numbers are represented as <length>’<base><value>
  - Examples: 8’b00010111, 2’hFF, 4’d1234, 8’b0001_0111 (“_” improves readability)
  - High impedance “z” (“?” is accepted in Verilog)
    - not driven by an output
    - default starting value of nodes
  - Undefined – “x”
    - not sure which value it is (usually because 2 nodes are driving it and conflicting)
    - Good to identify this for debugging

- Port/Variable naming
  - Try to choose meaningful names.
  - Lots of different naming standards.
    - Example: <name>_i are inputs, <name>_o are outputs, <name>_w are wires, etc.
  - Account for assertion (unHappy_L for assert low… 0=true)

Ways to Describe A Function/Block

- Structural
  - Consists only of module calls
- Declarative
  - Concurrently executed combinational logic
- Procedural
  - Sequentially executed program
    - A state machine (with storage) … More later…
    - Or combinational logic
- ~~Functional/Task~~
  - ~~Function calls~~
  - ~~Not mapped to hardware so we ignore this~~

Structural Description

```
module system;
   wire [7:0] bus_v1, const_s1;
   wire [2:0] regSpec_s1, regSpecA_s1,
    regSpecB_s1;
   wire [1:0] opcode_s1;
   wire Phi1, Phi2, writeReg_s1,
    ReadReg_s1,nextVector_s1
   clkgen clkgen1(Phi1, Phi2);
   datapath datapath1(Phi1, Phi2,
    regSpec_s1, bus_v1,    writeReg_s1,
   readReg_s1);
   controller controller1(Phi1, Phi2,
    regSpec_s1, bus_v1, const_s1,
    writeReg_s1, readReg_s1,
    opcode_s1, regSpecA_s1,
   regSpecB_s1,    nextVector_s1);
   patternsource patternsource1(Phi1,
    Phi2,nextVector_s1, opcode_s1,
    regSpecA_s1, regSpecB_s1,
   const_s1);
endmodule
```

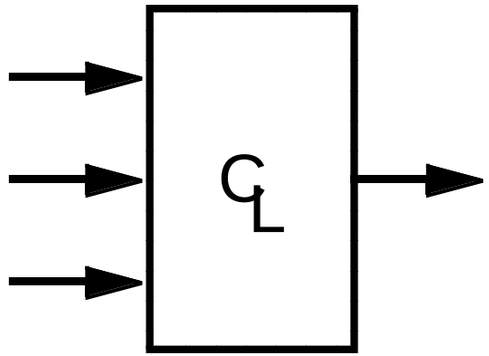
- Lowest level of abstraction
- Compose a module out of module calls.
  - Specify components, and wiring
- Maps a physical structure into Verilog.
  - One example to the right
- Hierarchical
  - List of modules/blocks
  - List of sub-blocks.
  - List of gates
  - List of transistors.
- Typically don’t go below a gate level list.
  - Only near the end of a design.
  - We’ll use it to implement functions as logic gates for the purposes of this class.

Declarative Statements

- Provides the logical relations between inputs and outputs.
- Assign outputs to be some function of the inputs (*continuously*)
  - Denoted by the key word is ***assign***
- Uses a C-like expression syntax
- Examples (*all execute in parallel*):

```
wire          nor, a, o, out;
wire  [4:0]   sum;
…
assign        nor = ~(b | c);
assign        a = x & y, o = x | y;
assign        sum[4:0] = a[3:0] + b[3:0];
assign        out = (sel) ? in1: in2; //conditional
```

assign models wires to be outputs of combinational logical elements



- Outputs are wires, and can be a single bit or multiple bits.
  - It is good practice to declare all variables even though Verilog allows undeclared single bit wires.

Declarative Order of Execution

- All assigns can be presumed to happen simultaneously
  - Each statement is occurring concurrently.
    ```
    assign      x = a;
    assign      a = b;
    ```
  - In C, the order of the above statements matters, but not in Verilog.
    - Above example, x = b no matter what because both statements are wires shorting those signals.
  - Consider the wiring
    ```
    assign      out = a;
    assign      out = b;
    ```
    - This yields a warning and is not allowed.

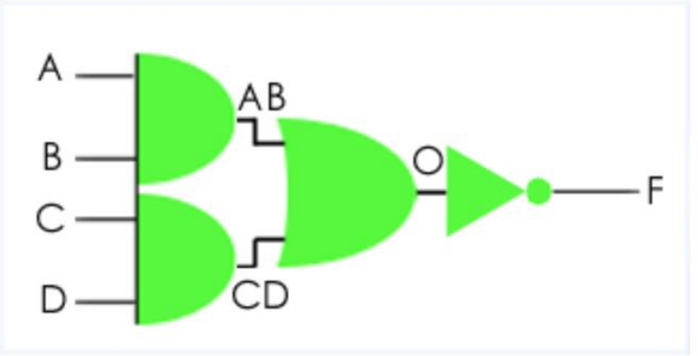
Example

- In In this module body, there are four continuous assignment statements.

- And-Or-Invert gate.
  - Note the alternate module declaration

```
module AOI (
    input A, B, C, D,
    output       F
    );
```

- The assign statements are independent and executed concurrently.
  - ***Non-Blocking assignment***
  - Any change of any of the inputs causes a re-evaluation of the statement.




```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  wire F;  // the default
  wire AB, CD, O;  // necessary

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
endmodule
// end of Verilog code
```

https://www.doulos.com/knowhow/verilog_designers_guide/

Shifting and Concatenation

- Shifting:
  - Let x[7:0] = 8'b0101_1101
    (Right shift by 1) X>>1 0010_1110
    (Left shift by 2) X<<2 0111_0100

Fills with 0's and not wrap around

- Concatenation
  - Format: {operand, … , operand} or {(#){operand}}
  - Let A=3'b111, B=3'b010, C=3'b011, D=3'b100, E=8'b1010_1010

{B,C}	6'b010001
{A,B,3'b001}	9'b111010001
{A,B[0],C[1]}	5'b11101
{(3){D[2]}	3'b111
{{(3){E[7]},E[7:2]}	8'b1111_0101

Copies operand # times

A shift (>>4) that extends the MSB.
Sign extension!
A straight >> would not be correct.

  - Outputs must be sized to the right vector length

Operators

Operator Type	Operator Symbol	Operation Performed
Arithmetic	*	multiply
	/	divide
	+	add
	-	subtract
	%	modulus
	+	unary plus
	-	unary minus
Logical	!	logical negation
	&&	logical and
	||	logical or
Equality	==	equality
	!=	inequality
	===	equality w/ {x,z}
	!===	inequality w/ {x,z}

Not synthesizable

Operator Type	Operator Symbol	Operation Performed
Bit-wise	~	bitwise negation
	& (|)	bitwise and (or)
	^ (^~ or ~^)	bitwise ex-or (ex-nor)
Reduction	& (|)	reduction and (or)
	~& (~|)	reduction nand (nor)
	^	reduction ex-or
	^~ or ~^	reduction ex-nor
Relational	> (<)	greater (less) than
	>=	greater than or equal
	<=	less than or equal
Concatenation	{ }	concatenation
Conditional	<cond>?<>:<else>	conditional
Shift	>>	right shift
	<<	left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift

Operates on all bits of a vector

w/ sign extension may be non-synthesizable

Procedural Statements

- Flow control (implies sequential ordering)
  - Denoted with keyword ***always*** or ***initial*** provides functionality of a tiny program that executes sequentially.
- “initial”
  - Purely behavioral for test bench
  - The sequence that is run only at the **beginning** of execution.
- “always”
  - Can sometimes be synthesized.
  - Each always block is run concurrently and executed whenever it is activated.
  - always @(<activation list>)

Procedural Statements

- Inside an ***initial*** or ***always*** block, can use standard control flow statements:

```
if (<conditional>) then
    <statements>; //statements can be compound by using begin <statements>; end
else
    <statements>;

case (<var>)
    <value1>: <statements>;
    <value2>: <statements>;
    …
    default: <statements>
```

Procedural Statements

- Ones that are purely for behavioral modeling and not synthesizable

```
for (i = 0; i < 256; i = i + 1) begin
      <statements>;
end

while (<conditional>) begin
end

repeat (<number>) begin
end
```

Always example

- Example:

```
reg out;
always @ (reset)
    begin
        // more than 1 statement allowed inside here
        if (x==y) then
                out = in1;
        else
                out = in2;
        end
```

Output must be a reg

Inputs can be either wires or reg

`Always` Block For Combinational Logic

Let's make an "always" block implement a combinational logic

- No unset outputs
  - Are all outputs given a value with an explicit assignment statement at the end of the block?
    - If not, then it is unset.
  - If the output is ***always set***, then the always block **forms combinational logic**.
  - If you leave it unset, it implies the previous value => storage
- Make it activate for all input transitions.
  - ***Activation list determines when to execute the block***
  - So, make a complete list of all conditional and RHS variables (inputs) in the activation list

```
always @ (x or y or in1 or in2)
    begin
        // more than 1 statement allowed inside here
        if (x==y) then
                out= in1;
        else
                out = in2;
        end
```

More About Activation Lists

- Tells the simulator when to run this block
  - Allows the user to specify when to run the block and makes the simulator more efficient.
  - But also enables subtle errors to enter into the design.
- Two forms of activation list in Verilog:
  - @(signalName1 or signalName2 or …)
    - Evaluate this block when any of the named signals change (either positive or negative change)
  - @(posedge signalName1) or @(negedge signalName2)
    - Makes something trigger on the transition of a signal (register).
    - Evaluates only on one edge of a signal.
    - Can have @(posedge signal1 **or** posedge signal2)
      - Edges are singular events so must be “or”
      - Don’t use because difficult to map to an actual gate
- You can use always @(*)
  - All inputs considered (short-cut)
  - Use comma’s instead of or’s @(signalName1, signalName2, …)
- You cannot use always without an activation list.
  - Leads to an infinite loop

More Examples

- *If* statement
- What happens if no *else*?
  - reg f; //means that *f* can keep state.
  - *f* stays the value of a even when *sel* changes to 0.
- Case statement
  - Helps with multiple input multiplexer.
  - Selecting Finite State Machine state transitions.
- *default* covers the condition when not all cases are covered on the LHS.
  - Otherwise, keeps previous value… not combinational logic any more.
  - Example uses it to flag when not all cases are covered.

```
reg f;
always @(sel or a or b)
begin
  if (sel == 1)
    f = a;
  else
    f = b;
end
```

```
module mux (a,b,c,d,sel,y);
input a, b, c, d;
input [1:0] sel;
output y;

reg y;

always @ (a or b or c or d or sel)
case (sel)
  0 : y = a;
  1 : y = b;
  2 : y = c;
  3 : y = d;
  default : $display("Error in SEL");
endcase

endmodule
```

Different *Case*s

- Normal *case* statements can handle x's and z's
  - Good for debugging to flag these situations
  - Statement3 can be a $display of error message
- May need to spell out don't care
  - Use "?" or "z"
  - *casez* instead of *case*
  - casez also allows us to keep going despite z's (not realistic)
- May want to keep going despite x's
  - *casex* instead of *case*
  - Treat x's as also don't cares (stat2 selected if mask=11001100)
  - Not as safe for logic synthesis since x's may indicate errors...

```
reg [7:0] r, mask;
          •
          •
          •
mask = 8'bx0x0x0x0;
casex (r ^ mask)
   8'b001100xx: stat1;
   8'b1100xx00: stat2;
   8'b00xx0011: stat3;
   8'bxx001100: stat4;
endcase
```

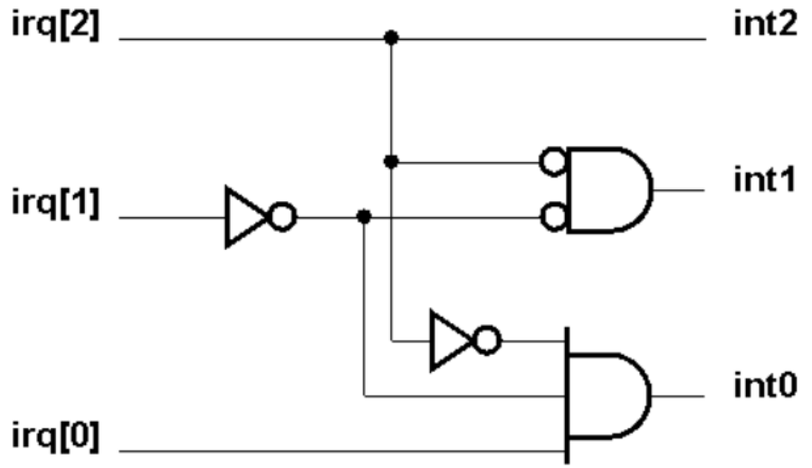
```
case (select[1:2])
    2'b00: result = 0;
    2'b01: result = flaga;
    2'b0x,
    2'b0z: result = flaga ? 'bx : 0;
    2'b10: result = flagb;
    2'bx0,
    2'bz0: result = flagb ? 'bx : 0;
    default: result = 'bx;
endcase
```

```
reg [7:0] ir;
          •
          •
          •
casez (ir)
   8'b1???????: instruction1(ir);
   8'b01??????: instruction2(ir);
   8'b00010???: instruction3(ir);
   8'b000001??: instruction4(ir);
endcase
```

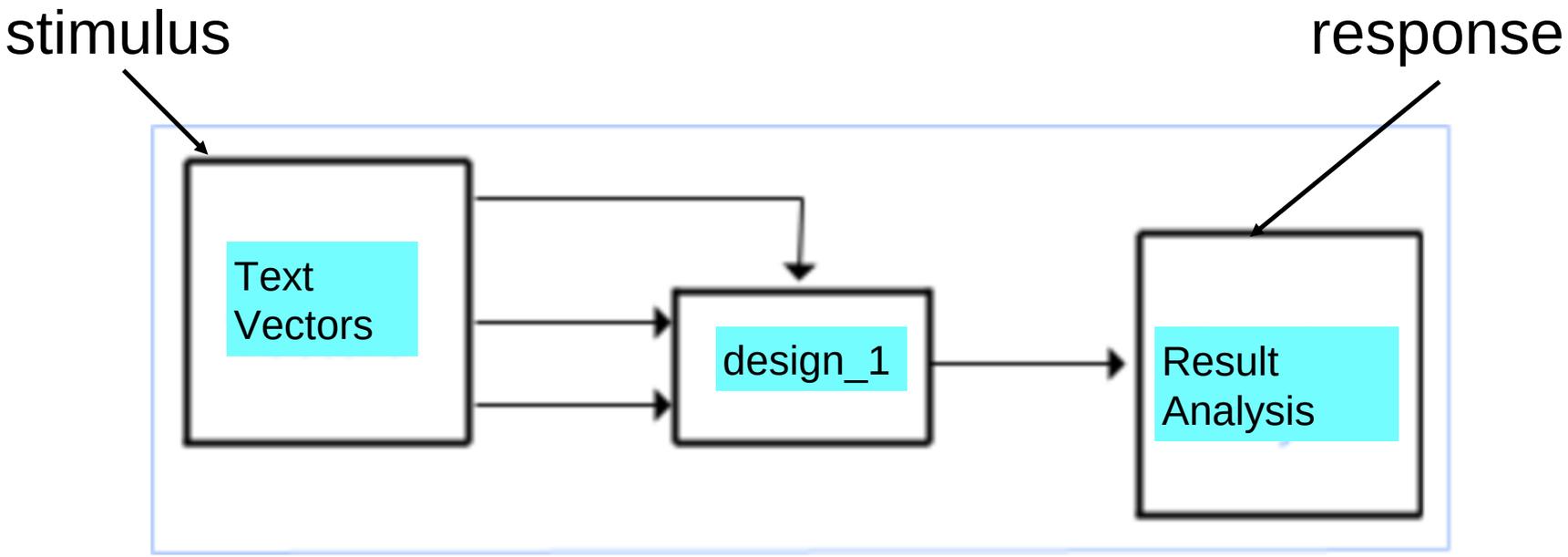
Care with *Case*s

- Case statements are actually prioritized
  - The 2nd case entry doesn't happen unless after the 1st not match.
  - May not be what the actual hardware implies – especially when cases are mutually exclusive.
- Full case
  - All binary possibilities of the selection is covered in the case statements. Otherwise, include a default.
  - Desirable for most combinational logic implementation.
  - Avoid storage of old value (***unless you intentionally want to build a storage element)***
- Parallel case
  - Only one case is triggered for any change in the selection.
  - Again, desirable for most combinational logic implementation.
  - Exception is **priority encoder**. ***You intentionally want to select the first case that hits***.

```
module intctl1a
  (output reg       int2, int1, int0,
   input      [2:0] irq                );

  always @* begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```



Building Test Benches



- Test benches help you to verify that a design is correct.
- You can use things in test bench that are not synthesizable because the code of the test bench is not designed to be downloaded on hardware

https://www.doulos.com/knowhow/verilog_designers_guide/

Test Benches

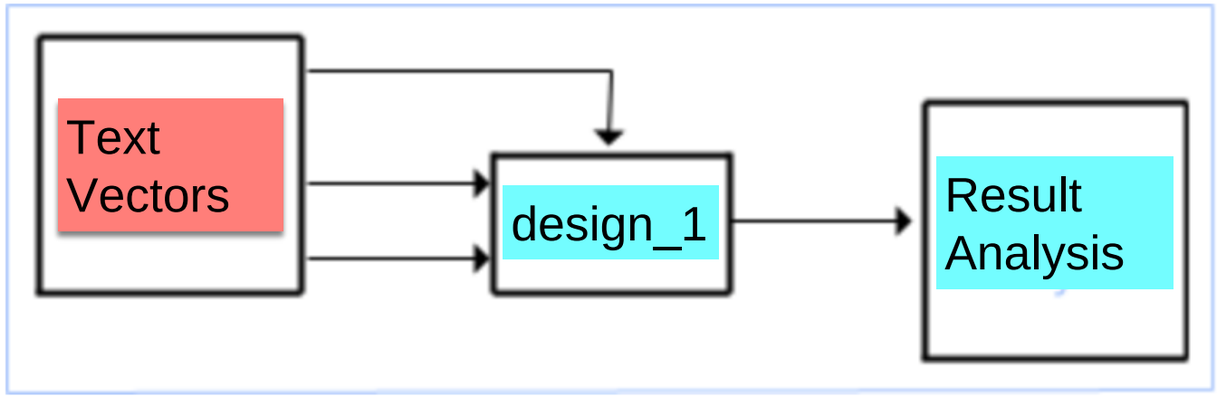
```
module design_1_test    // No ports!

  initial
  // Stimulus
  ...

design_1 M (SEL, A, B, F);

  initial
  // Analysis
  ...

endmodule
```

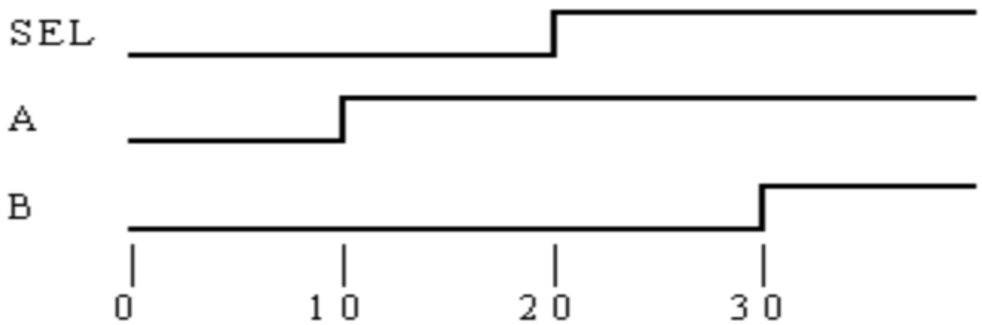
- Recall the “initial” statement
  - They are not synthesizable
- The stimulus and the response can be coded using a pair of initial blocks or combined in a single initial block.
  - An initial block (like always blocks) contains sequential statements that describe the behavior of signals in a test bench.
  - It’s ok to use non-synthesizable looping structures
- In the stimulus, we need to generate waveform on the A, B and SEL inputs

https://www.doulos.com/knowhow/verilog_designers_guide/

Test Benches – Stimulus

```
initial  // Stimulus
begin
  SEL = 0; A = 0; B = 0;
  #10 A = 1;
  #10 SEL = 1;
  #10 B = 1;
end
```





- SEL = 0; A = 0; B = 0;
  - Contains three sequential statements. SEL is set to 0, then A, then B. All three are set to 0 at time = 0.
- #10 A = 1; #10 SEL = 1; #10 B = 1;
  - The notation is to force the simulator to advance in time for a Test Bench to execute a sequence of events.
  - 10 time units after A is set to 1, SEL is set to 1.
  - Another 10 time units later (so we are now at simulation time = 30 time units), B is set to 1.

https://www.doulos.com/knowhow/verilog_designers_guide/

A Note on Delays in Verilog

- Verilog simulated time is in “units” or “ticks”.
- `timescale 1ns/10ps defines the ticks to be 1ns (the internal time step is 1/100th)
- Simulated time
  - Unrelated to the wall-clock to run the simulator.
  - Models the time in the modeled machine
    - When the computer completes with all the “events” that occur at the current simulated time
    - The computer increases time until another signal is scheduled to change values.
- User must specify delay values explicitly to Verilog
  - # delayAmount (not synthesizable)
    - When the simulator sees this symbol, it stops “evaluating” the particular expression and schedules the delayAmount of simulated time (# of ticks) to enact the change in the signal.
    - Delays are used to model the delay in functional units (in synthesized designs)
  - Can be tricky to use properly
- The standard cell library contains annotate delay information.
  - Synthesized results can have realistic delay simulation

Delay Control

- Can be tricky so don't use it except in Test Bench
- Declarative – "*Delayed Assignment*"
  - Make out a delayed version of the input (by 10 ticks) in an assign or wire statement.
    assign #100 out = in;
  - Anywhere else to put delay is not allowed

- Procedural – "*Delayed Evaluate*"

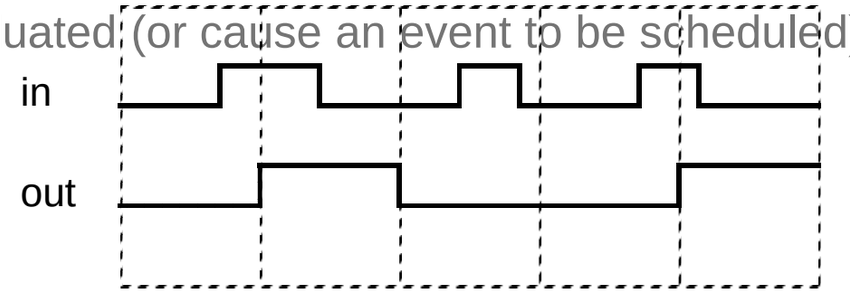
```
always @(phi or in)
        #100 if (phi) then out = in;
```

  - Wait 100 ticks after either input changes, then checks to see if phi == 1, and then updates the output.
  - Example – clock source

```
always
        #100 out = ~out;
```

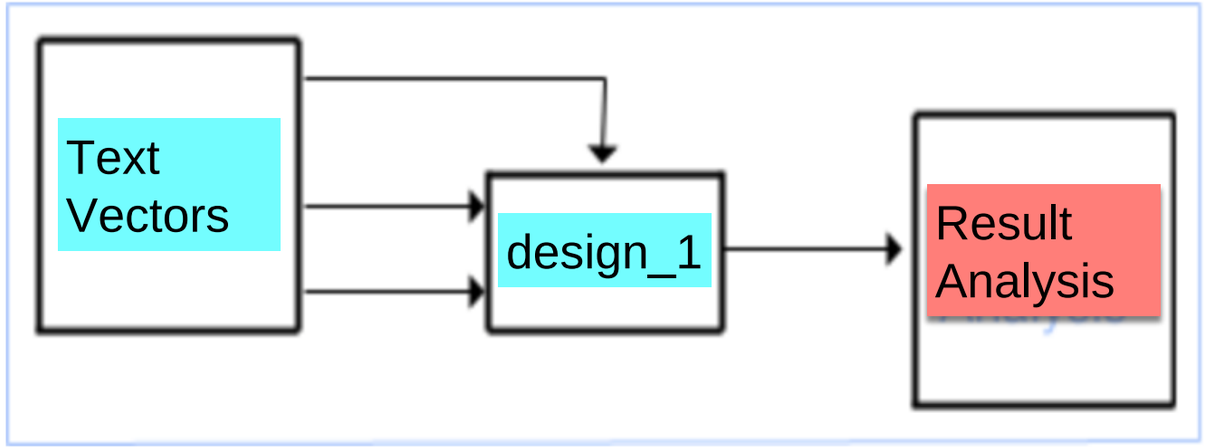
  - An always block is not reactivated until every line of code is evaluated (or cause an event to be scheduled).
    - So while waiting for the delayed event, new inputs are ignored.

```
always
        #100 out = in;
```



Test Benches – Responses

```
initial  // Response
  $monitor($time, , SEL, A, B, F);
```



- $monitor is a system task that is part of the Verilog language.
  - Print values to the screen corresponding to the arguments that you pass to the task when it is executed.
  - The $monitor task is executed whenever any one of its arguments changes.
- $time is a system function. It returns the current simulation time.
  - In the above example, $time is an argument to $monitor.
  - However, $time changing does not cause $monitor to execute - $monitor does not print to the screen the values of all of the arguments every time the simulation time changed.
- $display(“test\n”) is a function that prints the words “test” when executes.
- Waveform viewers are also available which can be easier.

```
 0 0000
10 0101
20 1100
30 1111
```

https://www.doulos.com/knowhow/verilog_designers_guide/

Example of Test Bench

```
`timescale 1ns/10ps // specify system timescale
`define TCK 2 // define 2ns clock period
module TestClkAdder ; // name of testbench
reg [3:0] A_i, B_i ; reg CLK ; wire [3:0] S_o ; integer i ; // declare variable
CLKADD U00(.SUM(S_o), .AI(A_i), .BI(B_i),.CLK(CLK)) ; // instantiate design
reg [7:0] TestVec [0:511] ; // declare an array to store 512 8-bit test data
initial CLK = 0 ; always #(0.5*`TCK) CLK = ~CLK ; // generate clock
initial begin // start simulation
 $readmemb("TestData.txt", TestVec) ; // read data from external .txt file
  $display("*** Clocked-Adder Simulation Starts *** \n") ;
  $monitor("Time=%g, S=%d, A=%d, B=%d", $time, S_o, A_i, B_i);
  for(i=0; i<512; i=i+1) // feed data into design for 512 cycles
     #(`TCK+0.2) {A_i,B_i} = TestVec[i] ;
  #(`TCK) $finish ;  // finish the simulation
end
endmodule
```

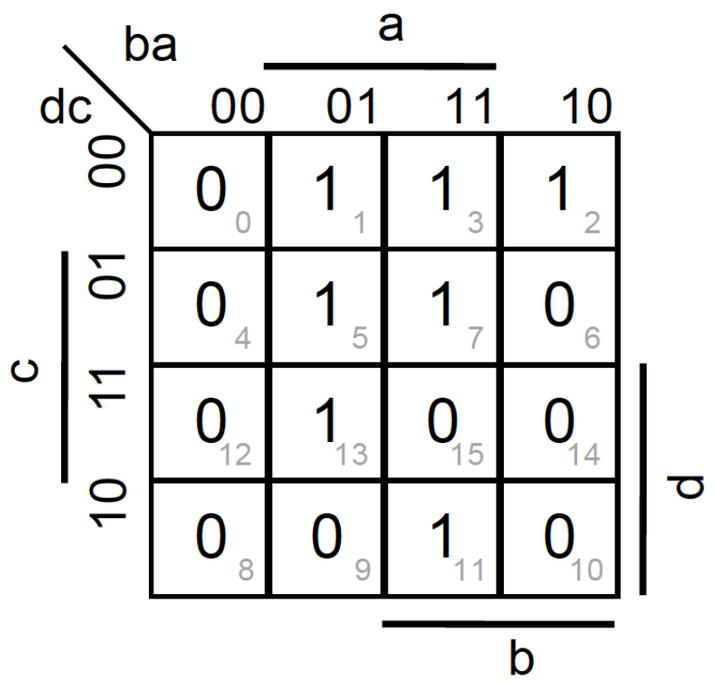
`timescale – defines a "tick" to 1ns

`define – for global variables

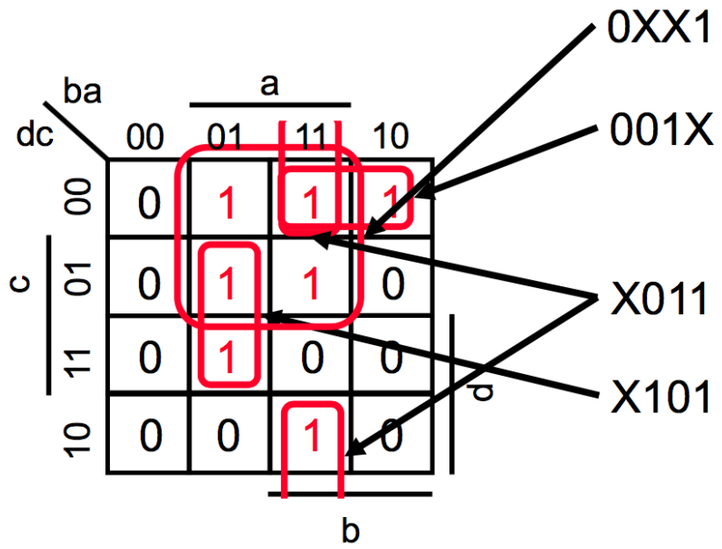
Produces 010101… indefinitely

Reads a text file into an array

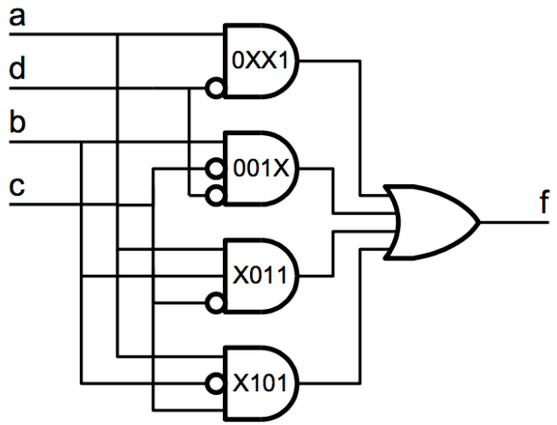
Example From Lecture – 4-bit Prime-or-1

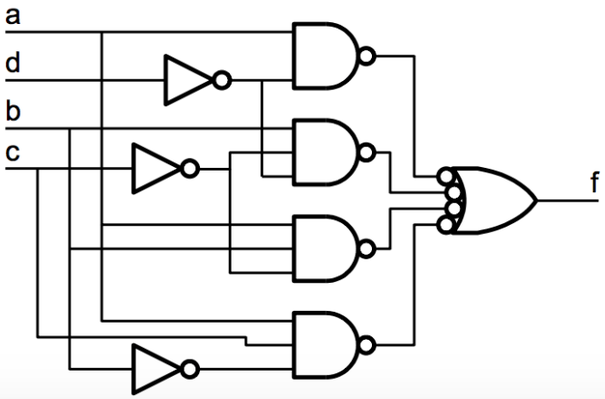


Example from Lecture – 4-bit Prime



dcba
0xx1
001x
x101
x011





Example From Lecture – 4-bit Prime

Using case

```
module prime(in, isprime) ;
  input [3:0] in ;                  // 4-bit input
  output      isprime ; // true if input is prime
  reg         isprime ;

  always @(in) begin
    case(in)
      1,2,3,5,7,11,13: isprime = 1'b1 ;
      default:         isprime = 1'b0 ;
    endcase
  end
endmodule
```

Here you described what you want, not how to design it

Example From Lecture – 4-bit Prime

Using casex

```
module prime1(in, isprime) ;
  input [3:0] in ;              // 4-bit input
  output      isprime ; // true if input is prime
  reg         isprime ;

  always @(in) begin
    casex(in)
      4'b0xx1: isprime = 1 ;
      4'b001x: isprime = 1 ;
      4'bx011: isprime = 1 ;
      4'bx101: isprime = 1 ;
      default: isprime = 0 ;
    endcase
  end
endmodule
```

Example From Lecture – 4-bit Prime

Using assign

```
module prime(in, isprime) ;
  input [3:0] in ;                  // 4-bit input
  output      isprime ; // true if input is prime

  wire isprime = (in[0] & ~in[3]) |
                 (in[1] & ~in[2] & ~in[3]) |
                 (in[0] & ~in[1] & in[2]) |
                 (in[0] & in[1] & ~in[2]) ;
endmodule
```

Example From Lecture – 4-bit Prime

Test Bench

```
module test_prime ;
  reg [3:0] in ;
  wire isprime ;

  // instantiate module to test
  prime  p0(in, isprime) ;

  initial begin
    in = 0 ;
    repeat (16) begin
      #100
      $display("in = %2d isprime = %1b",in,isprime) ;
      in = in+1 ;
    end
  end
endmodule
```

Example From Lecture – 4-bit Prime

Testing Results

```
# in =  0 isprime = 0
# in =  1 isprime = 1
# in =  2 isprime = 1
# in =  3 isprime = 1
# in =  4 isprime = 0
# in =  5 isprime = 1
# in =  6 isprime = 0
# in =  7 isprime = 1
# in =  8 isprime = 0
# in =  9 isprime = 0
# in = 10 isprime = 0
# in = 11 isprime = 1
# in = 12 isprime = 0
# in = 13 isprime = 1
# in = 14 isprime = 0
# in = 15 isprime = 0
```

Summary

References

- Doulos Verilog:
  - https://www.doulos.com/knowhow/verilog_designers_guide/
- EDAPlayground:
  - http://www.edaplayground.com