ECE/ENGRD 2300
Digital Logic and Computer Organization

Introduction to FPGAs
& the Verilog HDL

Spring 2024

Cornell University

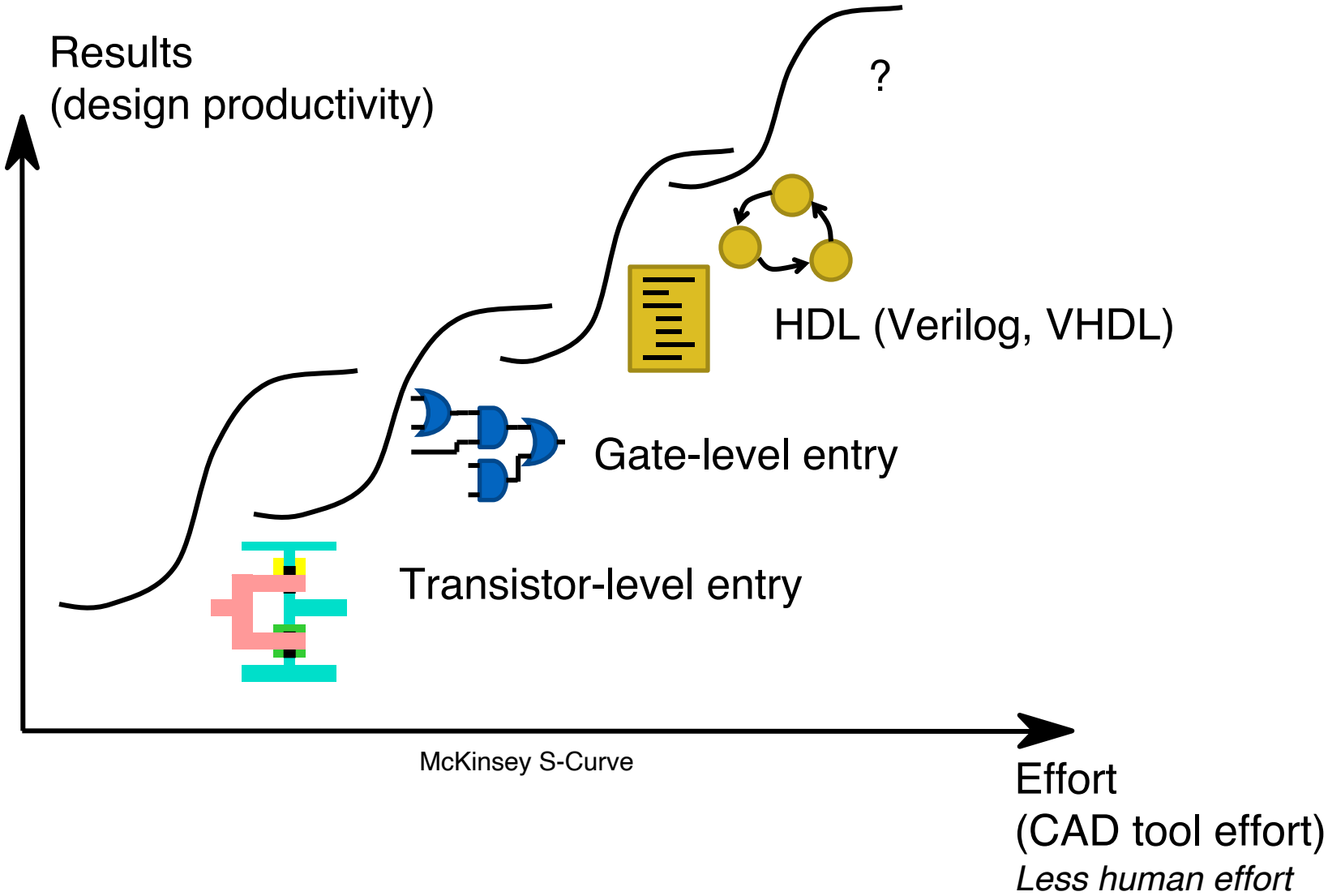
Learning Goals

- Understand FPGA's structure and functionalities
- Verilog basics
  - Signal, vectors, constants
  - Operators, modules
  - Wire, register types
- Verilog coding styles: structural v.s. behavioral
- Continuous assignment
- Procedural assignment
  - Blocking v.s. non-blocking assignment
- Understand Verilog testbenches

Course Plan

- Lab 1: Display data with 7-segment display
  - Build circuit using gates
- Labs 2-4: Hardware design using **Verilog**
  - … and deploy on **FPGAs**
- Lab 5: Assembly programming
  - … extension of Lab 4

Evolution of Design Abstractions



[source: Keutzer]

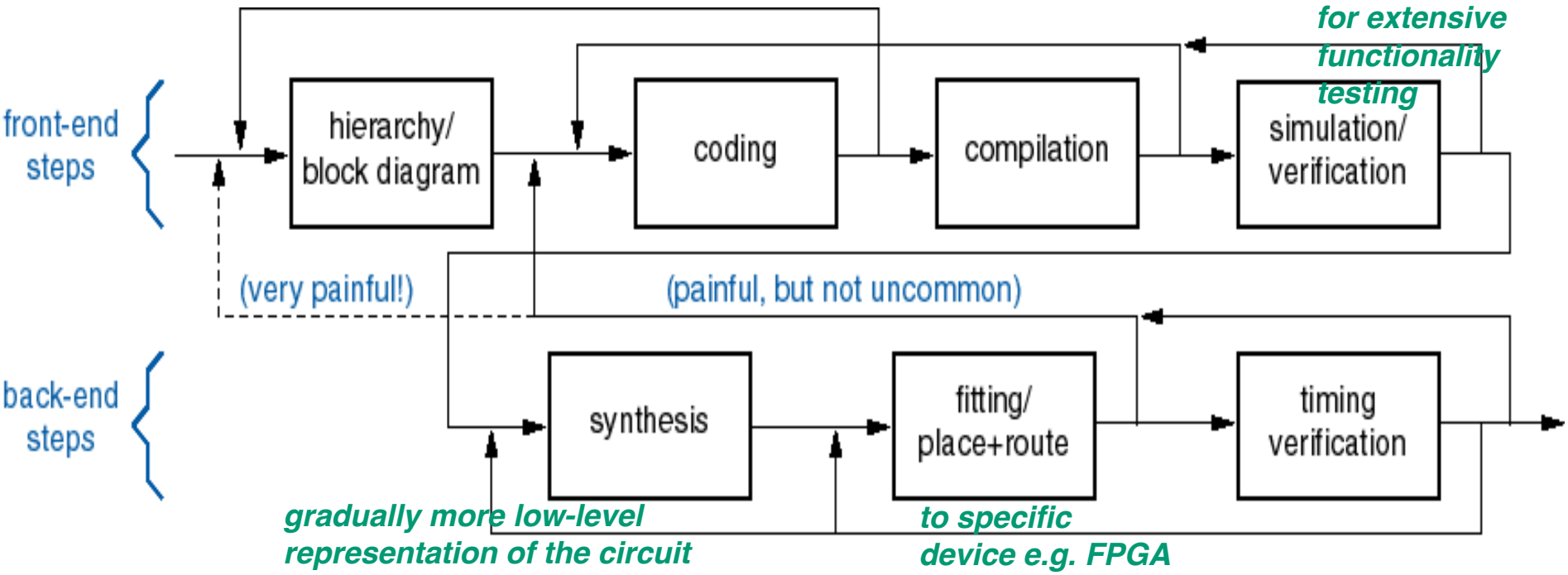
Hardware Description Languages

- Hardware Description Language(HDL)
  - A language for describing hardware designs
- Advantages of HDLs
  - Efficiently code large, complex designs
    - Program at a more abstract level than schematics
    - More readable than schematics
  - EDA/CAD tools automatically generate hardware
- Industry standards
  - Verilog
  - VHDL
  - SystemVerilog: a successor to Verilog, gaining popularity
- Reading: Chapter 4

HDL-based Design Flow: Overview

- HDL code written to describe hardware after brainstorming over a given problem
- HDL code, via testing scripts, can be simulated by simulation tools.
- HDL code is synthesized by sophisticated compiler tools into hardware circuits.

HDL-based Design Flow: Detailed Look



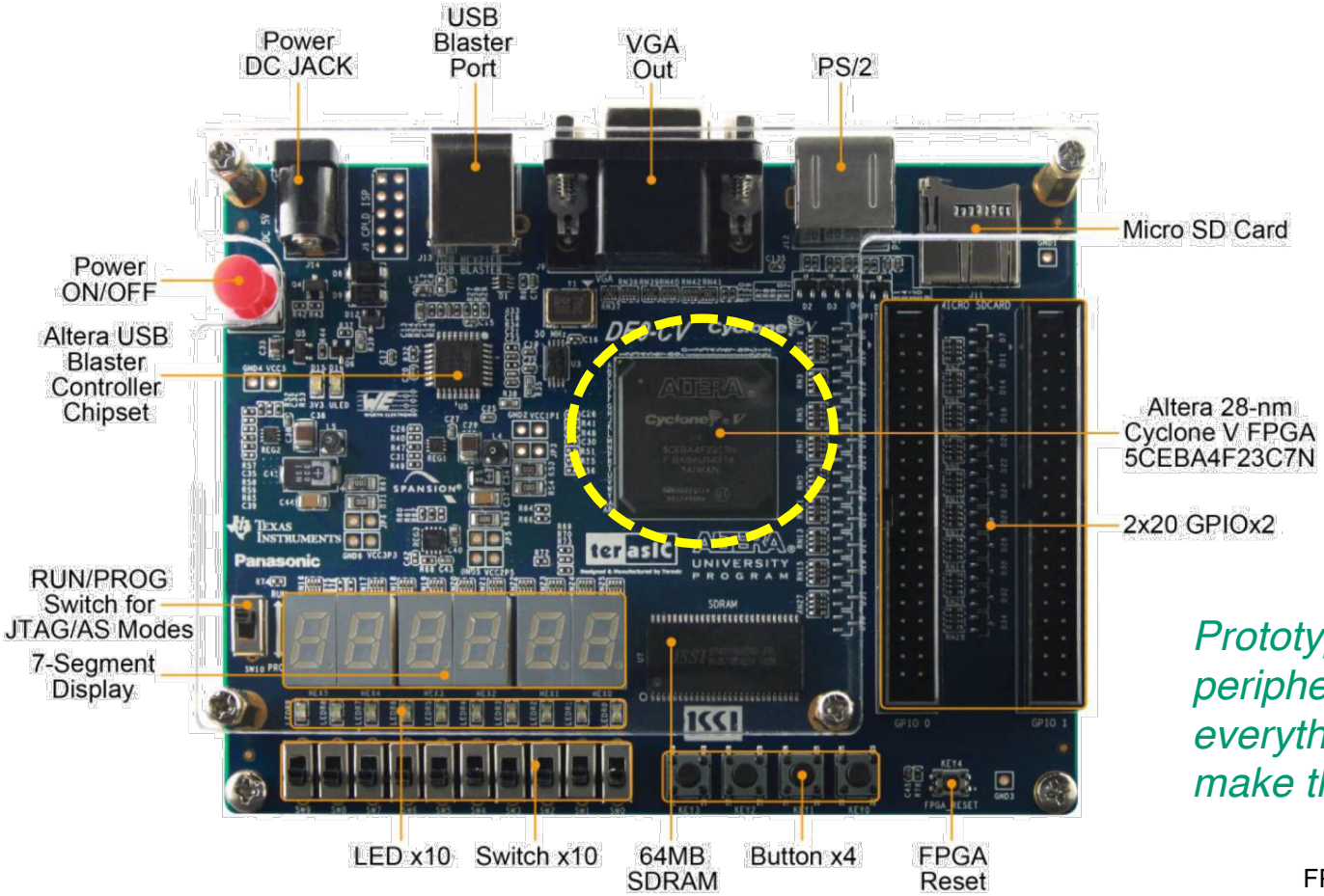
- Back-end differs by target technology
  - **FPGA**, ASIC, full-custom

The compilation tool is intelligent enough to parse high-level circuit descriptions, and can also simplify logic (recall K-Maps?)

FPGAs

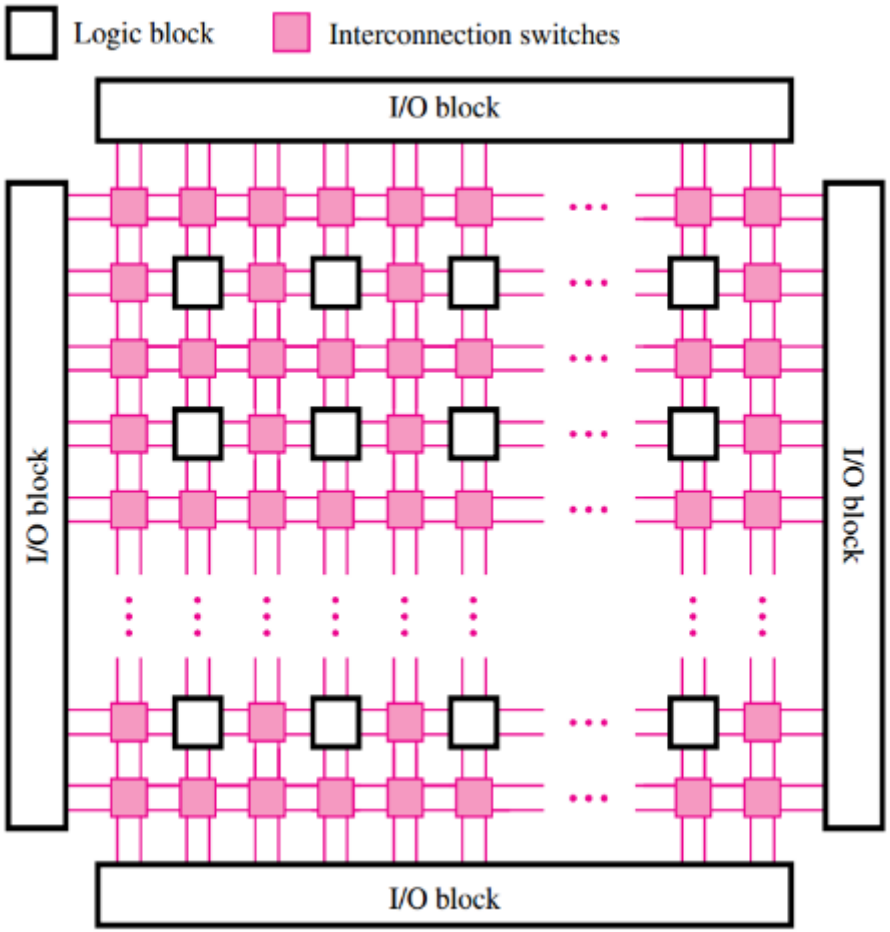
The platform to test/deploy our designed hardware

- **F**ield **P**rogrammable **G**ate **A**rray [FPGA]
  - is an Integrated Circuit (IC) that can be programmed in the field *after* manufacturing.

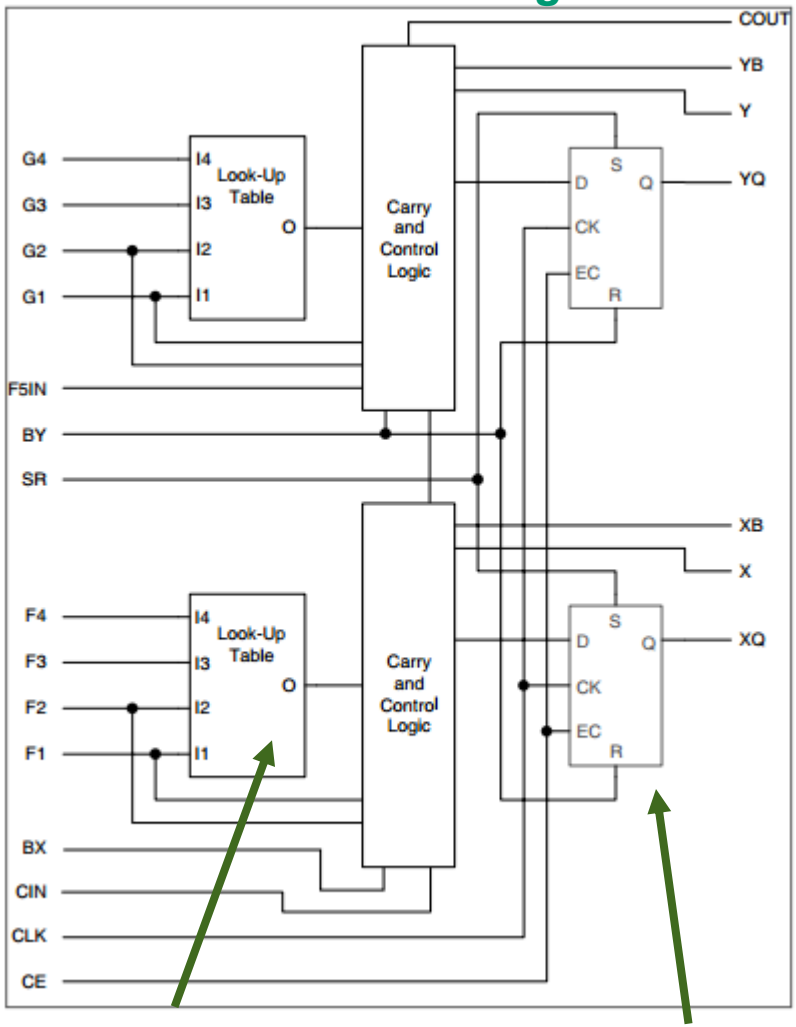


Prototyping board with peripherals and everything required to make the FPGA work

FPGA Structure



Inside a Logic Block



I/O blocks connected to peripherals

Stores Truth Tables

Flip-flop storage element

[source: Brown/Vranesic DLD/Verilog Textbook]

Why FPGAs

- Parallel execution in hardware
- Reconfigurable unlimited no. of times
- Readily available boards so quick to develop
- HDL Code → Hardware with little manual effort

- Uses
  - Accelerating/prototyping algorithms in hardware
  - variety of fields like defense, medical, machine learning

FPGA Ecosystem

- Multiple vendors

We'll use this in our course

… among others

- Let’s look at a couple of Verilog programs mapped to FPGAs

Recall Lab 1

- 4-bit Input to 7-Segment Decoder
- Digits 0-9

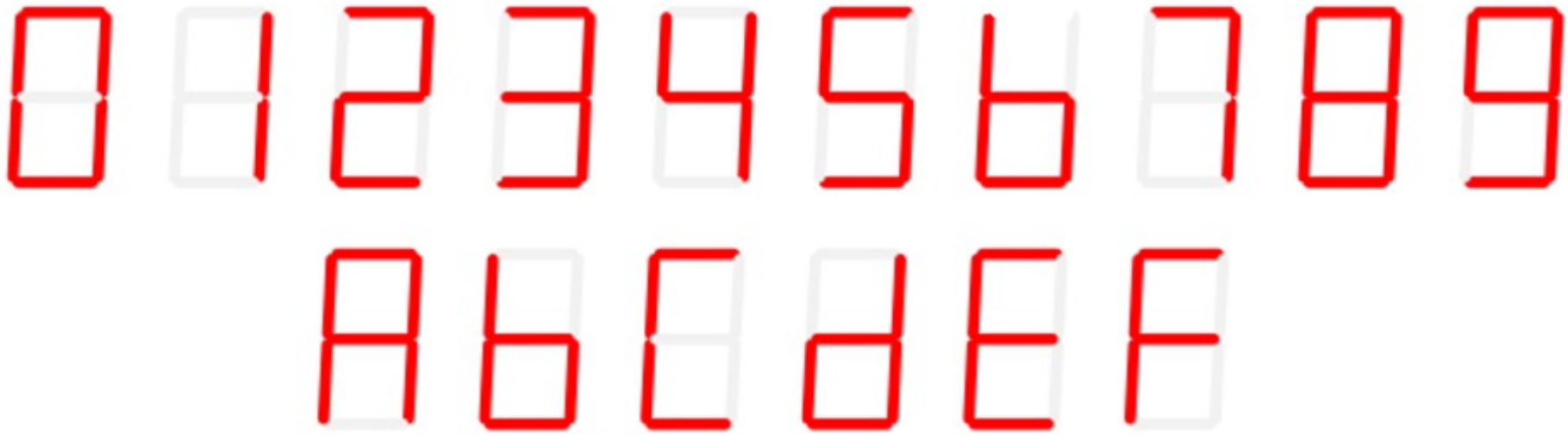
I_3	I_2	I_1	I_0	Digit
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

Each segment's expression was derived using K-Maps

$$a = I_3 + I_1 \cdot I_0 + I_2 \cdot I_0 + \overline{I_2} \cdot \overline{I_0}$$

7-Segment Decoder

- Extended to all 16 inputs



Quartus II 64-Bit - C:/Users/UMAR/Documents/2300-Demo/Tutorial/demo1/demo1 - demo1

File Edit View Project Assignments Processing Tools Window Help

Project Navigator

Entity

Cyclone V: 5CEBA4F23C7

single_digit_display

hex_to_seven_seg:centisecDisplay

Hierarchy Files Design Units IP Compone

Tasks

Flow: Compilation Customize...

Task	Time
Compile Design	
Analysis & Synthesis	
Fitter (Place & Route)	
Assembler (Generate programming files)	
TimeQuest Timing Analysis	
EDA Netlist Writer	
Program Device (Open Programmer)	

demo1.v hex_to_seven_seg.v

```
28        .out(BB[2])
29      );
30      SOFT u4(
31        .in(B[3]),
32        .out(BB[3])
33      );
34    `endif
35
36      always @ (BB)
37      begin
38        case (BB)
39          // segment order: GFEDCBA (active low)
40          4'h0 : SSEG_L = 7'b1000000;
41          4'h1 : SSEG_L = 7'b1111001;
42          4'h2 : SSEG_L = 7'b0100100;
43          4'h3 : SSEG_L = 7'b0110000;
44          4'h4 : SSEG_L = 7'b0011001;
45          4'h5 : SSEG_L = 7'b0010010;
46          4'h6 : SSEG_L = 7'b0000010;
47          4'h7 : SSEG_L = 7'b1111000;
48          4'h8 : SSEG_L = 7'b0000000;
49          4'h9 : SSEG_L = 7'b0010000;
50          4'hA : SSEG_L = 7'b0001000;
51          4'hB : SSEG_L = 7'b0000011;
52          4'hC : SSEG_L = 7'b1000110;
53          4'hD : SSEG_L = 7'b0100001;
54          4'hE : SSEG_L = 7'b0000110;
55          4'hF : SSEG_L = 7'b0001110;
56          default : SSEG_L = 7'b1111111;
57        endcase
58      end
59
60    endmodule
61
```

Verilog code For 7-Segment Decoder

7 segments defined ON or OFF for all 4-bit inputs

(straightforward & intuitive)

IP Catalog

Installed IP

Project Directory

No Selection Available

Library

Basic Functions

Bitec

DSP

Interface Protocols

Memory Interfaces and Controllers

Processors and Peripherals

University Program

Search for Partner IP

Add...

All <<Filter>> Find... Find Next

Type	ID	Message

Messages System Processing

Let's compile this code and see the hardware mapping on the FPGA

0% 00:00:00

Technology Map Viewer - Post-Fitting - C:/Users/UMAR/Documents/2300-Demo/Tutorial/demo1/demo1 - demo1

File Edit View Tools Window Help

Purely combinational circuit

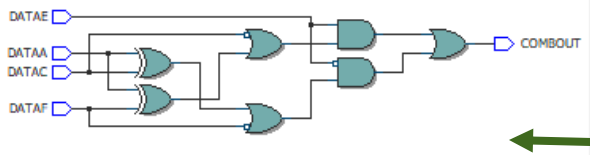
Search altera.com

Page: 1 of 1

Properties

Block: single_digit_display|hex_to_seven_seg:centisecDisplay|WideOr0~0|F

Lutmask: 5F5F5A5AAFAFFFFF



Gate Level Representation of a single segment output

WideOr0~0 F

	DATAF	DATAE	DATAC	DATAA	OUT
1	0	0	0	0	1
2	0	0	0	1	1
3	0	0	1	0	1
4	0	0	1	1	1
5	0	1	0	0	1
6	0	1	0	1	1
7	0	1	1	0	0
8	0	1	1	1	1
9	1	0	0	0	0
10	1	0	0	1	1
11	1	0	1	0	1
12	1	0	1	1	0
13	1	1	0	0	1

Ports Truth Table Equation

Netlist Navigator Properties

demo1:1

hex_to_seven_seg:centisecDisplay

KEY[0..3]~input

4-bit Binary Input

WideOr0~0

2 DATAA
1 DATAC COMBOUT
0 DATAE
3 DATAF

LOGIC_CELL_COMB

WideOr0~0

Truth tables for all 7 segments

WideOr1~0

1 DATAA
2 DATAD COMBOUT
0 DATAE
3 DATAF

LOGIC_CELL_COMB

WideOr1~0

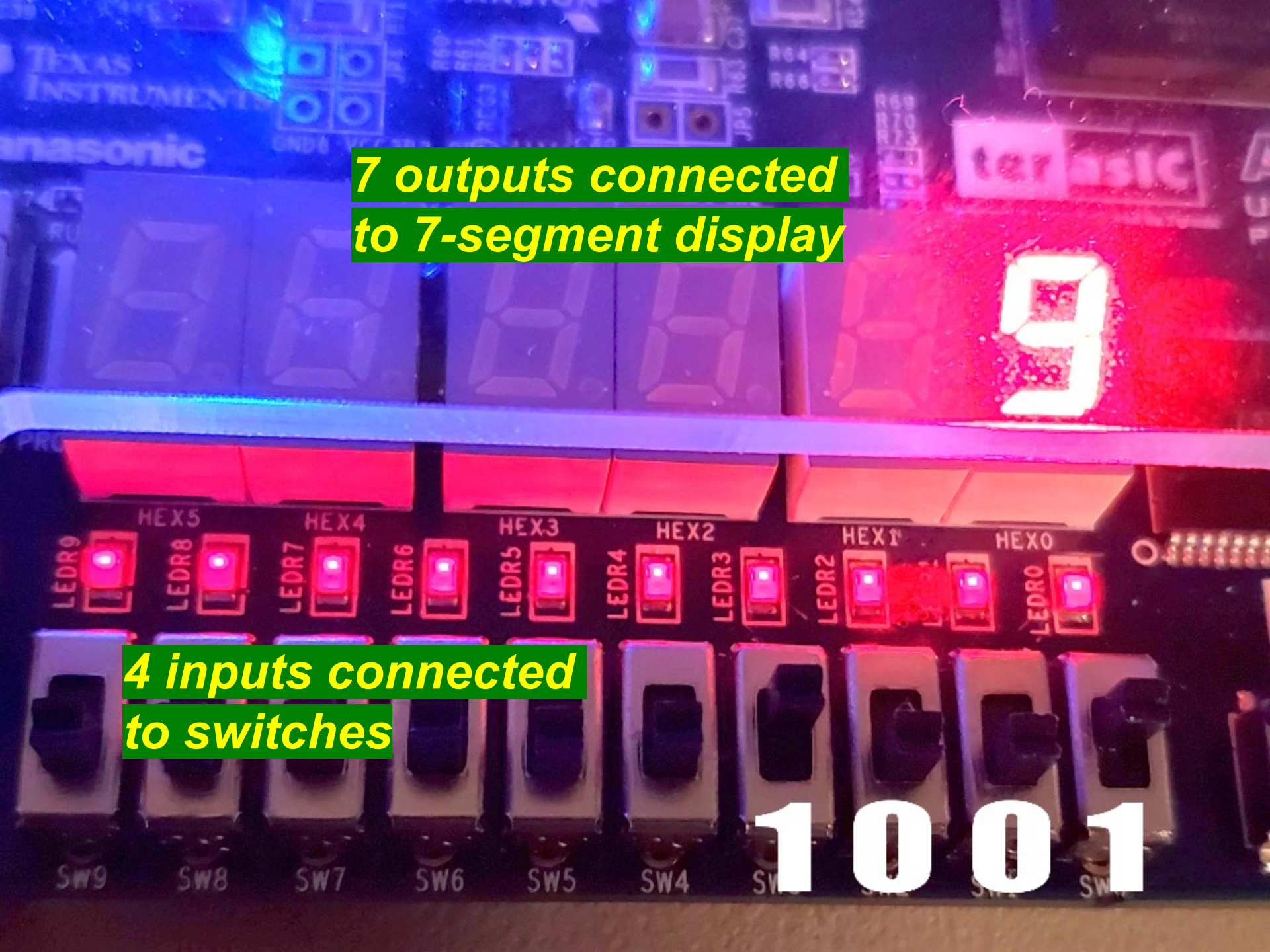
WideOr2~0

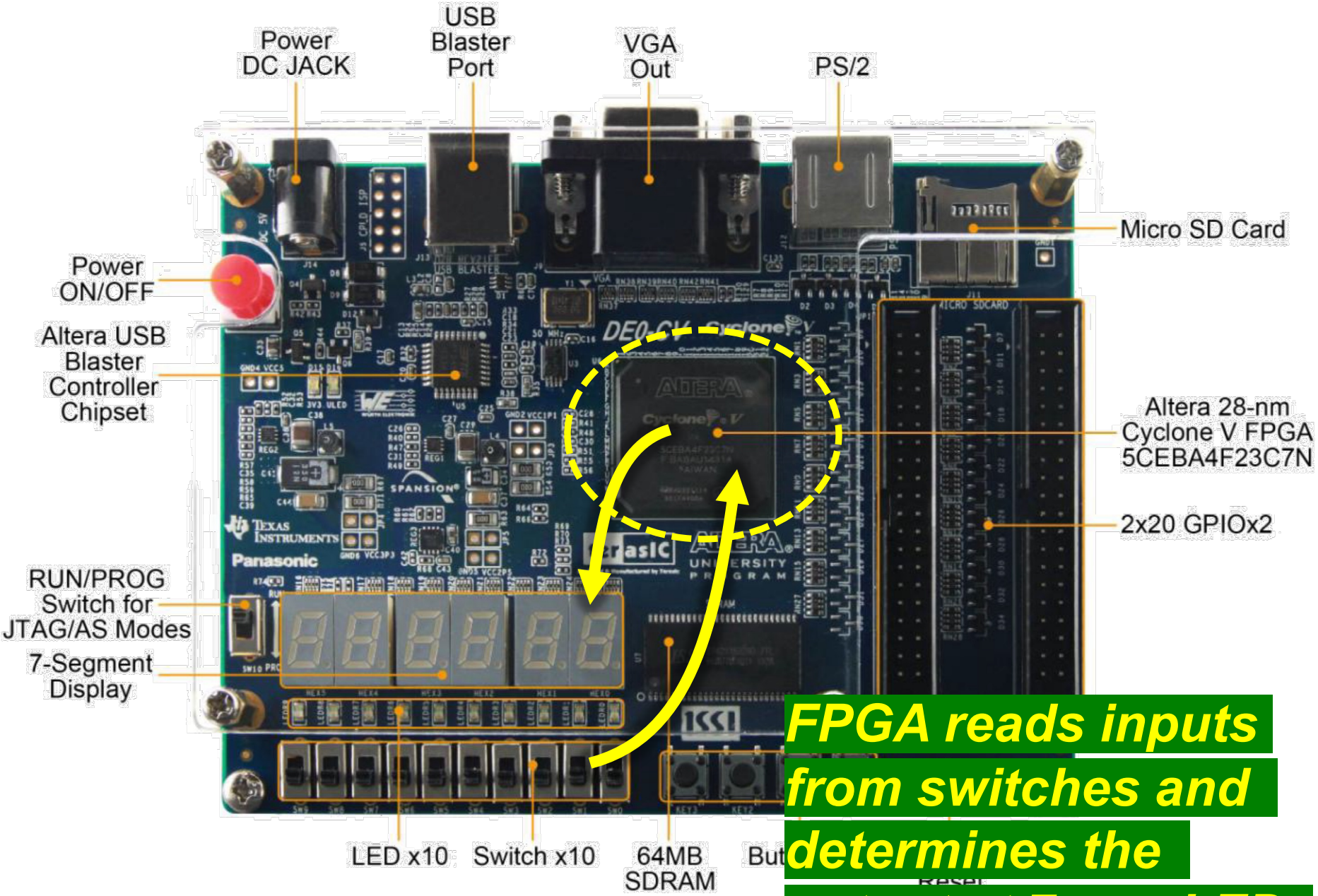
2 DATAA
1 DATAC COMBOUT
0 DATAE
3 DATAF

WideOr2~0

4-input Look-up Table (Truth Table) for a single segment output

100% 00:00:01


7 outputs connected
to 7-segment display
4 inputs connected
to switches
1001
HEX5
HEX4
HEX3
HEX2
HEX1
HEX0
LEDR9
LEDR8
LEDR7
LEDR6
LEDR5
LEDR4
LEDR3
LEDR2
LEDR0
SW9
SW8
SW7
SW6
SW5
SW4



FPGA reads inputs from switches and determines the output at 7-seg LEDs

Sequential circuit example

Technology Map Viewer - Post-Fitting - C:/Users/UMAR/Documents/2300-Demo/Tutorial/demo0/tffp - tffp

File Edit View Tools Window Help

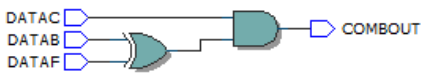
Search altera.com

Page: 1 of 1

Properties

Block: |tffp|Q~0|F

Lutmask: 30303030C0C0C0C0



Q~0 F

	DATAF	DATAC	DATAB	OUT
1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	1
5	1	0	0	0
6	1	0	1	0
7	1	1	0	1
8	1	1	1	0

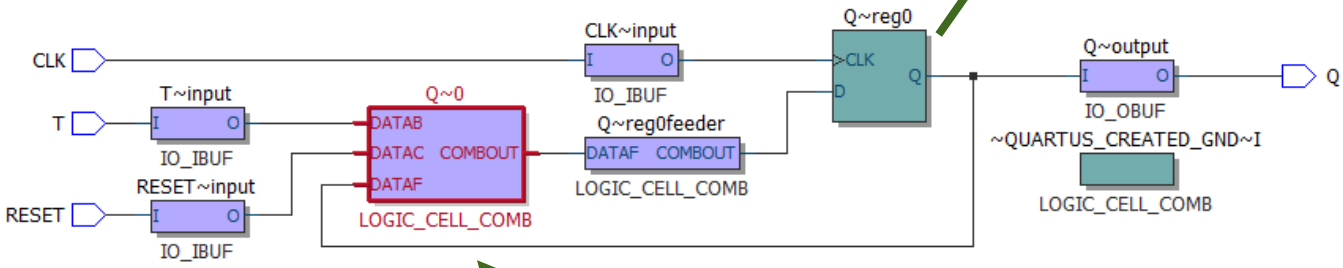
Ports Truth Table Equation

Netlist Navigator Properties

tffp:1

Result of compiling a T Flip-Flop in Verilog (Prelab 2A)

D Flip-Flop storage element as no direct T Flip-Flop available



Truth Table Logic determines when to toggle/reset/retain the D Flip-Flop value based on inputs T and RESET

Notice the output feedback

0% 00:00:00

VERILOG HDL

Verilog

- Developed in the early 1980s by Gateway Design Automation (later bought by Cadence)
- Supports modeling, simulation, and synthesis
- We will use a subset of language features
  - Synthesizable vs. non-synthesizable

Verilog HDL

- Verilog is not a computer programming language
- Verilog is not a computer programming language
- Verilog is not a computer programming language

- Verilog is a Hardware Description Language
  - It describes hardware
  - Things happen simultaneously / in parallel
  - …. whereas software is sequential

Signal Values

- Verilog signals can have 1 of 4 values
  - 0 Logical 0, or false
  - 1 Logical 1, or true
  - z High impedance, floating
  - x Unknown logical value
    - May be a 0, 1, z, in transition, or don't cares.

You'll be mostly concerned with Logical 0 and 1 only.

Actual hardware has levels 0 and 1 only.

X and Z values might arise in your simulation tool.

Vectors

- Multi-bit values are represented as bit vectors (grouping of 1-bit signals)
  - Right-most bit is least significant
  - Example
    - input [7:0] byte1, byte2, byte3;

- Common operations on bit vectors
  - Bit selection
    - byte1[5:2] or Zbus[3:7]
  - Concatenation
    - {byte1,byte2}
  - Bitwise Boolean operators
    - byte1 & byte2

Constants

4’b1001

Base (b, d, h, o)

Decimal number representing bit width

- Binary constants
  - 8’b00000001
  - 4’b0111
- Decimal constants
  - 4’d10
  - 8’d345
  - 32’d65536

Operators

- Bitwise Boolean operators

  ~ NOT

  & AND

  ^ Exclusive OR

  | OR

  Operate bit-by-bit on multi-bit signals

- Arithmetic operators

  + Addition

  – Subtraction

  * Multiplication

  / Division

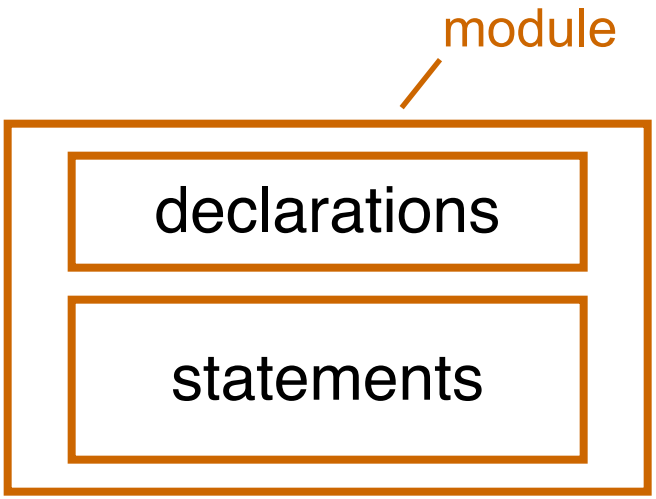
  % Modulus

  << Shift left

  >> Shift right

Verilog Program Structure

- System is a collection of modules
  - Module corresponds to a single piece of hardware



- Declarations
  - Describe names and types of inputs and outputs
  - Describe local signals, variables, constants, etc.

- Statements specify what the module does

Verilog Program Structure

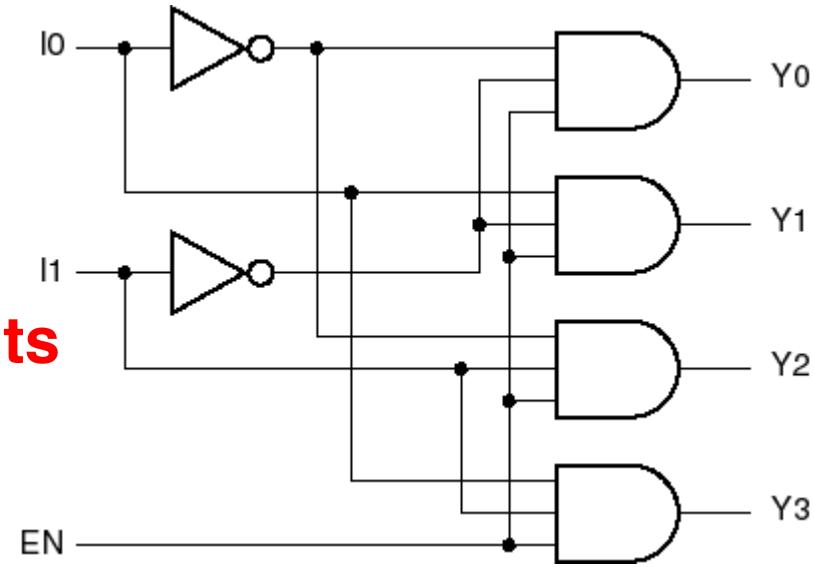
```
module V2to4dec( i0,i1,en,y0,y1,y2,y3 );
   input i0,i1,en;
   output y0,y1,y2,y3;
   wire noti0,noti1;

   not U1(noti0,i0);
   not U2(noti1,i1);
   and U3(y0,noti0,noti1,en);
   and U4(y1,     i0,noti1,en);
   and U5(y2,noti0,     i1,en);
   and U6(y3,     i0,     i1,en);
endmodule
```

List of inputs and outputs

Declarations

Statements

Structural Coding Style



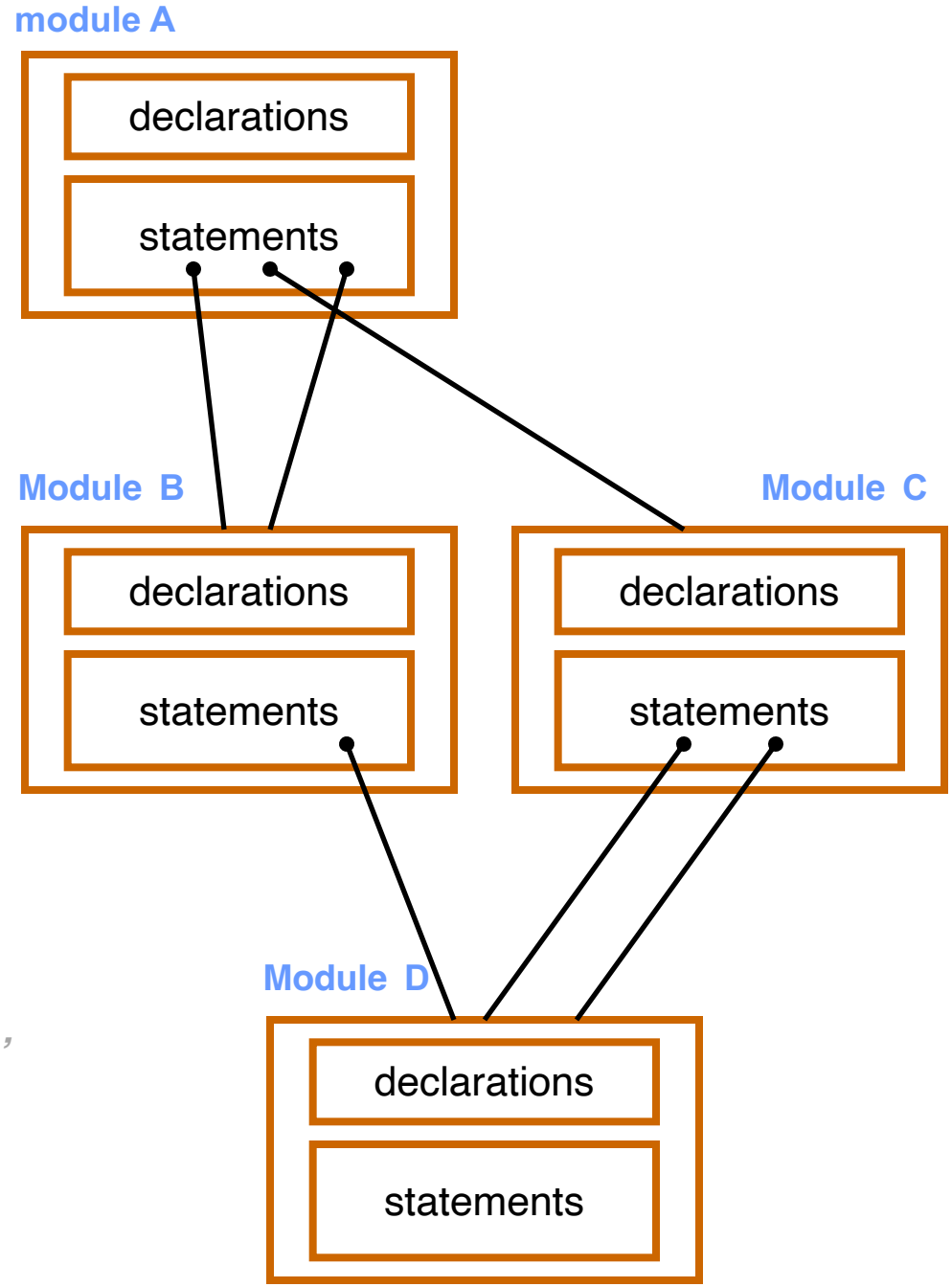
In Declarations, can have something like: input [3:0] B;

This would be a 4-bit bus, with individual wire names
B[0] (LSB), B[1], B[2], B[3] (MSB)

Verilog Hierarchy

A module can instantiate other modules inside, forming a module hierarchy

Akin to programming language 'classes' instantiating objects of other classes



Verilog Programming Styles

- Structural ***You just saw this a couple of slides ago***
  - Textual equivalent of drawing a schematic
  - Uses *instance* statements to instantiate other modules and uses wires to connect them

- Behavioral ***More efficient - we'll mostly use this style in our labs***
  - Specify what a module does with a high-level description
  - Uses *procedural code* (e.g., in *always* blocks) or *continuous assignment* (i.e. *assign*) constructs

We can mix the structural and behavioral styles in a Verilog design

Net and Variable Types

- Verilog provides multiple net and variable types

You must be clear on their usage!
They're a common source of errors.

- We will main use two types
  - ***wire***: represents a physical connection (net) between hardware elements
    - Used for structural style and continuous assignments
    - Default if you do not specify a type
    - Can only be used to model combinational logic
    - Cannot be used in the left-hand side in an always block
  - ***reg***: a variable that can be used to store state (registers)
    - Used in the procedural code (*always* blocks)
    - May also be used to express combinational logic
    - Can be used to model both combinational & sequential logic
    - Cannot be used in the left-hand side of a continuous assignment statement

More on this later.

Internal Wires

- Sometimes it is useful to name internal signals and assign intermediate values
  - Declaration:

    ```
    wire temp_value;
    ```

  - Assignment:

    ```
    assign temp_value = some_statement;
    ```

Example: 2-to-1 MUX

- select = Selector signal
- If select = 0, out = x
- If select = 1, out = y

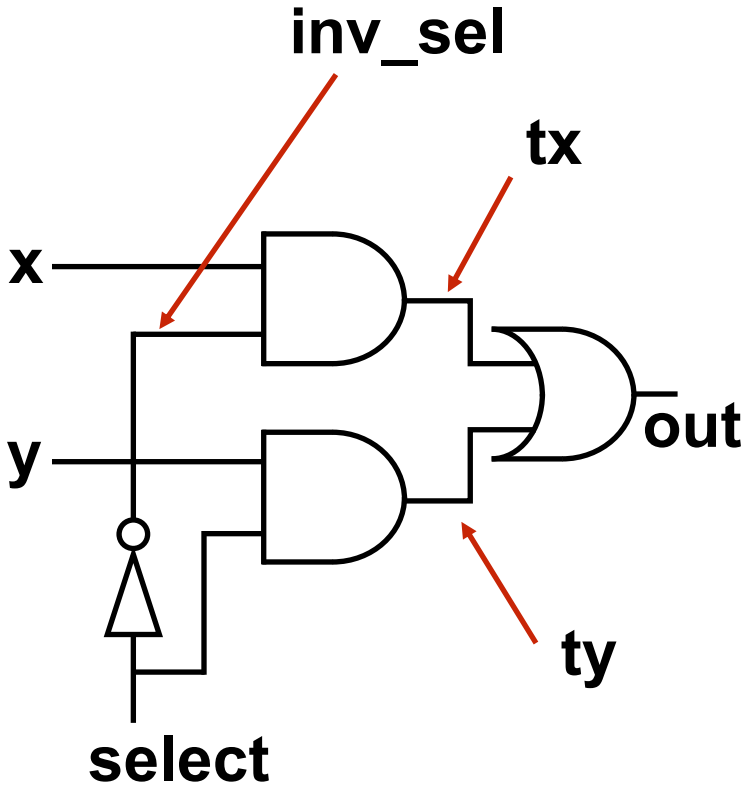
out = select' x + select y

select	x	y	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Structural Style

```
module MUX2_1 (x, y, select, out);
  input  x, y;
  input  select;
  output out;

  wire inv_sel, tx, ty;

  NOT not0(inv_sel,select);
  AND and0 (tx, x, inv_sel);
  AND and1 (ty, y, select);
  OR  or0 (out, tx, ty);

endmodule
```

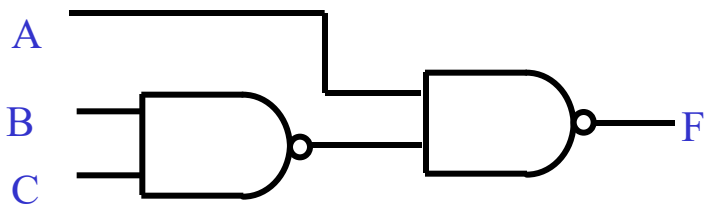
Internal wires used to connect gates.



The order of the gates instantiation does not matter. Essentially describing the schematic textually.

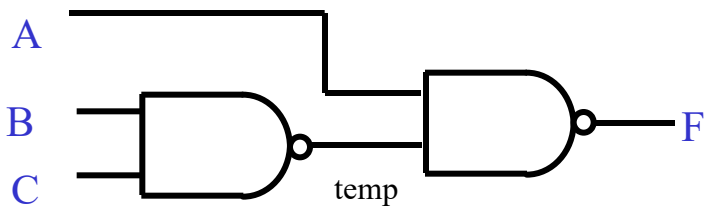
Structural Practice Example

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


   nand U1(                 );

   ….



endmodule
```



Structural Practice Example

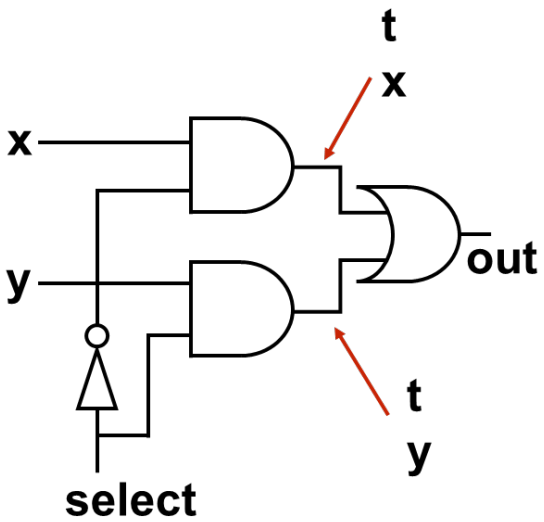
```
module mycircuit(A, B, C, F );
   input A, B, C;
   output F;
   wire temp;

   nand U1(temp, B, C);

   nand U2(F, A, temp);



endmodule
```



Behavioral Coding Style

Behavioral Coding Style: Continuous Assignments

- An *assign* statement represents continuously executing <u>combinational logic</u>




```
module MUX2_1 (x, y, select, out);
  input  x, y;
  input  select;
  output out;

  assign out = (~select & x) | (select & y);

endmodule
```

Output out is a wire by default.

Assign used to set values for wires

- Multiple continuous assignments happen in parallel; the order does not matter

Continuous Assignments

- An *assign* statement means continuously executing combinational logic
  - LHS must be a ***wire*** type
  - RHS is recomputed when a value in the RHS changes
  - The new value of the RHS is assigned to the LHS

- Example:
  - **assign** y0 = en & ~i0 & ~i1;

Behavioral Coding Style: Combinational Logic with Always Blocks

```
module MUX2_1 (x, y, select, out);
  input  x, y;
  input  select;
  output reg out;

  always @(x, y, select)
  begin
    out = (~select & x) | (select & y);
  end

endmodule
```

- **An *always* block is reevaluated whenever a signal in its sensitivity list changes**
- **Formed by *procedural assignment* statements**
  - ***LHS inside the always block must be reg***

Here, the signal is a 'reg' even though modeling combinational logic! reg can be used to model both combinational and sequential logic.

This is Behavioral Style Too!

```
module MUX2_1 (x, y, select, out);
  input  x, y;
  input  select;
  output reg out;

  always @(x, y, select)
  begin
        if (select == 0)
                out = x;
        else
                out = y;
 end

endmodule
```

Conditional Statement

- ***Closely resembles verbal circuit description!***
- ***High-level and intuitive.***
- ***Synthesis Tool is intelligent enough to be able to infer a mux!***

Conditionals

- Logical operators used in conditional expressions
  - && logical AND
  - || OR
  - ! logical NOT
  - == logical equality
  - != logical inequality
  - > greater than
  - >= greater than or equal
  - < less than
  - <= less than or equal
- Don’t confuse with Bitwise Boolean operators

Concurrent Elements/Blocks

```
module temp (…..);
  input  …;
  output …;

 assign … = ...;

 always @(…)
   begin  …
   end

 always @(…)
   begin  …
   end

endmodule
```

- *Always* blocks execute **concurrently** with other *always* blocks, instance statements, and continuous assignment statements in a module

Sensitivity Lists in *Always* Blocks

- Include every variable in the always block sensitivity list
  - always @ (signal list)

- Simple alternative
  - always @ (*)

Used to describe Combinational Logic

- For edge-triggered behavior
  - always @ (posedge/negedge SIGNAL_NAME)

Used to describe Sequential Logic

More later!

Assignments in Verilog

- Continuous assignments apply to **combinational** logic only

- *Always blocks* contain a set of procedural assignments (blocking or non-blocking)
  - Can be used to model <u>either **combinational** or **sequential** logic</u>
  - Always blocks execute concurrently with other always blocks, instance statements, and continuous assignment statements in a module

Procedural Statements

- Procedural statements are similar to conventional programming language statements
  - Begin-end blocks
    - begin *procedural-statement* … *procedural-statement* end
  - If
    - if (*condition*) *procedural-statement* else *procedural-statement*
  - Case
    - case (*sel-expr*) *choice* : *procedural-statement* … endcase
  - Blocking assignment
    - *variable name* = *expression* ;
  - Nonblocking assignment
    - *variable name* <= *expression* ;

You saw this in the Behavioral example.

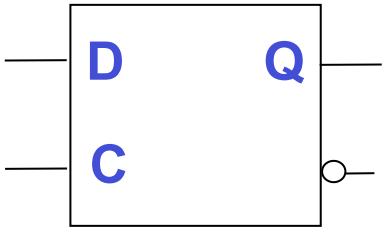
More later!

- So far you've seen how to implement **combinational** logic.
  - output as a function of input
  - determined using logic gates

- Now for **sequential** logic…
  - introduce storage elements like latches, flip-flops

Sequential Logic in Always Blocks

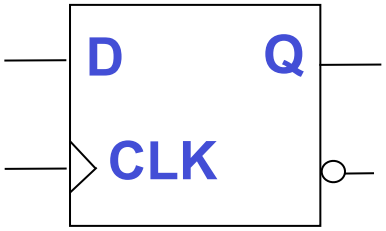
```
reg Q;

always @( clk, D )
begin
  if ( clk )
      Q <= D;
end
```



D latch

- **Sequential logic can only be modeled using always blocks**

```
always @( posedge clk )
begin
  Q <= D;
end
```



DFF

- **Thus, Q must be declared as a "reg"**

Flip-Flop with a reset?

```
reg    q;

always @ (posedge clk or posedge reset)
   if (reset==1)
      q <= 1'b0;
   else
      q <= d;
```

Rising clock edge

Asynchronous Reset in between clock edges

Flip-Flop assignment

Flip-Flop with a reset?

```
reg    q;

always @ (posedge clk)
   if (reset==1)
      q <= 1'b0;
   else
      q <= d;
```

Rising clock edge

Synchronous Reset

Flip-Flop assignment

Blocking Assignments

- Blocking assignments (=)
  - Simulation behavior: Right-hand side (RHS) evaluated <u>sequentially</u>; assignment to LHS is <u>immediate</u>

```
reg Y, Z;
always @ (posedge clk)
begin
   Y = A & B;
   Z = ~Y;
end
```

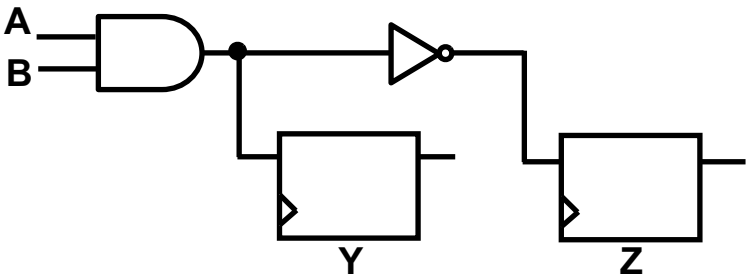
Y and Z are flip-flops in actual hardware

Blocking assignments

Simulator interpretation

$Y_{next} = A \& B$
$Z_{next} = \sim(A \& B)$

Resulting circuit (post synthesis)



Nonblocking Assignments

- Nonblocking assignment (<=)
  - Simulation behavior: RHS evaluated in parallel (order doesn't matter); Assignment to LHS is delayed until end of always block

```
reg Y, Z;
always @ (posedge clk)
begin
   Y <= A & B;
   Z <= ~Y;
end
```

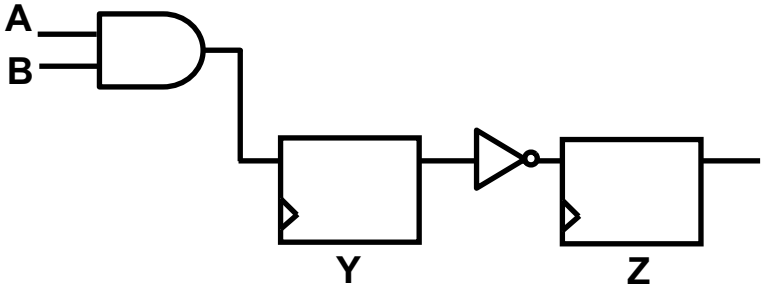
Y and Z are flip-flops in actual hardware

Nonblocking assignments

Simulator interpretation

Z_{next} = ~Y // reading the old Y
Y_{next} = A & B

Resulting circuit (post synthesis)



Rules to Follow

- Use blocking assignments to model combinational logic within an always block.
  - or just implement combinational without an always block, using assign statements
- Use non-blocking assignments to implement sequential logic.
- Do not mix blocking and non-blocking assignments in the same always block.
- Do not make assignments to the same variable from more than one always block

VERILOG TESTBENCHES

To extensively test our HDL design using simulations before deploying it in hardware

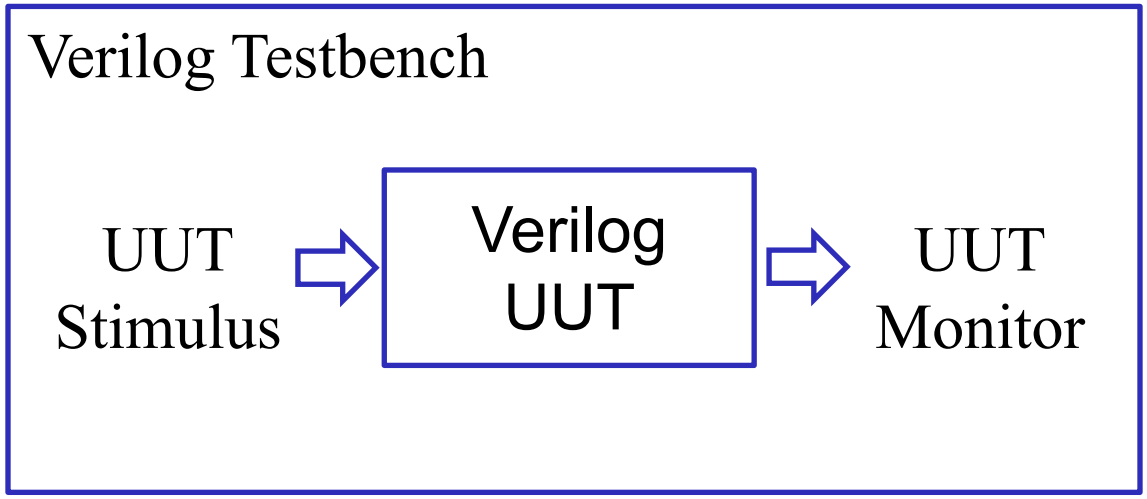
We should have a comprehensive test case coverage (including edge cases)

Why Testbenches?

- Generate stimulus *i.e. test inputs*
- Monitor UUT (Unit Under Test) output

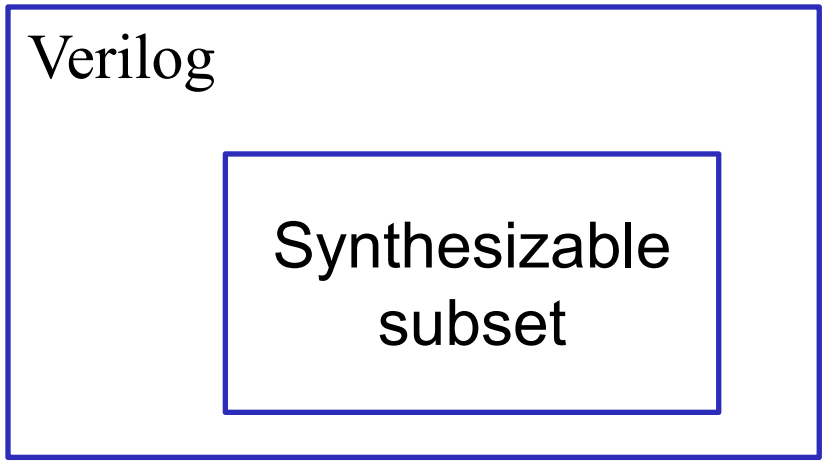
  and hope it matches the expected output

- Automation



Testbenches vs. Hardware

- Important distinction
  - Verilog testbenches **DO NOT** describe hardware (i.e., non-synthesizable)
  - Verilog testbenches look like a software program



e.g. we frequently use 'delays' in testing when applying new test inputs, and print out signal values in the output log

Clearly such features are only relevant to simulation, they CAN'T translate to hardware

Building Blocks of a Testbench

- Timescale
- Module definition
- UUT instantiation
- Stimulus generation
- Monitoring output

Timescale

- Specifies the simulation granularity
- Syntax

```
`timescale time_unit base / precision base
```

unit of delays *for fractional delay*

- Example

```
// sets the granularity at which we simulate
`timescale 1 ps / 1 ps
```

Test Module Definition

- A testbench is a module too!
- No inputs or outputs

```
module treg4bit_test();
```

Module Declarations

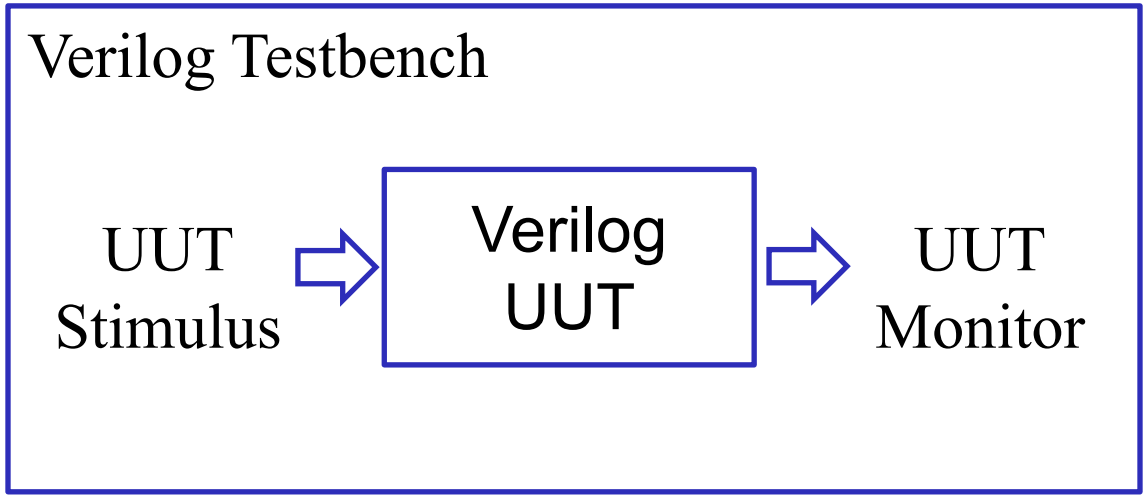
- Define input and output signals that will link to our UUT

```
// inputs
reg          CLK50;
reg          RESET;
reg  [3:0]   IN;

// outputs
wire [3:0]   OUT;
```

UUT Instantiation

- Same as instantiating a submodule

```
// instance of treg4bit
 treg4bit UUT (
   .CLK(CLK50),
   .RESET(RESET),
   .IN(IN),
   .OUT(OUT)
 );
```



Stimulus Generation: Clock

- Generate a 50MHz clock

```
always
 begin
    CLK50 = 1'b0;
    CLK50 = #10000 1'b1;
    #10000;
 end
```

#10000 means a delay i.e. wait for 10000 time units before changing the value of CLK50

The total time period is 20000 time units

Stimulus Generation: Initial Block

- Use ***initial*** block to write sequential test cases for the circuit

```
initial
begin

  // reset circuit
  ...

  // test cases
  ...

end
```

Stimulus Generation: Reset

- At beginning of *initial* block, give initial values to the circuit

```
// reset circuit
RESET = 1'b1;
IN = 4'b0;

#20000;
// test whether reset is correct
...

// enable and start running
RESET = 1'b0;
```

Stimulus Generation: Set Inputs

- Set inputs to values for the test case

```
// enable toggling for only some of the flip-flops
IN = 4'b0011;

#20000;  // wait a cycle to observe result
```

Monitoring Output

- Compare actual output with expected output
- Display information

```
if(OUT == 4'b0) begin
  $display("MSIM> Reset is correct, OUT (value in
  register) now %4b", OUT);
end
else begin
  $display("MSIM> ERROR: Reset is incorrect. OUT
  (value in register) should be 0000, but is %4b",
  OUT);
end
```

- Can also use waveforms to see inputs and outputs change over time

ModelSim ALTERA STARTER EDITION 10.3d

File Edit View Compile Simulate Add Transcript Tools Layout Bookmarks Window Help

sim - Default

Instance	Design unit	Des
tffp_test	tffp_test	Mo
UUT	tffp	Mo
#ALWAYS#42	tffp_test	Pro
#INITIAL#53	tffp_test	Pro
#vsim_capacity#		Cap

Objects

Name	179277 ps	
CLK50	1	Regi... I
Q	St1	Net I
RESET	0	Regi... I
T	1	Regi... I

Processes (Active)

Name	Type (filtered)
#INITIAL#53	Initial
#ALWAYS#42	Always

Wave - Default

	Msgs
/tffp_test/CLK50	1
/tffp_test/RESET	0
/tffp_test/T	1
/tffp_test/Q	St1
Now	180 ns
Cursor 1	179.277 ns

Input/Output signals

Waveform display

Transcript

```
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# MSIM>
# MSIM> TEST CASES
# MSIM>
# MSIM> Reset is correct, Q (value in flip-flop) now 0
# MSIM> No toggle state is correct, Q (value in flip-flop) now 0
# MSIM> Toggle state is correct, Q (value in flip-flop) now 1
# MSIM> Toggle state is correct, Q (value in flip-flop) now 0
# MSIM> No toggle state is correct, Q (value in flip-flop) now 0
# MSIM> Toggle state is correct, Q (value in flip-flop) now 1
# MSIM> Reset with toggle on is correct, Q (value in flip-flop) now 0
# MSIM> Toggle after reset is correct, Q (value in flip-flop) now 1
# ** Note: $stop    : C:/Users/UMAR/Documents/2300-Demo/Tutorial/demo0/tffp_test.v(165)
#    Time: 180 ns  Iteration: 0  Instance: /tffp_test
```

Testbench output i.e. print/display statements

Now: 180 ns Delta: 0 sim:/tffp_test/#INITIAL#53

Pause/Terminate simulation

- Pause simulation (can resume later)

```
$stop;
```

- Terminate simulation

```
$finish;
```

Learning Goals

- Understand FPGA's structure and functionalities
- Verilog basics
  - Signal, vectors, constants
  - Operators, modules
  - Wire, register types
- Verilog coding styles: structural v.s. behavioral
- Continuous assignment
- Procedural assignment
  - Blocking v.s. non-blocking assignment
- Understand Verilog testbenches