FPGA Programming

Class 3: HDL

September 13, 2017
Louis W. Giokas

This Week's Agenda

Monday	FPGA Device Description
Tuesday	Design Flow
Wednesday	HDL
Thursday	Synthesis and Layout
Friday	Programming the Chip

Course Description

We start with an introduction to the class of devices called Field Programmable Gate Arrays (FPGAs). The layout and design of several types and critical parameters will be described and discussed. It is important to understand the way the device is constructed to develop effective algorithms.

The device we will be using this week will be the Microsemi IGLOO2. We will also discuss other devices and their structure.

We will introduce two common Hardware Description Languages (HDL), but give examples in one (Verilog).

Today’s Agenda

- Overview
- Tutorial
- Examples
- Conclusion/Next Class

Overview

- A Hardware Description Language (HDL) is used to provide a precise description of the structure and behavior of electronic circuits in a textual form
- We use HDL to describe the functions to be implemented in the FPGA
- We also use it to describe all the control lines required by our application

Overview

- HDL History
  - First modern HDL was Verilog (mid-1980s)
  - VHDL was introduced by the US Department of Defense (DoD) in the late 1980s. It is based on the Ada language
  - Other HDLs existed, but these are the two used today
    - Example: System-C
  - System Verilog is used for verification
    - It is an extension of Verilog

Overview

- HDL Standards
  - HDLs in use have been made into IEEE standards
  - Verilog: IEEE 1364-2005
  - System Verilog: IEEE 1800-2012
  - VHDL: IEEE 1076-2008
  - Standardization ensures that code can be processed by compilers from multiple vendors
    - Also allows code developed separately to be combined

Overview

- The reason for the use of HDLs is that circuits began to get so complex that creating and maintaining circuit diagrams became too difficult
- Using concepts from computer engineering concepts such as layering and data hiding increased the size of circuitry that could be specified
- Still a complex process, but with good tools, manageable

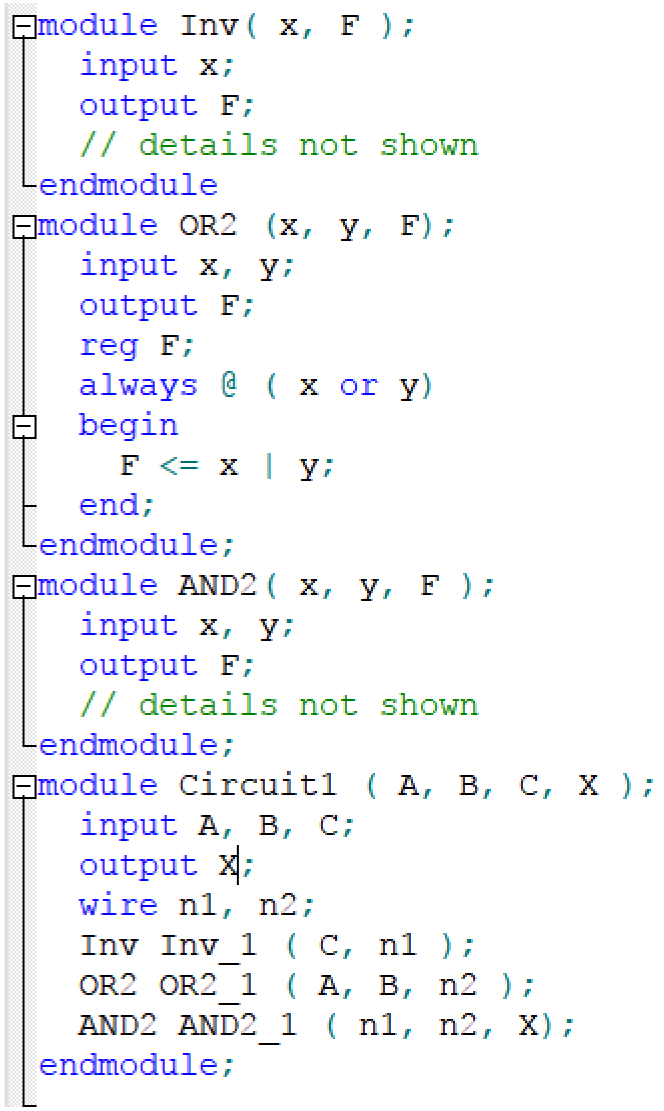
Tutorial

- We will go over a simple circuit that combines a few types of gates just to get the idea
- This is not a comprehensive example or tutorial, just an introduction
- Just want to highlight some of the syntax
- Out little example could be used a door opener

Tutorial

- The Verilog language allow the specification of logic elements and the connections between them
  - We specify inputs and outputs
  - We specify wires within the circuit
  - We specify structure of the circuit
  - We specify behavior of the circuit elements and overall circuit

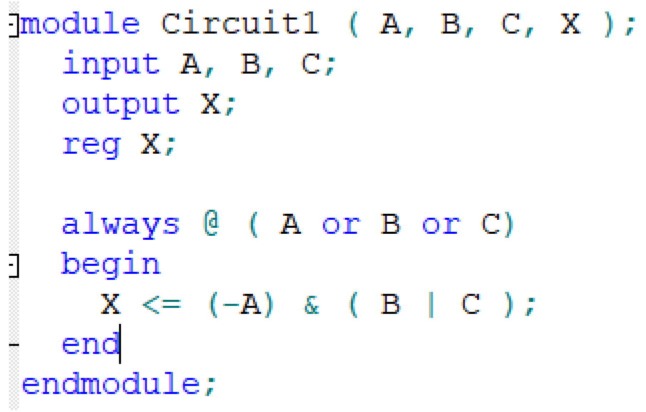
Tutorial



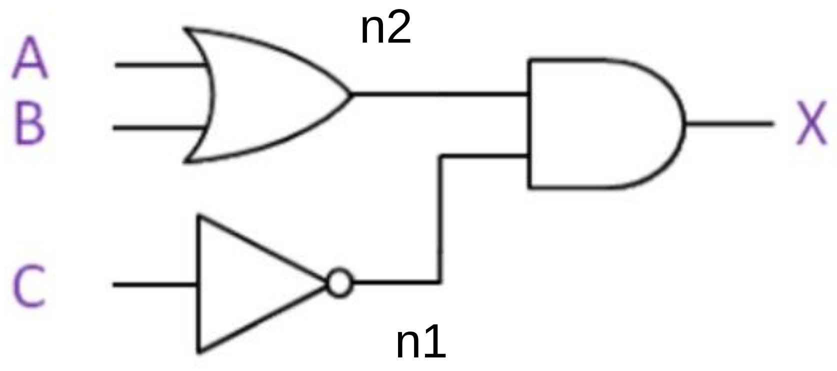
```
module Inv( x, F );
   input x;
   output F;
   // details not shown
endmodule
module OR2 (x, y, F);
   input x, y;
   output F;
   reg F;
   always @ ( x or y)
   begin
      F <= x | y;
   end;
endmodule;
module AND2( x, y, F );
   input x, y;
   output F;
   // details not shown
endmodule;
module Circuit1 ( A, B, C, X );
   input A, B, C;
   output X;
   wire n1, n2;
   Inv Inv_1 ( C, n1 );
   OR2 OR2_1 ( A, B, n2 );
   AND2 AND2_1 ( n1, n2, X);
endmodule;
```

Behavior

Structure



```
module Circuit1 ( A, B, C, X );
   input A, B, C;
   output X;
   reg X;

   always @ ( A or B or C)
   begin
      X <= (~A) & ( B | C );
   end
endmodule;
```



Presented by:

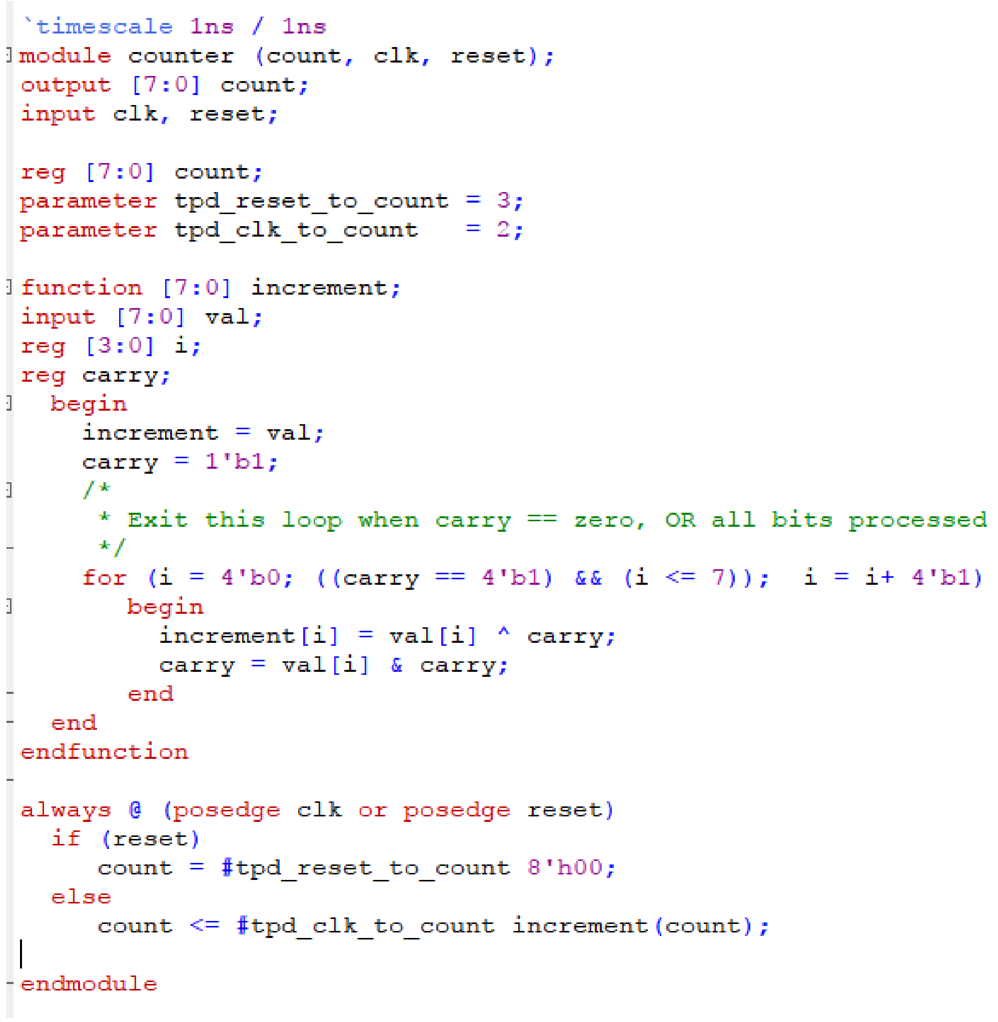
Tutorial

- Notice that we define the primitive circuit types and their behaviors as types
  - Generally these will be in a library
  - We then instantiate them, possibly many times
- We also have to define all the signals, inputs and wires as types, then we use them in the structural description
  - Built in types, such as *wire*, are available

Examples

- In the following we will go through several code snippets
  - The first couple will be complete routines
  - Then we will look at code from a FFT implementation
- The goal is to show the power, as well as the complexity
  - Code is color coded, showing keywords and user defined symbols

Examples

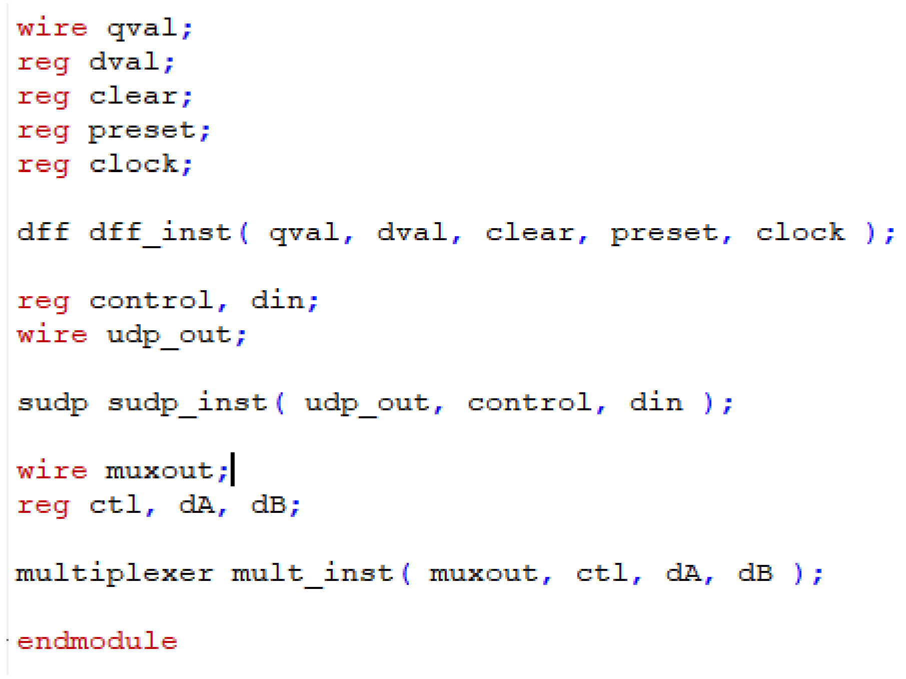


```
`timescale 1ns / 1ns
module counter (count, clk, reset);
output [7:0] count;
input clk, reset;

reg [7:0] count;
parameter tpd_reset_to_count = 3;
parameter tpd_clk_to_count   = 2;

function [7:0] increment;
input [7:0] val;
reg [3:0] i;
reg carry;
  begin
    increment = val;
    carry = 1'b1;
    /*
     * Exit this loop when carry == zero, OR all bits processed
     */
    for (i = 4'b0; ((carry == 4'b1) && (i <= 7)); i = i+ 4'b1)
       begin
         increment[i] = val[i] ^ carry;
         carry = val[i] & carry;
       end
  end
endfunction

always @ (posedge clk or posedge reset)
  if (reset)
     count = #tpd_reset_to_count 8'h00;
  else
     count <= #tpd_clk_to_count increment(count);

endmodule
```

A simple counter. Note the multibit entities.

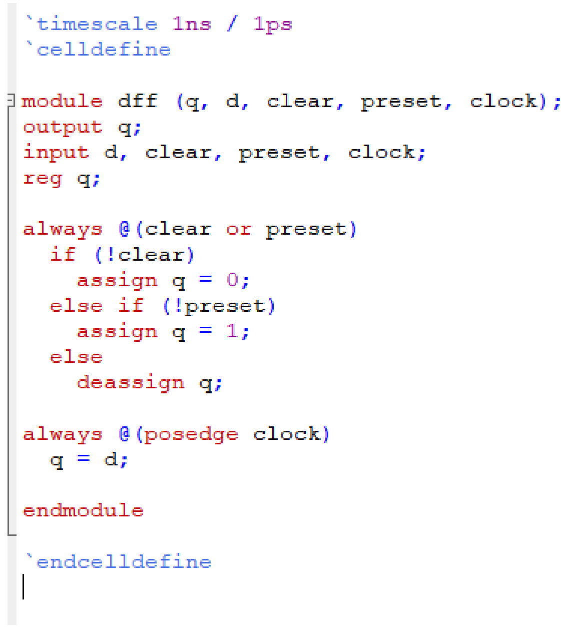
Presented by:

Examples



```
wire qval;
reg dval;
reg clear;
reg preset;
reg clock;

dff dff_inst( qval, dval, clear, preset, clock );

reg control, din;
wire udp_out;

sudp sudp_inst( udp_out, control, din );

wire muxout;
reg ctl, dA, dB;

multiplexer mult_inst( muxout, ctl, dA, dB );

endmodule
```

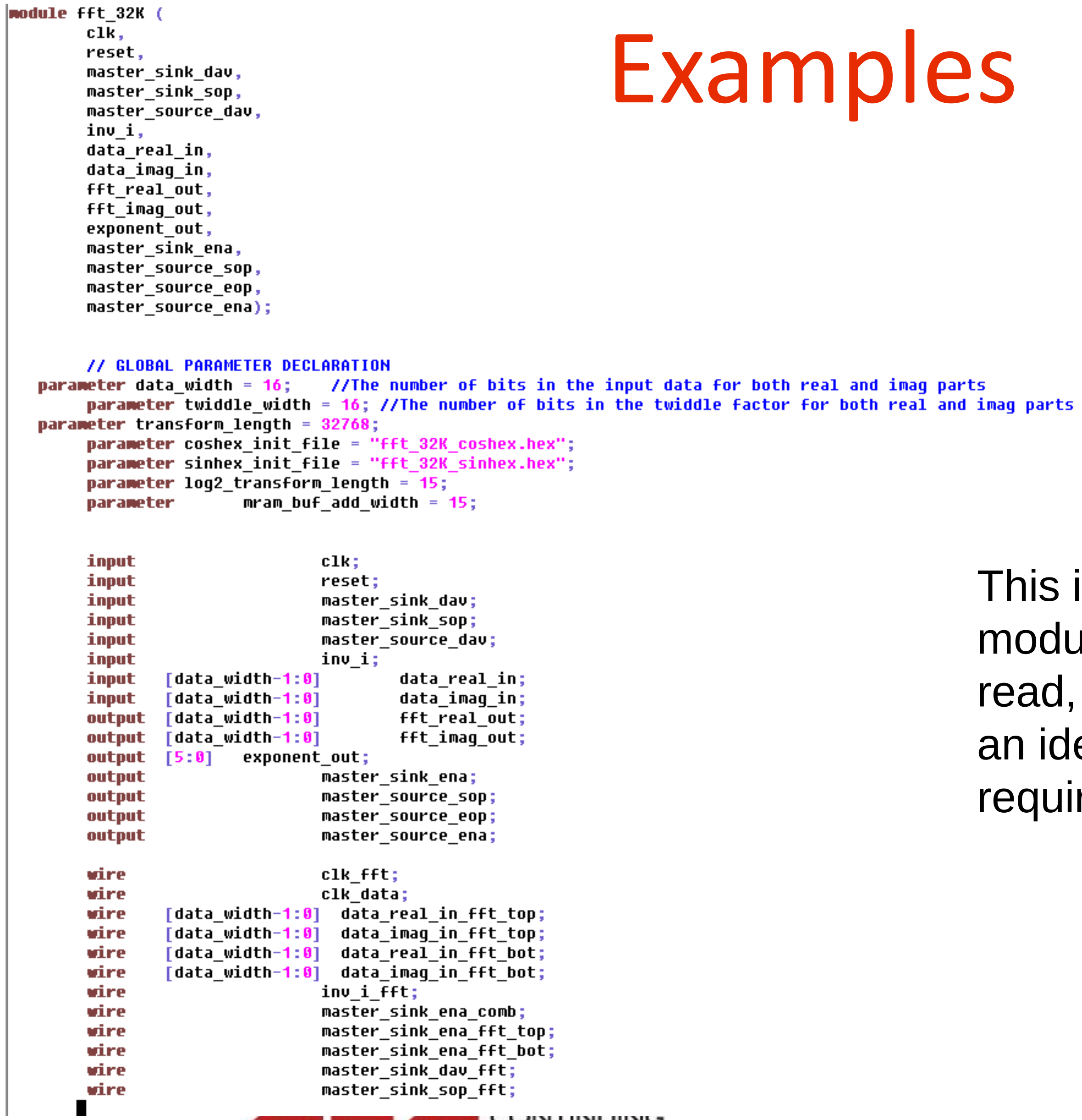
Shows use of clock signals. Below is the behavior of the dff type used in the main module.



```
`timescale 1ns / 1ps
`celldefine

module dff (q, d, clear, preset, clock);
output q;
input d, clear, preset, clock;
reg q;

always @(clear or preset)
  if (!clear)
    assign q = 0;
  else if (!preset)
    assign q = 1;
  else
    deassign q;

always @(posedge clock)
  q = d;

endmodule

`endcelldefine
```

Presented by:

Examples



```
module fft_32K (
        clk,
        reset,
        master_sink_dav,
        master_sink_sop,
        master_source_dav,
        inv_i,
        data_real_in,
        data_imag_in,
        fft_real_out,
        fft_imag_out,
        exponent_out,
        master_sink_ena,
        master_source_sop,
        master_source_eop,
        master_source_ena);


        // GLOBAL PARAMETER DECLARATION
   parameter data_width = 16;    //The number of bits in the input data for both real and imag parts
        parameter twiddle_width = 16; //The number of bits in the twiddle factor for both real and imag parts
   parameter transform_length = 32768;
        parameter coshex_init_file = "fft_32K_coshex.hex";
        parameter sinhex_init_file = "fft_32K_sinhex.hex";
        parameter log2_transform_length = 15;
        parameter       mram_buf_add_width = 15;


        input                   clk;
        input                   reset;
        input                   master_sink_dav;
        input                   master_sink_sop;
        input                   master_source_dav;
        input                   inv_i;
        input   [data_width-1:0]        data_real_in;
        input   [data_width-1:0]        data_imag_in;
        output  [data_width-1:0]        fft_real_out;
        output  [data_width-1:0]        fft_imag_out;
        output  [5:0]   exponent_out;
        output                  master_sink_ena;
        output                  master_source_sop;
        output                  master_source_eop;
        output                  master_source_ena;

        wire                    clk_fft;
        wire                    clk_data;
        wire    [data_width-1:0]  data_real_in_fft_top;
        wire    [data_width-1:0]  data_imag_in_fft_top;
        wire    [data_width-1:0]  data_real_in_fft_bot;
        wire    [data_width-1:0]  data_imag_in_fft_bot;
        wire                    inv_i_fft;
        wire                    master_sink_ena_comb;
        wire                    master_sink_ena_fft_top;
        wire                    master_sink_ena_fft_bot;
        wire                    master_sink_dav_fft;
        wire                    master_sink_sop_fft;
```

This is the beginning of a FFT module. I know it is hard to read, but I wanted to give you an idea of the amount of detail required.

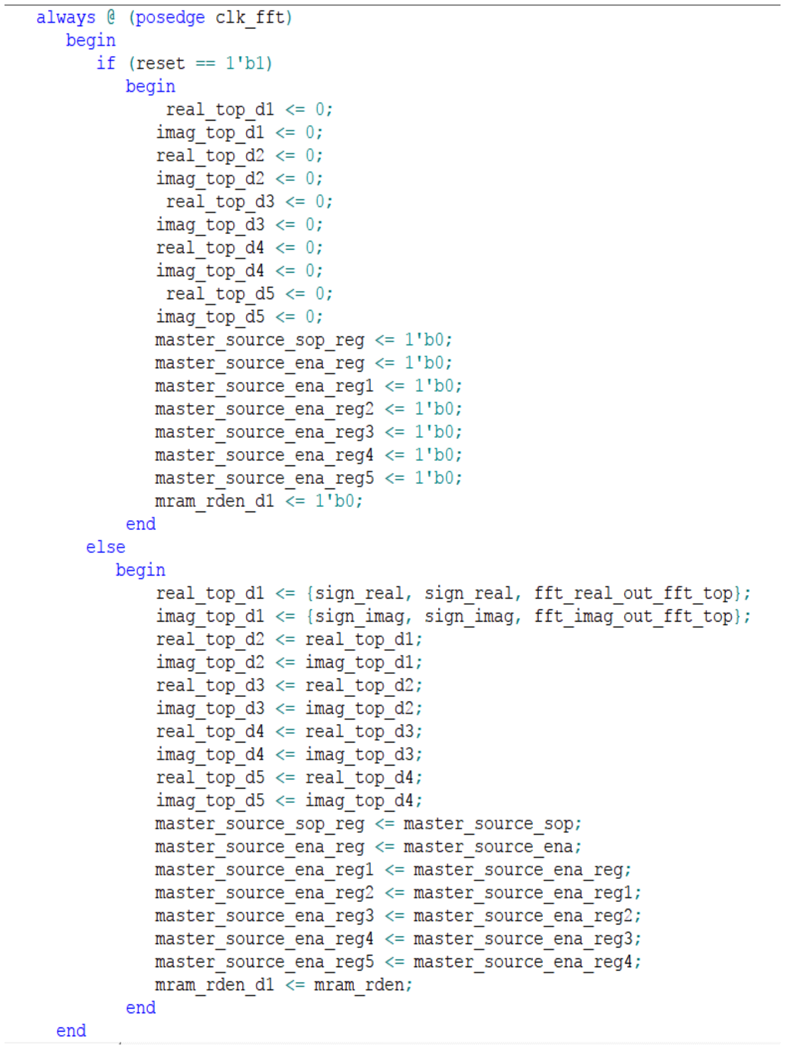
Presented by:

Examples

```
always @ (posedge clk_data_out)
 begin
   if(reset == 1'b1)
     begin
       tf_sp_d1 <= 1'b0;
       tf_sp_d2 <= 1'b0;
       cnt_dir_d1 <= 1'b0;
       cnt_dir_d2 <= 1'b0;
       count_reg <= 0;
      end
   else
     begin
       tf_sp_d1 <= reset_cnt;
       tf_sp_d2 <= tf_sp_d1;
       cnt_dir_d1 <= cnt_dir;
       cnt_dir_d2 <= cnt_dir_d1;
       count_reg <= count;
     end
 end

always @ (posedge clk_fft)
  begin
    if (reset == 1'b1)
       cnt_en_d1 <= 0;
    else
       cnt_en_d1 <= cnt_en;
    end
```

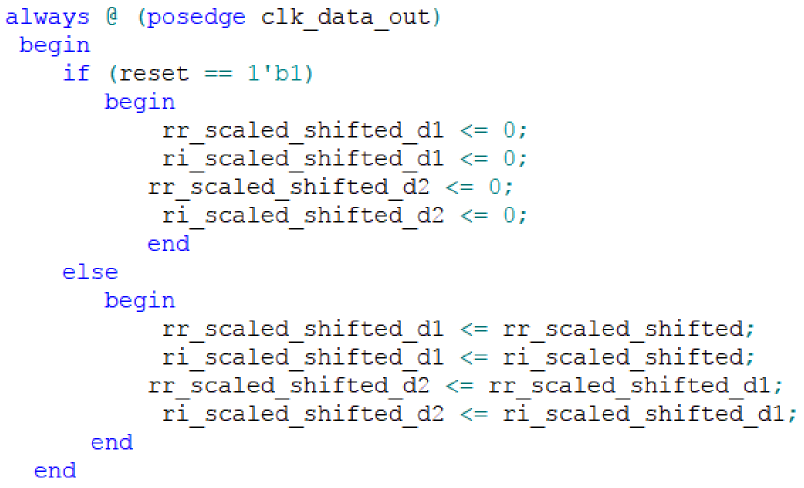
In this code snippet we are manipulating the bits of the real and imaginary parts. Note that there is lots of definition of signals preceding this.

Examples



```
  always @ (posedge clk_fft)
    begin
      if (reset == 1'b1)
        begin
            real_top_d1 <= 0;
           imag_top_d1 <= 0;
           real_top_d2 <= 0;
           imag_top_d2 <= 0;
            real_top_d3 <= 0;
           imag_top_d3 <= 0;
           real_top_d4 <= 0;
           imag_top_d4 <= 0;
            real_top_d5 <= 0;
           imag_top_d5 <= 0;
           master_source_sop_reg <= 1'b0;
           master_source_ena_reg <= 1'b0;
           master_source_ena_reg1 <= 1'b0;
           master_source_ena_reg2 <= 1'b0;
           master_source_ena_reg3 <= 1'b0;
           master_source_ena_reg4 <= 1'b0;
           master_source_ena_reg5 <= 1'b0;
           mram_rden_d1 <= 1'b0;
        end
     else
       begin
           real_top_d1 <= {sign_real, sign_real, fft_real_out_fft_top};
           imag_top_d1 <= {sign_imag, sign_imag, fft_imag_out_fft_top};
           real_top_d2 <= real_top_d1;
           imag_top_d2 <= imag_top_d1;
           real_top_d3 <= real_top_d2;
           imag_top_d3 <= imag_top_d2;
           real_top_d4 <= real_top_d3;
           imag_top_d4 <= imag_top_d3;
           real_top_d5 <= real_top_d4;
           imag_top_d5 <= imag_top_d4;
           master_source_sop_reg <= master_source_sop;
           master_source_ena_reg <= master_source_ena;
           master_source_ena_reg1 <= master_source_ena_reg;
           master_source_ena_reg2 <= master_source_ena_reg1;
           master_source_ena_reg3 <= master_source_ena_reg2;
           master_source_ena_reg4 <= master_source_ena_reg3;
           master_source_ena_reg5 <= master_source_ena_reg4;
           mram_rden_d1 <= mram_rden;
        end
   end
```

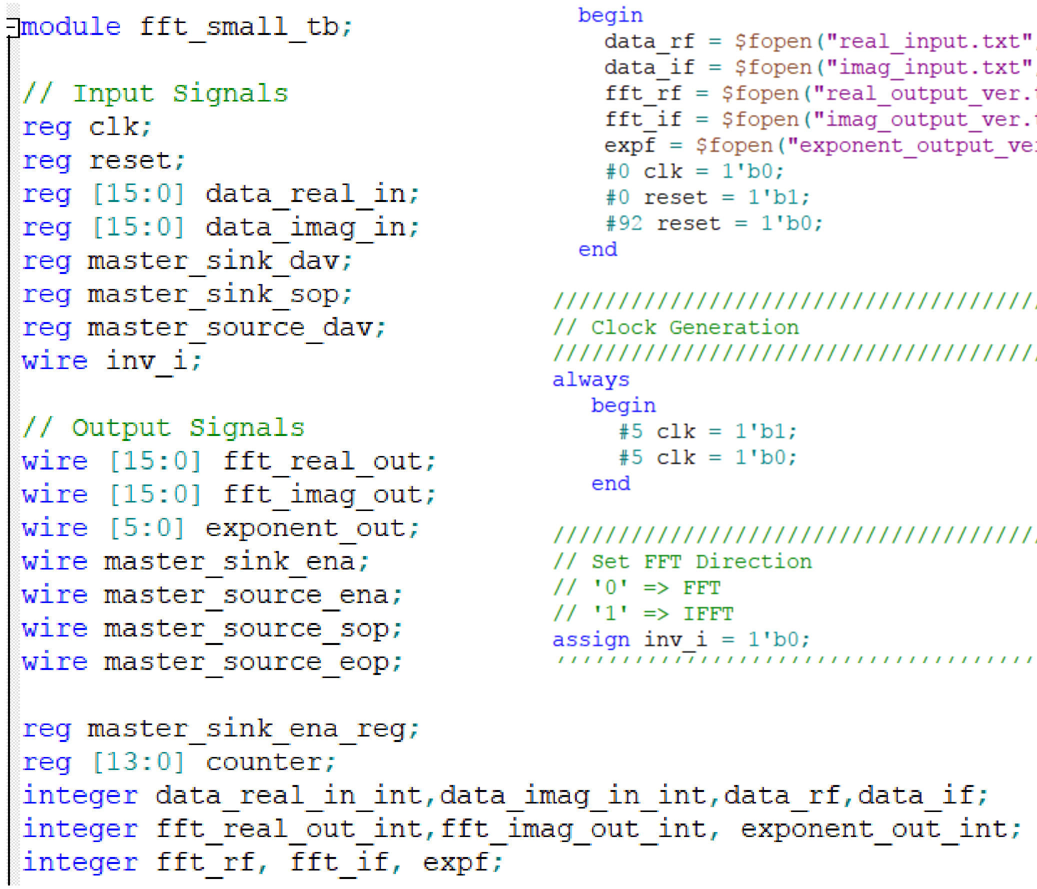
In this module we are combining the final results



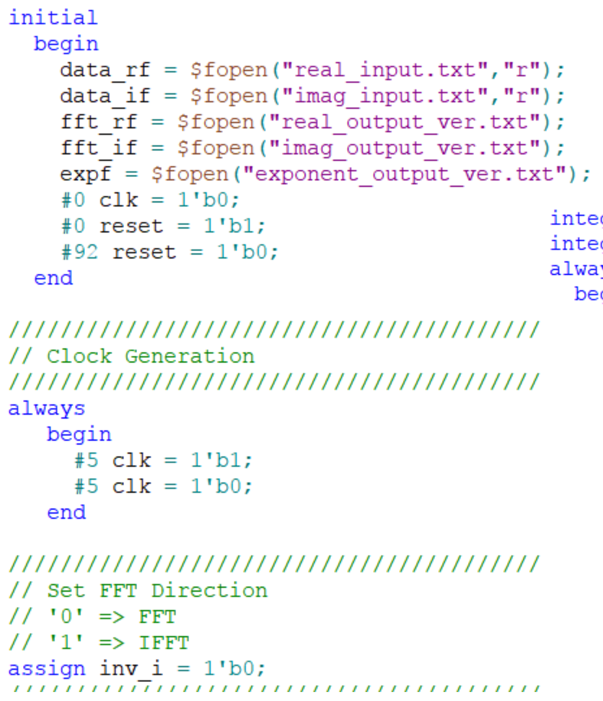
```
always @ (posedge clk_data_out)
 begin
    if (reset == 1'b1)
       begin
           rr_scaled_shifted_d1 <= 0;
           ri_scaled_shifted_d1 <= 0;
          rr_scaled_shifted_d2 <= 0;
           ri_scaled_shifted_d2 <= 0;
          end
    else
       begin
           rr_scaled_shifted_d1 <= rr_scaled_shifted;
           ri_scaled_shifted_d1 <= ri_scaled_shifted;
          rr_scaled_shifted_d2 <= rr_scaled_shifted_d1;
           ri_scaled_shifted_d2 <= ri_scaled_shifted_d1;
      end
  end
```

Examples

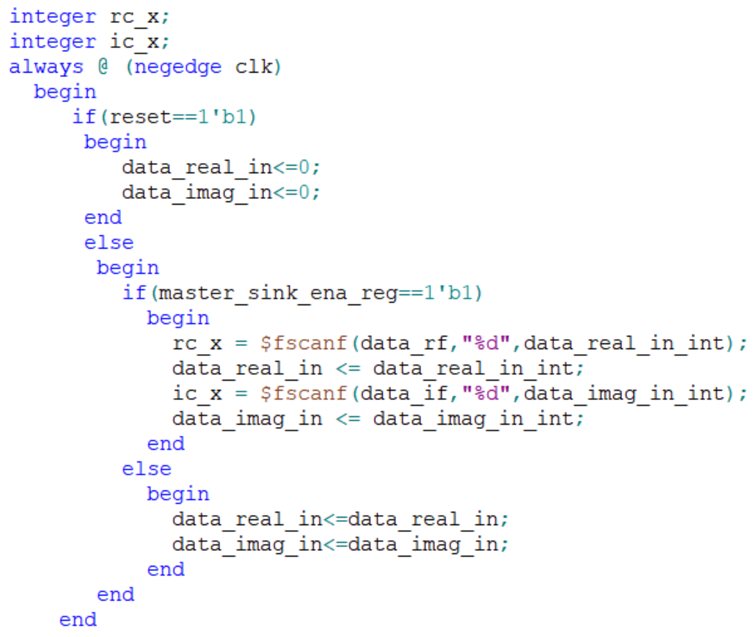
Finally, parts of the test bench



```
module fft_small_tb;

// Input Signals
reg clk;
reg reset;
reg [15:0] data_real_in;
reg [15:0] data_imag_in;
reg master_sink_dav;
reg master_sink_sop;
reg master_source_dav;
wire inv_i;

// Output Signals
wire [15:0] fft_real_out;
wire [15:0] fft_imag_out;
wire [5:0] exponent_out;
wire master_sink_ena;
wire master_source_ena;
wire master_source_sop;
wire master_source_eop;

reg master_sink_ena_reg;
reg [13:0] counter;
integer data_real_in_int,data_imag_in_int,data_rf,data_if;
integer fft_real_out_int,fft_imag_out_int, exponent_out_int;
integer fft_rf, fft_if, expf;
```



```
initial
  begin
    data_rf = $fopen("real_input.txt","r");
    data_if = $fopen("imag_input.txt","r");
    fft_rf = $fopen("real_output_ver.txt");
    fft_if = $fopen("imag_output_ver.txt");
    expf = $fopen("exponent_output_ver.txt");
    #0 clk = 1'b0;
    #0 reset = 1'b1;
    #92 reset = 1'b0;
  end

/////////////////////////////////////////
// Clock Generation
/////////////////////////////////////////
always
   begin
     #5 clk = 1'b1;
     #5 clk = 1'b0;
   end

/////////////////////////////////////////
// Set FFT Direction
// '0' => FFT
// '1' => IFFT
assign inv_i = 1'b0;
/////////////////////////////////////////
```



```
integer rc_x;
integer ic_x;
always @ (negedge clk)
  begin
     if(reset==1'b1)
      begin
         data_real_in<=0;
         data_imag_in<=0;
      end
      else
       begin
         if(master_sink_ena_reg==1'b1)
           begin
             rc_x = $fscanf(data_rf,"%d",data_real_in_int);
             data_real_in <= data_real_in_int;
             ic_x = $fscanf(data_if,"%d",data_imag_in_int);
             data_imag_in <= data_imag_in_int;
           end
         else
           begin
             data_real_in<=data_real_in;
             data_imag_in<=data_imag_in;
           end
       end
   end
```

Presented by:

Conclusion/Next Class

- Today we have been introduced to the history and place of HDLs in the design process
- We have looked at a simple tutorial example to get a feel for the syntax
- We have "glanced" at some examples to get a feel for what is required to design systems in HDL
- Tomorrow we will look at the processes of synthesis (from HDL) and layout on the fabric