ECE 351
Verilog and FPGA Design

Lecture 2: FPGAs, ASICs, ASSPs, SoCs (wrap-up)
Simulation w/ QuestaSim

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Portland State UNIVERSITY

FPGA's, ASIC's, SoC's (wrap-up)

Source material drawn from:
- *ASIC Design Methodology Primer*, IBM ASIC Products Application Note, Initial publication 5/98
- Wikipedia
- Slideware provided by Xilinx
- Dr Song's lecture notes
- Roy's ECE 540 lecture notes

ASIC's, ASPP's and Full custom IC's

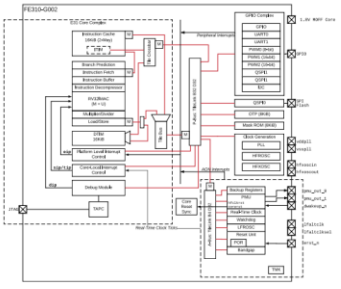
- **A**pplication **S**pecific **I**ntegrated **C**ircuit - a custom chip designed for a very specific purpose.
  - Ex: a chip with a DSP front-end designed for a specific model of cardiac monitor made by only one manufacturer
  - Companies implement their ASIC designs in a single silicon die by mapping their functionality to a set of predesigned and verified library of circuits provided by the ASIC vendor. The components of the library are described in the ASIC vendor's databook
- **A**pplication **S**pecific **S**tandard **P**roduct - a semi-custom chip designed for a specific application and sold to multiple customers
  - Ex: an integrated circuit that does video or audio encoding/decoding
- Full custom chip – an integrated circuit designed by a single company for a specific application, usually huge volumes
  - Ex: an Intel CPU
  - Full-custom chips are developed for a specific semiconductor technology, often with huge teams of architects, logic designers, circuit designers, validation engineers, layout designers, et. al. and take several years to design

FPGA's

- **F**ield **P**rogrammable **G**ate **A**rray - An integrated circuit designed to be configured by a customer or a designer after manufacturing—hence field-programmable
  - The FPGA configuration is generally specified using a hardware description language (HDL)
  - FPGAs can be used to implement any logical function that an ASIC can perform but the ability to update the functionality after shipping and the low non-recurring engineering costs relative to an ASIC design offer advantages for many applications
  - FPGAs contain programmable logic components called "configurable logic blocks" and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" when the FPGA is configured (and reconfigured…and reconfigured…and reconfigured…and…)

SoC's

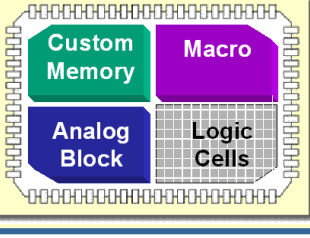
- **S**ystem **o**n **C**hip or SOC – An integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip.
  - Ex: an embedded system such as a cell phone or cable box
  - SoC's may contain digital, analog, mixed-signal, and often radio-frequency functions—all on a single chip substrate
- An SoC can be implemented as an ASIC or as an ASSP or in an FPGA or as a full custom chip
  - Roy's definition: An SoC contains at least one embedded CPU running an application interfaced to one or more vendor-supplied, 3rd party or custom IP blocks



SiFive FE310-G002 top-level block diagram

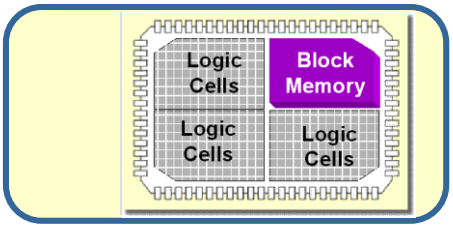
ASIC: Standard Cell

- Advantages
  - Lowest price for high-volume production (greater than 200K per year)
  - Fastest clock frequency (performance)
  - Unlimited size
  - Integrated analog functions
    - Custom ASICs
  - Low power
- Disadvantages
  - Highest non-recurring engineering costs
  - Longest design cycle
  - Limited vendor IP with high cost
  - High cost for ECO's



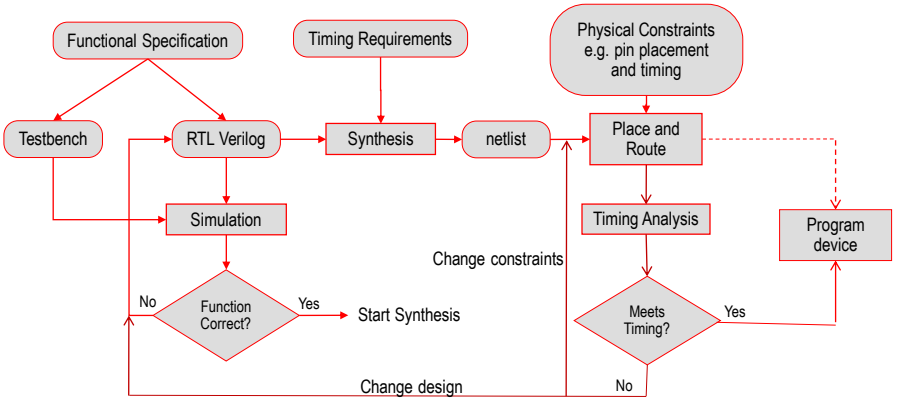
FPGA

- Advantages
  - Lowest cost for low-volume to medium-volume production
  - No non-recurring engineering costs
  - Standard product
  - Fastest time to market
  - Extensive library of IP
    - Inexpensive compared to ASIC vendors
  - Ability to make bug fixes quickly and inexpensively
- Disadvantages
  - Slower performance
  - Size limited to ~25 million system gates
  - Digital only



Portland State UNIVERSITY

FPGA Design Flow



Portland State UNIVERSITY

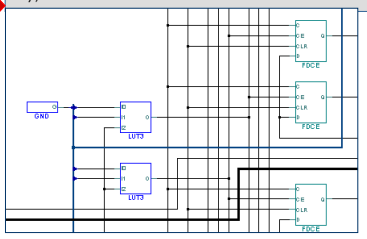
What is Synthesis?

Synthesis is the process of converting a design expressed in register-transfer level Hardware Description Language (HDL) into a netlist of gates or logic primitives that can be mapped directly into an ASIC or FPGA

```
always @(posedge clk or posedge rst)
   if (rst) begin
      {oe, we, data_oe, done} <= 4'b0000;
      state   <= IDLE;
   end
   else case (state)
      IDLE: begin
         {oe, we, data_oe, done} <= 4'b0000;
         if (req) begin  // access requested
            if (rd_wr) begin  // access is read
               oe    <= 1;    // enable drive data bus
               state <= R1;
            end
            else begin         // access is write
               data_oe <= 1;  // drive onto data bus
               state   <= W1;
            end
         end
      end
   W1: begin
      we    <= 1;    // begin write pulse
      state <= W2;
   end
```

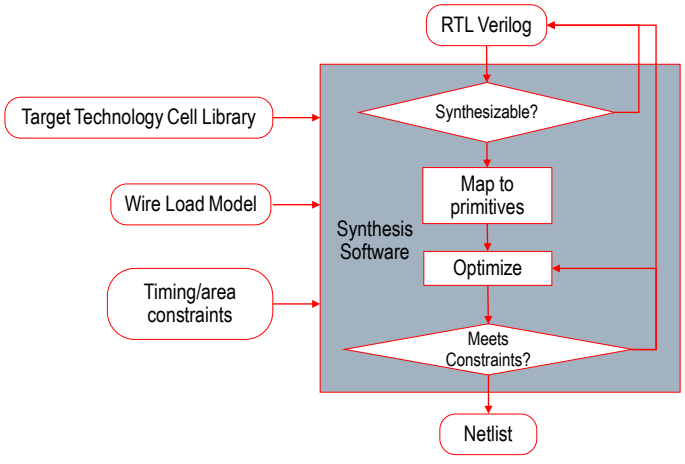
Synthesis

```
defparam \DIO/BTG_reg_state_i_state(1) .INIT = 1'b0;
 X_FF \DIO/BTG_reg_state_i_state(1)  (
    .I(\DIO/BTG_FSM_CB_state_i_FSM_c_state(2)/FROM ),
    .CE(VCC),
    .CLK(clk_int),
    .SET(GND),
    .RST(\DIO/BTG_FSM_CB_state_i_FSM_c_state(2)/FFX/RST ),
    .O(\DIO/BTG_FSM_CB_state_i_FSM_c_state [2])
 );
defparam i39d6x4.INIT = 16'h7FFF;
 X_LUT4 i39d6x4 (
    .ADR0(UtilCounter[0]),
    .ADR1(UtilCounter[3]),
    .ADR2(UtilCounter[2]),
    .ADR3(UtilCounter[1]),
    .O(\n39d6x4/GROM )
 );
```



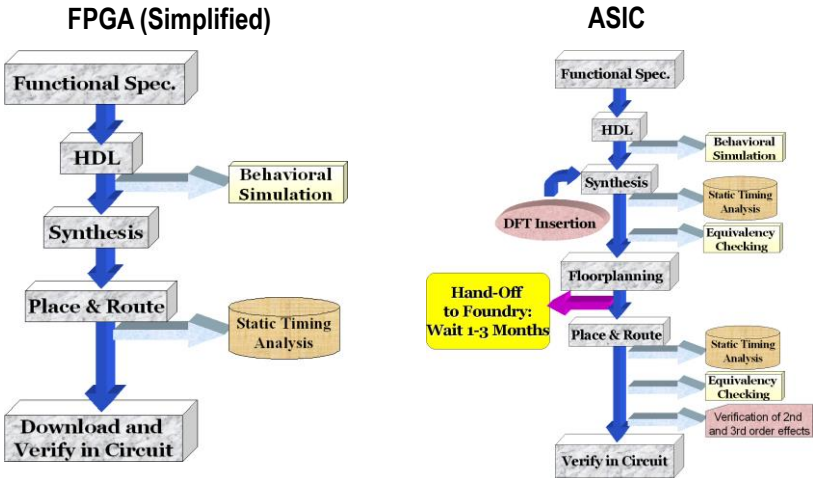
Portland State UNIVERSITY

Synthesis Flow



Portland State UNIVERSITY

Design Flow Comparison



Simulating Digital Systems

Source material drawn from:

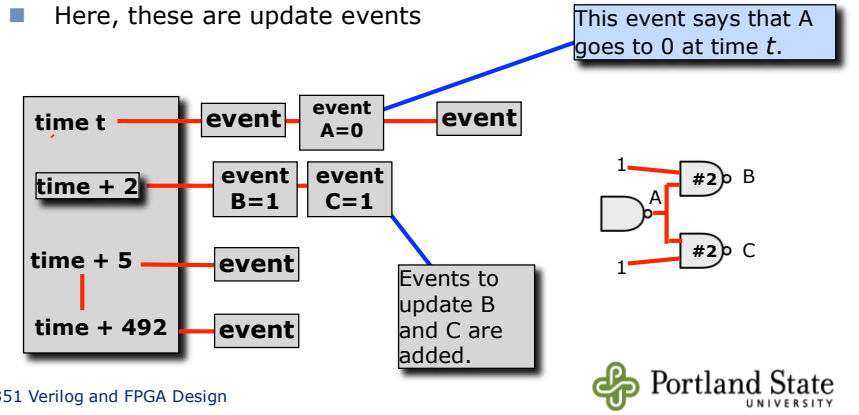
- *SystemVerilog for Design: Overview* by Donald Thomas
- *RTL Modeling with SystemVerilog for Simulation and Stimulus* by Stuart Sutherland

Some basic terms

- System (or model) state
  - Includes variables and things intended to become nets and registers
  - Every variable deep in instantiated modules is uniquely named (hierarchical naming)
  - Statically defined
- Events and their flavors
  - Events are tuples:
    - something to do, and
    - a time to do it
  - Update Event — a value-change scheduled to occur at a given time
    - e.g., b is set to 1 at time 15
  - Evaluation Event — (sometimes "execution event") a model (always, initial…) to resume executing at a given time

Event list

- A time-ordered list of events is maintained
  - All events for a given time are kept together in an event list ordered by time
  - Events are handled at the given time and possibly schedule their output to change at a later time (a new event)
  - Here, these are update events



Event regions

- Events in a begin-end statement group are scheduled into an event region and executed in the order in which the statements are listed
- Events from concurrent processesare scheduled into the event region in an arbitrary order chosen by the simulator
- Simulators will execute all scheduled events in a region before transitioning to the next region
- As events are processed, they are permanently removed from the event list
- Each region will be empty before simulation proceeds to the next region
- As events are processed in a later region, they can possibly schedule new events in a previous region
- The transition from one event region to the next is referred to as a delta. Each iteration through all event regions is referred to as a delta cycle

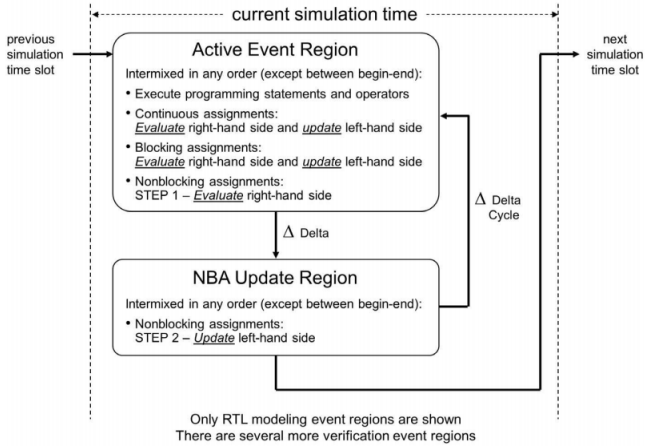
Sutherland Section 1.5.3.5.

Pseudo Code: Event-Driven Simulation


```
put all initial and always blocks in the event list
while something in time-ordered event list {
        advance simulation time to first event's time
        retrieve all events e for this time
        update state from all update-event values

        For each event e in arbitrary order {
            If it's an update event {
                follow fanout, evaluate gates there
                If an output changes
                    schedule update event for it
            }
            else // it's an evaluation event
                evaluate the model
        }
}
```

One traversal of the while loop is a *simulation cycle*.

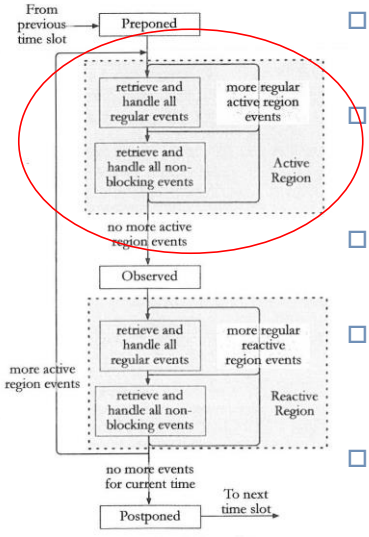
Event Scheduling



- Blocking assigns (`assign a = b`) – model combinational logic
- Non-blocking assigns (`a <= b`) – model sequential logic

Sutherland Figure 1-8

FYI: The "Whole" Simulation Kernel



- *Preponed* – time has been updated but event handling hasn't started. Values sampled for assertions
- *Active* – handling all regular and non-blocking events in the active region of event list. Events all have to do with design being modeled
- *Observed* – Assertions are executed using the values sampled in the *preponed* region
- *Reactive* – handling all regular and non-blocking events in the reactive region. Events all have to do with execution of program blocks
- *Postponed* – monitor events are handled. After this simulation time is advanced and the next time slot begins

Roy's ECE 571 slide – you won't be tested on this

Sutherland Example 1-7

```
////////////////////////////////////////////////////////////
// Design module with RTL model of a D-type register
////////////////////////////////////////////////////////////
//`begin_keywords "1800-2012"
module d_reg (
  input  logic       clk,
  input  logic [7:0] d,
  output logic [7:0] q
);

timeunit 1ns; timeprecision 1ns;

always_ff @(posedge clk)
    q <= d;

endmodule: d_reg
//`end_keywords
```


Sutherland Example 1-7 (cont'd)

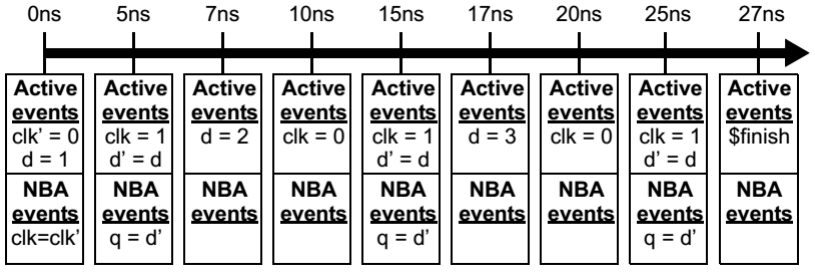
```
////////////////////////////////////////////////////////////
// Top module with clk oscillator
////////////////////////////////////////////////////////////
//`begin_keywords "1800-2012“
module top;
  timeunit 1ns; timeprecision 1ns;

  logic         clk;
  logic [7:0]   d;
  logic [7:0]   q;  test  i1 (.*);   // connect top module to test module

  d_reg i2 (.*);   // connect top module to d_reg module

  initial begin
    // clk oscillator
    clk <= 0;     // initialize clk at time 0
   forever #5 clk = ~clk;   // toggle clk every 5ns
endendmodule: top//`end_keywords
```

Sutherland Example 1-7 (cont'd)

```
////////////////////////////////////////////////////////////
// Test module with stimulus generator
////////////////////////////////////////////////////////////
//`begin_keywords "1800-2012“
module test (
  input  logic       clk,
  output logic [7:0] d,
  input  logic [7:0] q
);
  timeunit 1ns; timeprecision 1ns;

  initial begin
    d = 1;
    #7  d = 2;
    #10 d = 3;
    #10 $display("\n%m: No output-- checking that example compiles\n");
    #10 $finish;
endendmodule: test
//`end_keywords
```


Simulation timeline

Sutherland: Example 1_7



Sutherland Figure 1-9

Logic Simulation w/ QuestaSim

Example: ..\examples\ripple_carry_counter.pdf

QuestaSim Tutorial: ..\docs\Questa® SIM Tutorial.pdf

Using QuestaSim at PSU: ..\docs\UsingMentorQuestaAtPSU_R2_0.pdf

Some Definitions

- **Module** – the basic building block in Verilog
  - Can be an element or a collection of lower-level design blocks
  - Can provide abstraction…hides the details of the implementation (i.e. possible to modify block internals without affecting the rest of the design)
- **Instance** – Module is a template, instance is the actual object
- **Simulation** – The act of applying inputs to and monitoring the outputs from a Verilog model
- There are two distinct components of a simulation
  - **Design Block** – the implementation of the desired functionality
  - **Stimulus Block (Test Bench or Testbench)** – The inputs applied to the design block

Display Tasks

- Verilog supports system tasks for performing I/O, generating random numbers, initializing memory, etc.
  - Modeled after C library calls, but typically less flexible
  - All system tasks have the form `$<keyword>`
- Display tasks:
  - `$display()` – prints information on `stdout` w/ new line char
  - `$write()` – prints information on `stdout` w/o new line char
  - `$strobe()` – like `$display()` except prints at end of time step (e.g. all events have been processed)
  - `$monitor()` – continually monitors all of the arguments and prints whenever any of the signals being monitored changes

$display()

- `$display()` - displays values of variables or strings or expressions
  - *u*sage: `$display(format_desc, p1, p2, p3,…,pn);`
  - Format description is similar to C (`%d`, `%b`, etc. work)
  - Ex: `$display("ID of the port is %b", port_id);`
- You must explicitly tell Verilog to `$display` something…in other words, the `$display()` must be in a block of executable code
- Common problem: $display() is invoked in a situation where (simulation) time does not advance
  - Ex:

```
for (i = 0, i< 100, i = i+1)
    $display($time, x, y, z);
```

  - Simple fix: `#5 $display(…); // add some delay`

$monitor()

- `$monitor()` - displays values of variables or strings or expressions whenever their value(s) change
  - *u*sage: `$monitor(p1, p2, p3,…,pn);`
  - All of the variables in the list are displayed whenever one or more of them change
  - Ex:

```
$monitor($time, "clock = %b, reset = %b", clock, reset);
```

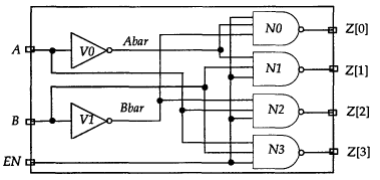
  - `$monitoron, $monitoroff` enable and disable monitoring

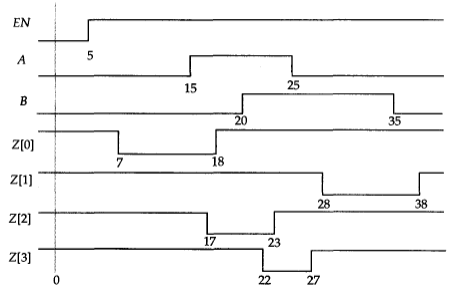
Delays

- SystemVerilog simulators model the behavior of a digital system over time and support delays (specified in time units)
  - `assign #2 sum = a ^ b  // #2 refers to time units`
  - Behavior: `sum` is assigned to `a ^ b` two time units after the current time whenever either `a` or `b` or both change
- You can associate time units to "physical" time using the `time unit and timeprecison statements`
  - `ex: timeunit 1ns; timeprecision 100ps` (or `timeunit 1ns/100ps`)
    - 1 time unit = 1ns, precision is 100ps (e.g. all delays rounded to 0.1ns)
  - Normally put into each module but only useful for simulation…synthesis ignores delays in the code because delays are technology-specific
  - No default defined in the spec…up to the vendor
- There are several ways to assign delays and they are dependent on context…more on this later

Delays (cont'd)


```
module decoder2x4 (
 input logic A, B, EN,
 output logic [0:3] Z
);

timeunit 1ns/1ns;

logic Abar, Bbar;

assign #1 Abar = ~A;
assign #1 Bbar = ~B

assign #2 Z[0] = ~(Abar & Bbar & EN);
assign #2 Z[1] = ~(Abar & B & EN);
assign #2 Z[2] = ~(A & Bbar & EN);
assign #2 Z[3] = ~(A & B & EN);

endmodule : decoder2x4
```

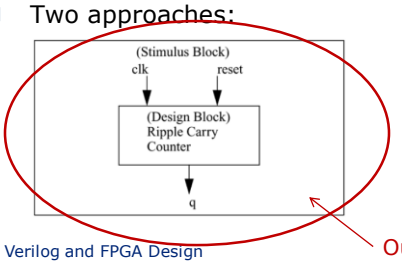


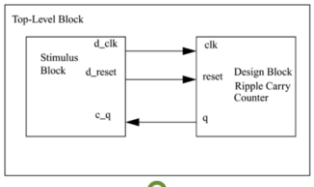


Original example: Bhasker, Verilog HDL Primer, 3rd Edition

Test Benches

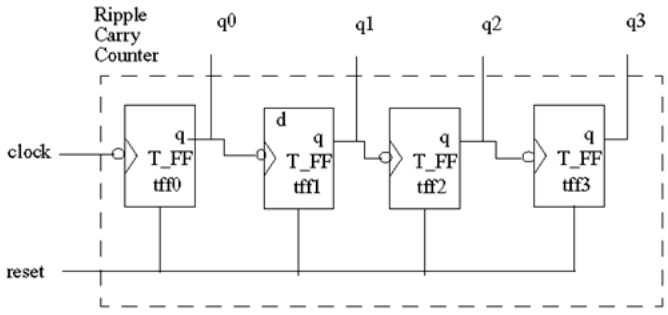
- The functionality of a design can be tested by applying stimulus and checking for a correct result
  - The stimulus block is commonly called a Test Bench
  - Test benches can be written in Verilog using the full feature set of the language
  - You can build one or more test benches, each focusing on a specific piece of functionality
  - Obvious…but worth saying – The correctness of the design depends heavily on the effectiveness of the test cases
  - Two approaches:

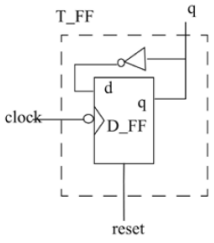




Our Example

Example: 4-bit Ripple Carry Counter





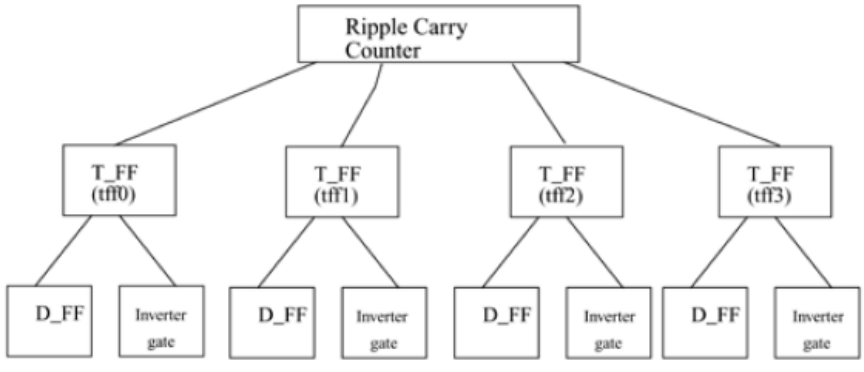
reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0

Specification:

Increments count (*q[0:3]*) by 1 on every negative edge of *clock*; count is set 4'b0000 when *reset* is asserted

Source: Palnitkar, Verilog HDL, 2nd Edition

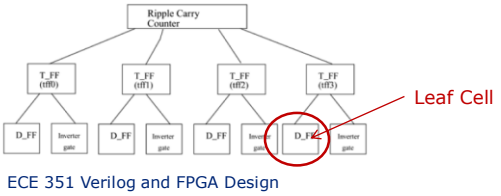
Hierarchy



Source: Palnitkar, Verilog HDL, 2nd Edition

Design Block – D Flip-Flop Sub-block

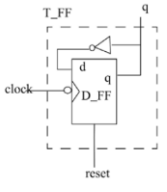

```
// D flip-flop with asynchronous reset – RTL model
module DFF(
  output logic q,
  input logic d, clk, reset
);

always_ff @(negedge clk or posedge reset) begin
  if (reset)
    q <= 1'b0;
  else
    q <= d;
end

endmodule : DFF
```

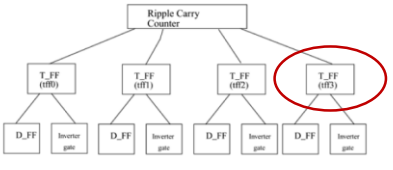


Portland State UNIVERSITY

Design Block – TFF Sub-block


```
// Toggle Flip-flop
module TFF (
  output logic q,
  input logic clk, reset
);

logic d;

// instantiate a D flip-flop
DFF dff0(.q, .d, .clk, .reset);

// create the toggle flip-flop by driving input with inverted output
not n1(d, q); // not is a Verilog provided primitive.

endmodule : TFF
```

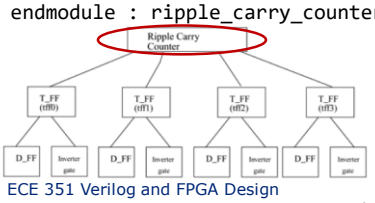


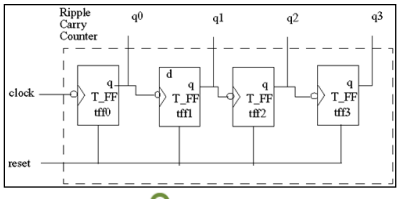


Portland State UNIVERSITY

Design Block – Top-Level

```
// Top level module for 4-bit Ripple Carry Counter
module ripple_carry_counter (
  output [3:0] count,
  input clk, reset
);
// internal signals
logic q0, q1, q2, q3;

//instantiate 4 toggle flip-flops
TFF tff0(q0,clk, reset);
TFF tff1(q1,q0, reset);
TFF tff2(q2,q1, reset);
TFF tff3(q3,q2, reset);
assign count = {q3, q2, q1, q0};

endmodule : ripple_carry_counter
```





Source: Palnitkar, Verilog HDL, 2nd Edition

Ex: Stimulus Block for Ripple Carry Counter

```
// Stimulus module to test the ripple carry counter
// Toggle reset and watch it count
// NOTE: There are no external ports - typical for test benches
module stimulus;
  logic clk;
  logic reset;
  logic[3:0] q;

  // instantiate the design under test (DUT)
  ripple_carry_counter DUT(.*);

  // Create the clk signal that drives the design block.
  initial begin
    clk = 1'b0;
  end // initial block

  always begin
    #5 clk = ~clk;
  end // always block

  …
```


Source: Palnitkar, Verilog HDL, 2nd Edition

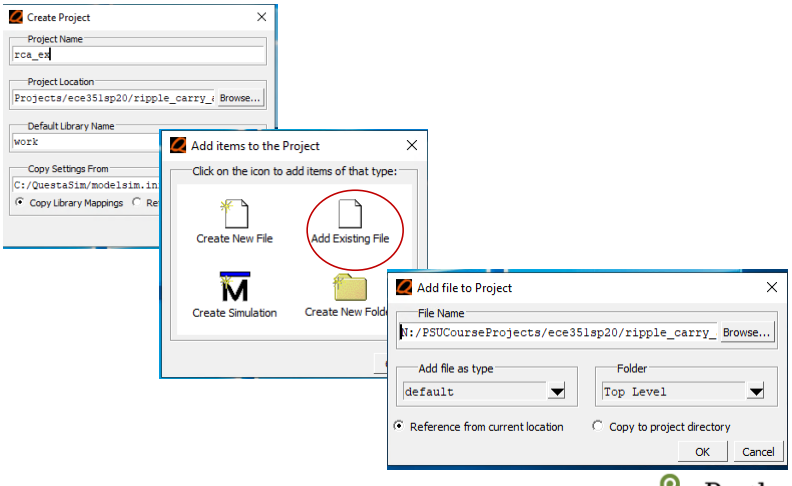
Ex: Stimulus Block for Ripple Carry Counter


```
// Monitor the outputs
  initial begin
    $monitor($time, " Output q = %d",  q);
  end

  // Control the reset signal that drives the design block
  initial begin : test_vectors
    #0  reset = 1'b1;
    #15   reset = 1'b0;
    #180reset = 1'b1;
    #10   reset = 1'b0;
    #20   $stop;
  end : test_vectors
endmodule : stimulus
```


Using the QuestaSim GUI

Create a new QuestaSim Project and add the source files

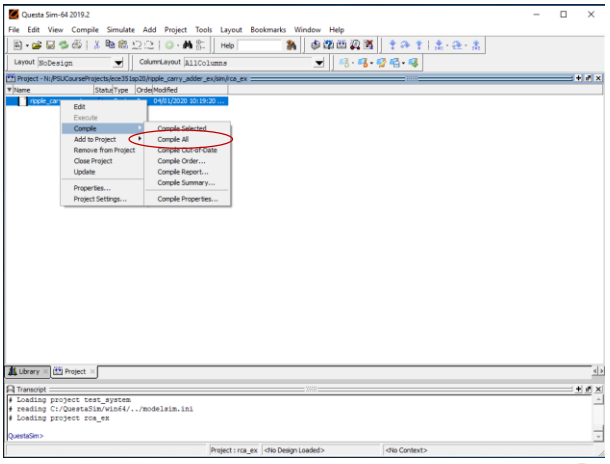
`File/New Project…`



Using the QuestaSim GUI (cont'd)

Compile the source files

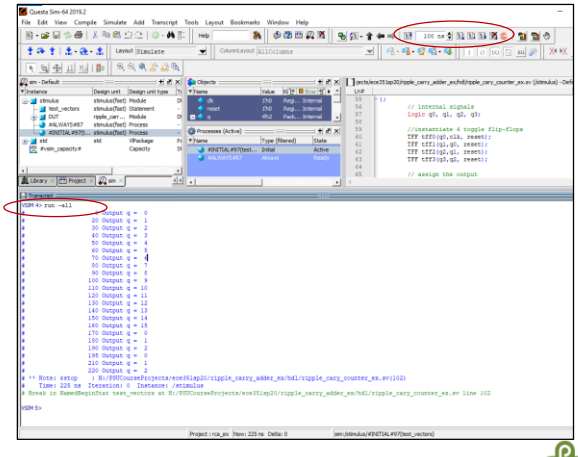
`Compile/Compile All`



Using the QuestaSim GUI (cont'd)

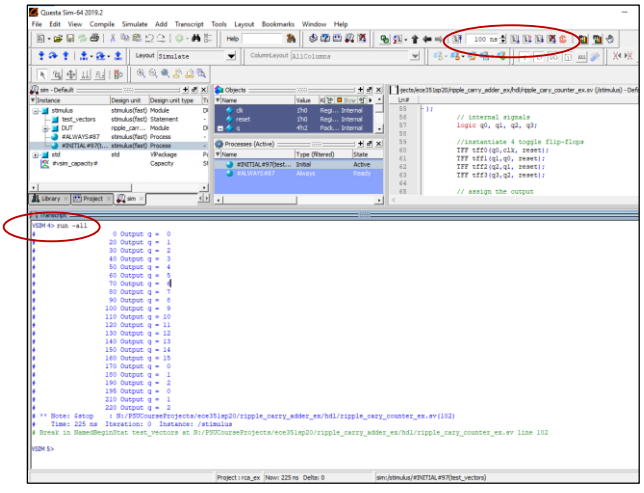
Load the simulation model

`Select Library tab, open work directory, select top module, right click and select Simulate`

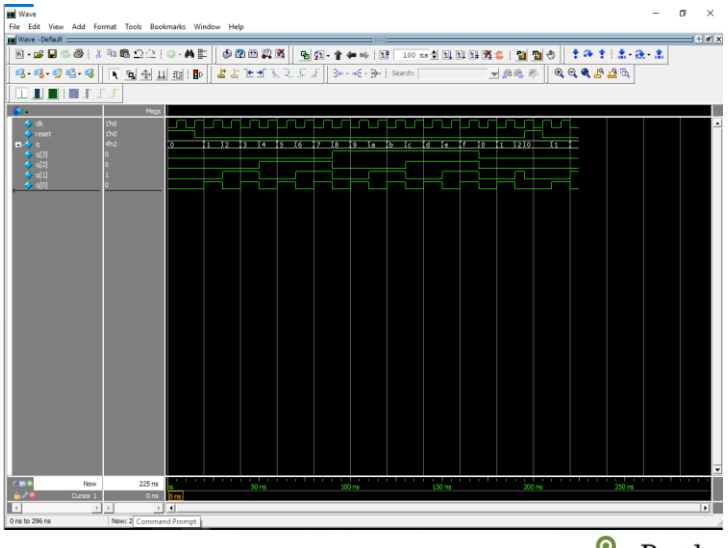


Using the QuestaSim GUI (cont'd)

Run the simulation



Using the QuestaSim GUI (cont'd)



Next Time

- Topics:
  - SystemVerilog language rules
  - Modules, ports, and hierarchy
  - Literals, nets, and vars
- You should:
  - Read Sutherland Ch 3
- Homework, projects and quizzes
  - Homework #1 will be assigned Tue, 14-Apr