The quickest Verilog guide

Omar Eldash – EECE 371

Verilog starts with defining the black box of the system planned ahead

Module Block_Name (inputs and outputs);

Begin

.

.

.

Endmodule;

There is two ways to define inputs and outputs in the module

1- *Module Block_Name(in1,in2,in3, … out1, out2,out3,…)*
2- *Module Block_Name(input in1, input in2, input in3,…, output out1, output out2, output out3, …)*

After the module definition comes the definition of variables in a typical design flow. Variables can be defined as wires or registers (reg) where reg are used when there is a storage element needed. Wires are just like board level is a signal carrier.

wire w1;

wire [n-1:0] w2;

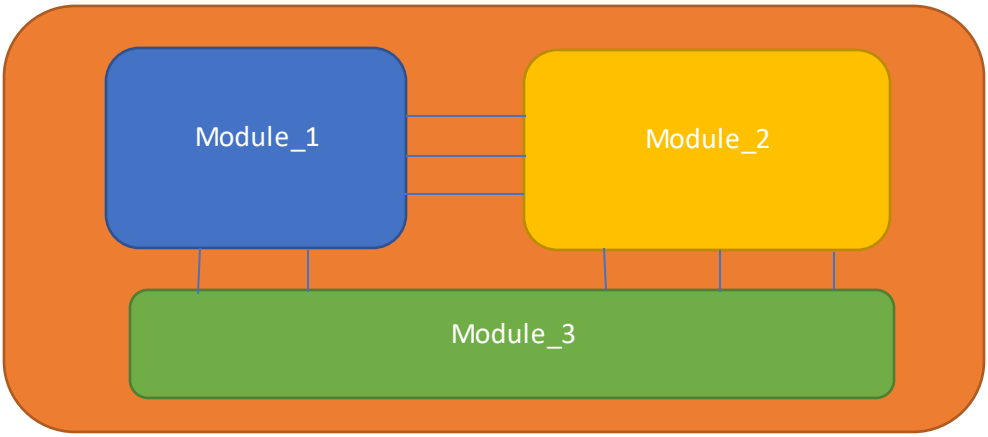
reg[n-1:0] r1;

Function Description

A traditional way to do functions is by using assign statement.

Assign some_port = Logic Function (^ xor, | or, & and) or mathematical function;

Using instantiation for hierarchical design (structural)



Module main_module(ports declaration);

Wire p0,p1,p2,p3,p4,p5,p6,p7,p8;

Module_1 U0(p1,p2,p3,p4,p5);

Module_2 U1(p1,p2,p3,p6,p7,p8);

Module_3 U2(p4,p5,p6,p7,p8);

The above approach, ports have to be announced in the same order they are mentioned originally in there module definition.

You can think of instantiation the same way you bring discrete components and place them on a breadboard. You need to know the pins which you connect between different ICs.

Another way to instantiate modules without the need of using in order declaration is to map local variables (wires and regs) to original ports.

Module_1 U0(.module_port1(local_wire1),.module_port2(local_wire2), etc.)

Instantiation is important for such hierarchical design as well as test bench design.

Assign statements and instantiation are both described in the main body of the verilog code after module definition and input/output/signals declarations.

Functional and behavioral description can be further described in always statements either as concurrent or sequential code. Concurrent statements are those meant to be running in the same time on hardware unlike typical sequential code.

Always statement:

This statement works by declaring

```
Always @ ()
Begin
.....
End
```

Whatever comes after @ is called the sensitivity list for the always block and it defines that the function will be only executed/implemented in a way that it's triggered by this list of signals.

Multiplexer example

```
Always @ (in0 or in1 or sel)
Begin
If (sel ==0) begin
        Y = in0;
End else begin
        Y= in1;
End
end
```

The above statement use what is called blocking assignment using (=). Blocking assignments work sequentially. Another type is non-blocking assignment and represents concurrent execution for logic gates.

To use always statements inferring a clocked system the statement is declared as follows:

```
Always @(posedge clk) ……
```

This basically infers a flip-flop or a pipelined clocked system.

Example Codes:

Full Adder

```
module full_adder
  (
   i_bit1,
   i_bit2,
   i_carry,
   o_sum,
   o_carry
   );

  input  i_bit1;
  input  i_bit2;
  input  i_carry;
  output o_sum;
  output o_carry;

  wire   w_WIRE_1;
  wire   w_WIRE_2;
  wire   w_WIRE_3;

  assign w_WIRE_1 = i_bit1 ^ i_bit2;
  assign w_WIRE_2 = w_WIRE_1 & i_carry;
  assign w_WIRE_3 = i_bit1 & i_bit2;

  assign o_sum   = w_WIRE_1 ^ i_carry;
  assign o_carry = w_WIRE_2 | w_WIRE_3;


  // FYI: Code above using wires will produce the same results as:
  // assign o_sum   = i_bit1 ^ i_bit2 ^ i_carry;
  // assign o_carry = (i_bit1 ^ i_bit2) & i_carry) | (i_bit1 & i_bit2);

  // Wires are just used to be explicit.

endmodule // full_adder
```

Ripple Carry Adder

```
`include "full_adder.v"

module ripple_carry_adder
  #(parameter WIDTH)
  (
   input [WIDTH-1:0] i_add_term1,
   input [WIDTH-1:0] i_add_term2,
   output [WIDTH:0]  o_result
   );

  wire [WIDTH:0]     w_CARRY;
  wire [WIDTH-1:0]   w_SUM;

  // No carry input on first full adder
  assign w_CARRY[0] = 1'b0;

  genvar             ii;
  generate
    for (ii=0; ii<WIDTH; ii=ii+1)
      begin
        full_adder full_adder_inst
            (
              .i_bit1(i_add_term1[ii]),
              .i_bit2(i_add_term2[ii]),
              .i_carry(w_CARRY[ii]),
              .o_sum(w_SUM[ii]),
              .o_carry(w_CARRY[ii+1])
              );
      end
  endgenerate

  assign o_result = {w_CARRY[WIDTH], w_SUM};   // Verilog Concatenation

endmodule // ripple_carry_adder
```

Testbench Example

```
`include "ripple_carry_adder.v"

module ripple_carry_adder_tb ();

  parameter WIDTH = 2;

  reg [WIDTH-1:0] r_ADD_1 = 0;
  reg [WIDTH-1:0] r_ADD_2 = 0;
  wire [WIDTH:0]  w_RESULT;

  ripple_carry_adder #(.WIDTH(WIDTH)) ripple_carry_inst
    (
     .i_add_term1(r_ADD_1),
     .i_add_term2(r_ADD_2),
     .o_result(w_RESULT)
     );

  initial
    begin
      #10;
      r_ADD_1 = 2'b00;
      r_ADD_2 = 2'b01;
      #10;
      r_ADD_1 = 2'b10;
      r_ADD_2 = 2'b01;
      #10;
      r_ADD_1 = 2'b01;
      r_ADD_2 = 2'b11;
      #10;
      r_ADD_1 = 2'b11;
      r_ADD_2 = 2'b11;
      #10;
    end

endmodule // ripple_carry_adder_tb
```

Shift Register

```
module shift_register_v(
input CLK,
input RST,
input DATA_IN,
output BIT_OUT,
output [7:0] BYTE_OUT
);
//————————————————-
// signal definitions
//————————————————-
//shift register signals
reg [7:0] bitShiftReg;
reg [7:0] byteShiftReg[11:0];
integer i;
//————————————————-
// shift register
//————————————————-
//shift register
always @(posedge CLK)
begin
//bit shift register
bitShiftReg <= {bitShiftReg[6:0],DATA_IN};
//byte shift register
byteShiftReg[0] <= bitShiftReg;
for(i=1;i<12;i=i+1)
byteShiftReg[i] <= byteShiftReg[i-1];
end
//————————————————-
// outputs
//————————————————-
//module output wires
assign BIT_OUT = bitShiftReg[7];
assign BYTE_OUT = byteShiftReg[11];
```

Serial to Parallel

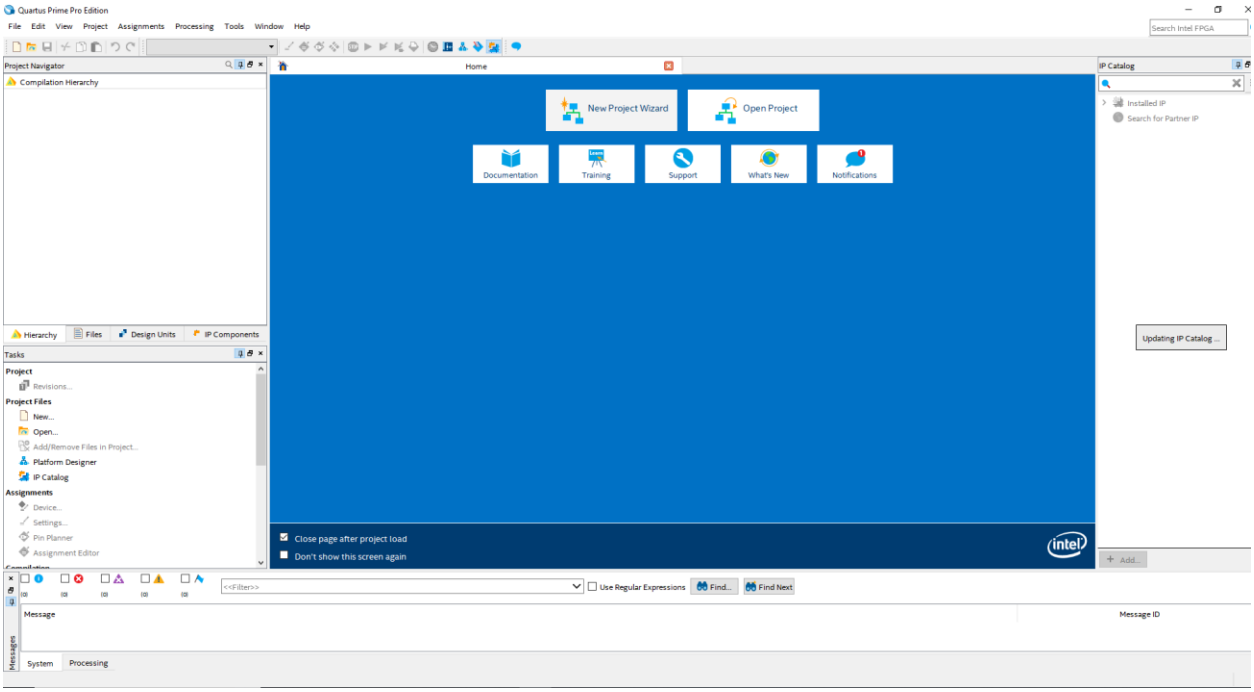
```
module deserial (clk, reset, serial_data_in,parallel_data_out);
input clk;
input reset;

input serial_data_in;
output reg [7:0] parallel_data_out;
always @ ( posedge clk or negedge reset)
begin
  if (!reset)
    parallel_data_out <= 8'b0;
  else
     parallel_data_out[7 :0] <= {parallel_data_out[6:0],serial_data_in};
end
endmodule
```

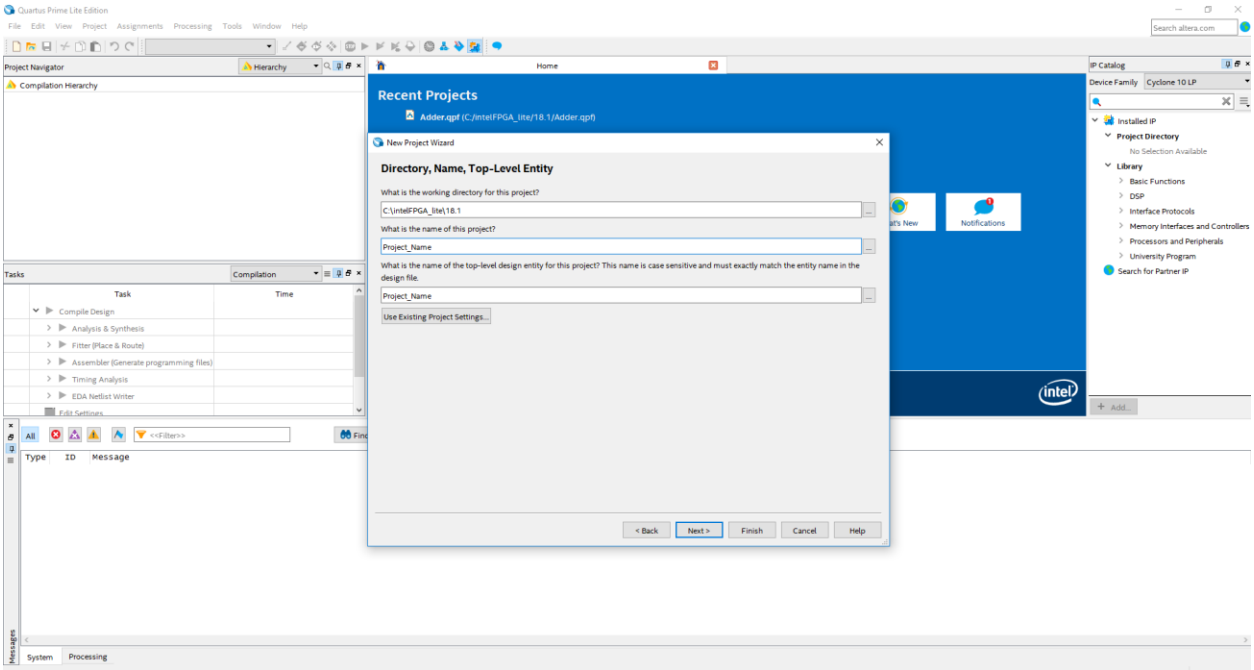
Parallel to Serial

```
module serial (clk, reset, loaden_o, serdes_factor,serial_data_out,parallel_data_in);
input clk;
input reset;
input loaden_o;
input serdes_factor;
input [7:0] parallel_data_in;
output reg serial_data_out;
reg [7:0] sft_reg;
reg [2:0] count_ps;
always @ ( posedge clk or negedge reset)
begin
  if (!reset)
  begin
  sft_reg  =8'b0;
  count_ps = 'b0;
  end
else
begin  if(loaden_o == 1'b1)
      sft_reg = parallel_data_in;
     else
     begin
      serial_data_out = sft_reg[0];
      if (serdes_factor == count_ps)
      count_ps =1'b0 ;
      else
      begin
      sft_reg = {sft_reg,sft_reg[7:1]};
      count_ps = count_ps +1;
      end
     end
end
end
endmodule
```

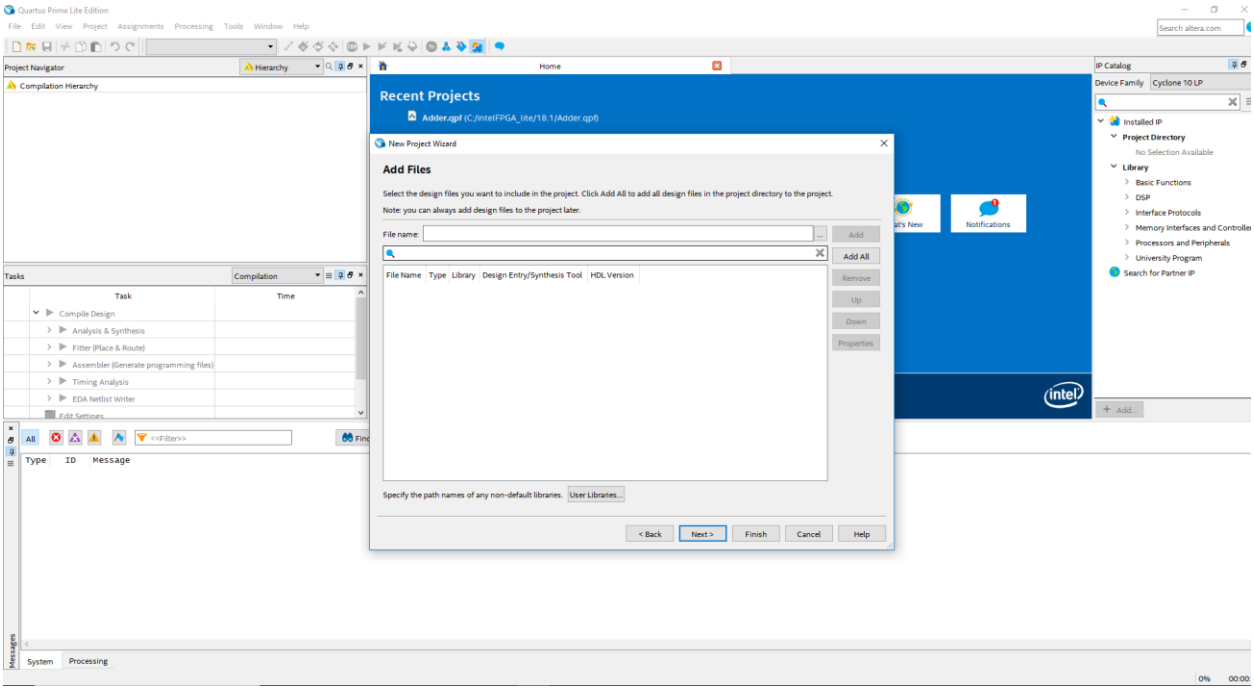
Quick Quartus Flow



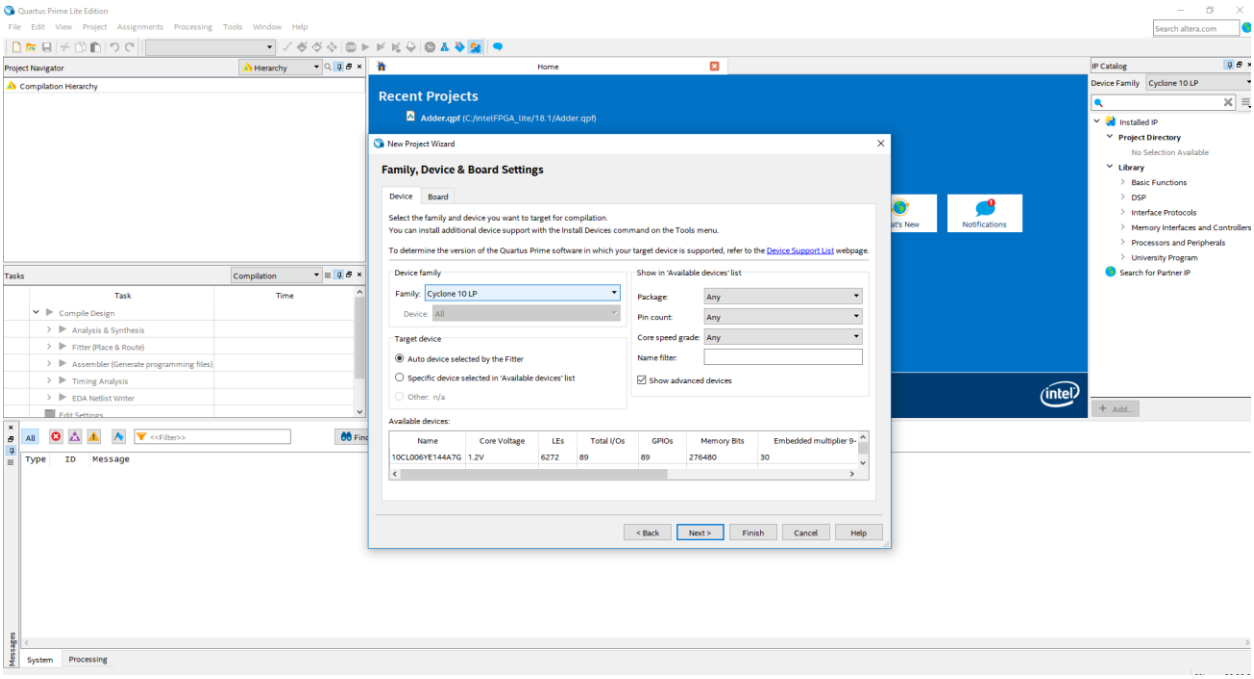
Create or open project



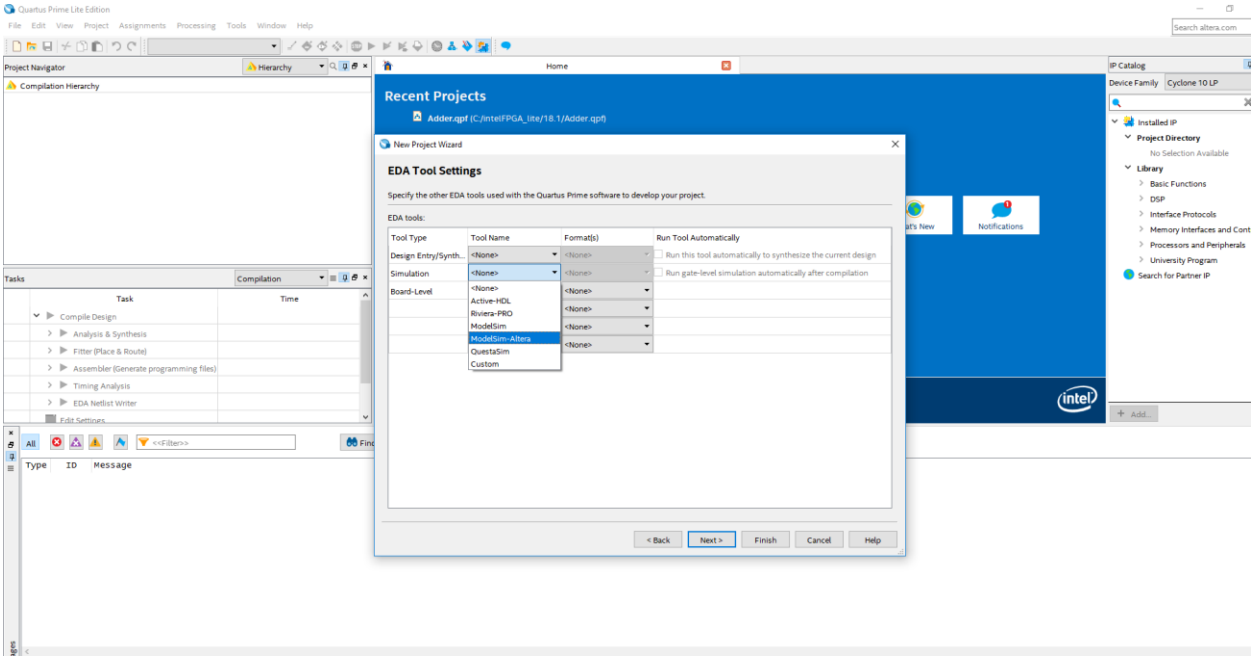
Add file to the project if exists (Verilog/VHDL/etc.)



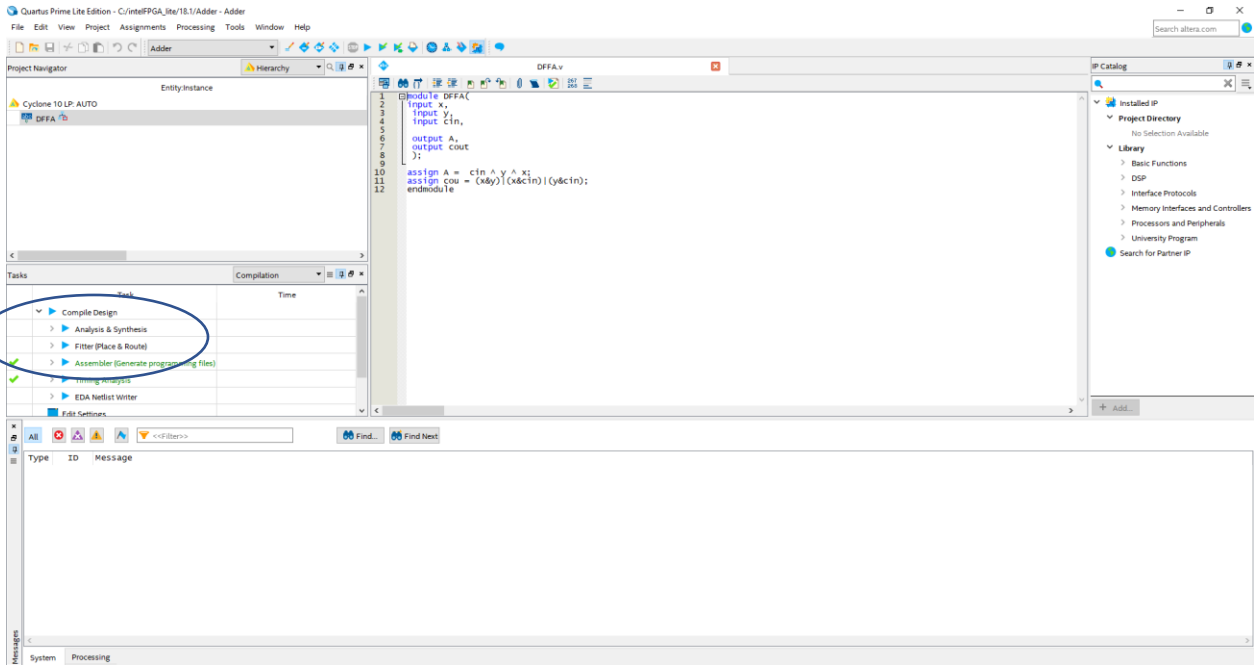
Select Device (FPGA family)



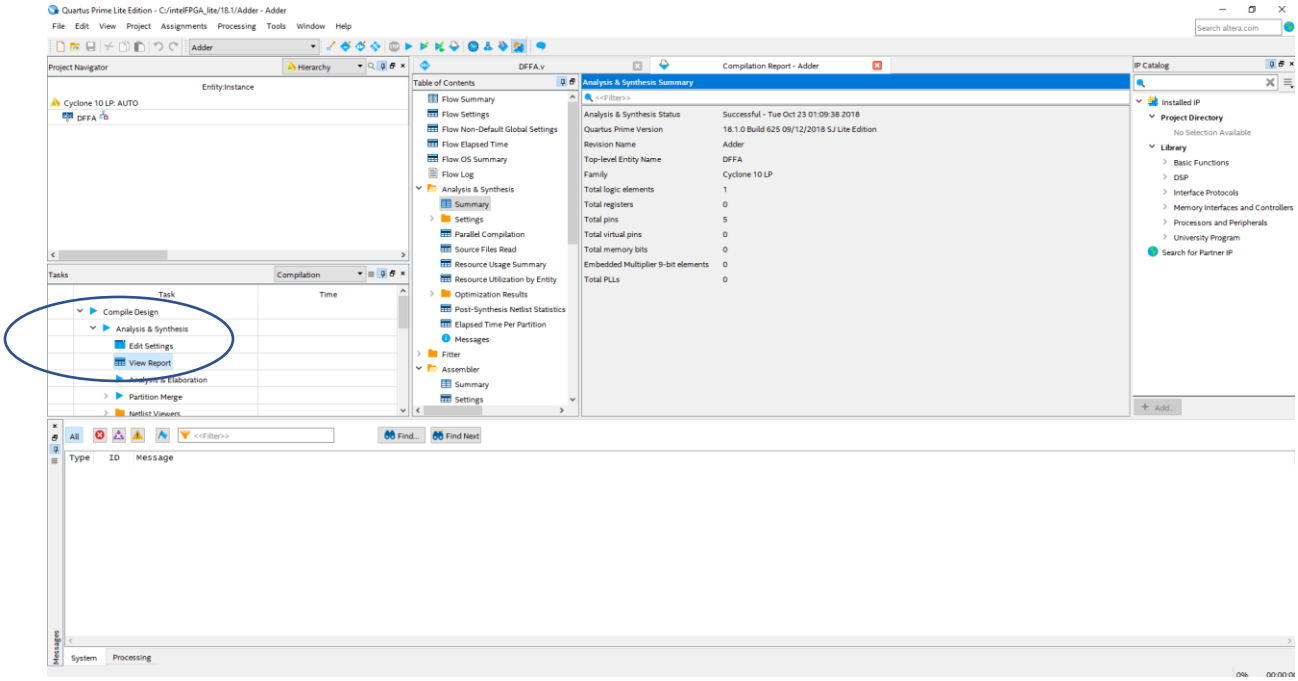
Select Simulation tool if you plan to simulate within Altera platform



Write/Import your code and compile design



View reports of utilization



RTL viewer is a good way to visualize your system blocks