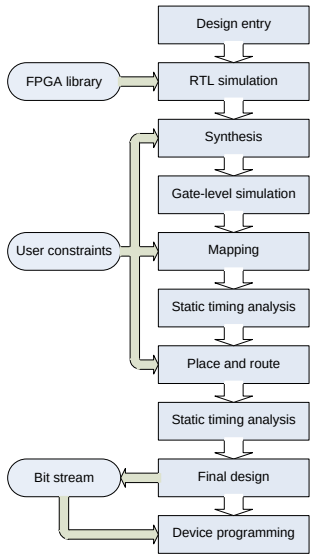
Verilog Tutorial

May 2014

Ando KI

Contents

Standard FPGA-based design flow



Contents

Hello world

- Try a simple example to display text on the screen.

- GUI based
  - Step 1: coding using text editor, e.g., vi or vim.
  - Step 2: compilation
  - Step 3: simulation

- Command based
  - Step 1: coding using text editor, e.g., vi or vim.
  - Step 2: compilation
  - Step 3: simulation

Hello world: coding

- Make a directory say 'ex_hello'
- Go to the directory
- Create a Verilog file 'hello.v'


```
module top;

    initial $display("Hello world!");

endmodule
```

'module' and 'endmodule' specifies a block.

Top-level module contains whole design, which does not have port.

Initial construct is executed at the start of simulation.

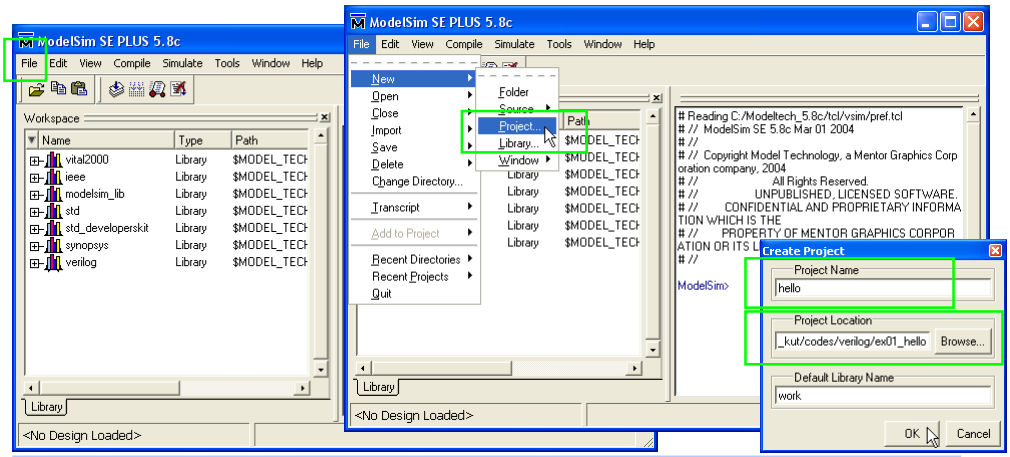
initial_construct ::= **initial** statement

Display task puts information on the terminal.

display_task ::= **$display** (list_of_arguments);

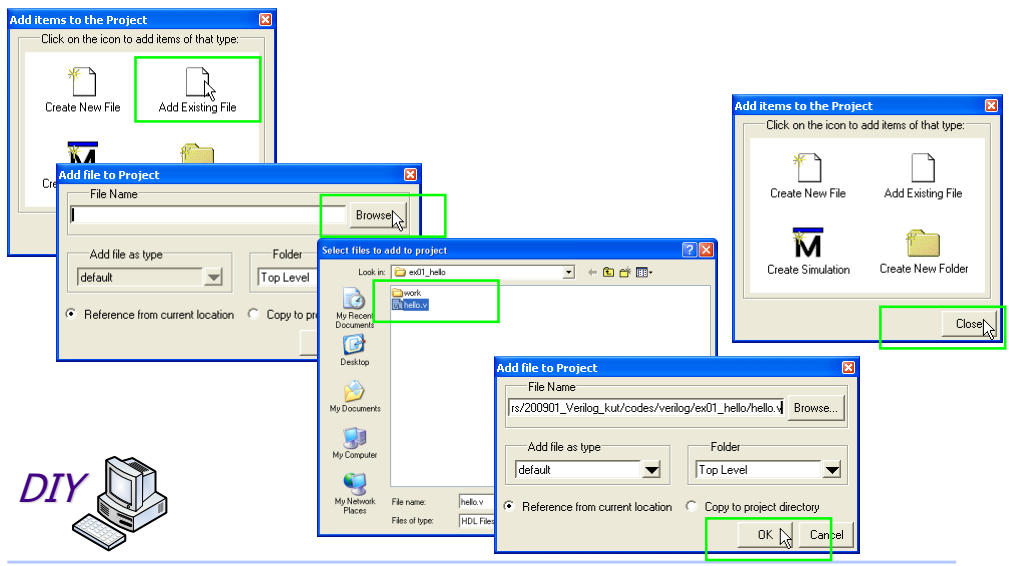
Create a new project

- Invoke ModelSim
- File → New → Project
- Specify ‘Project Name’ and ‘Project Location’

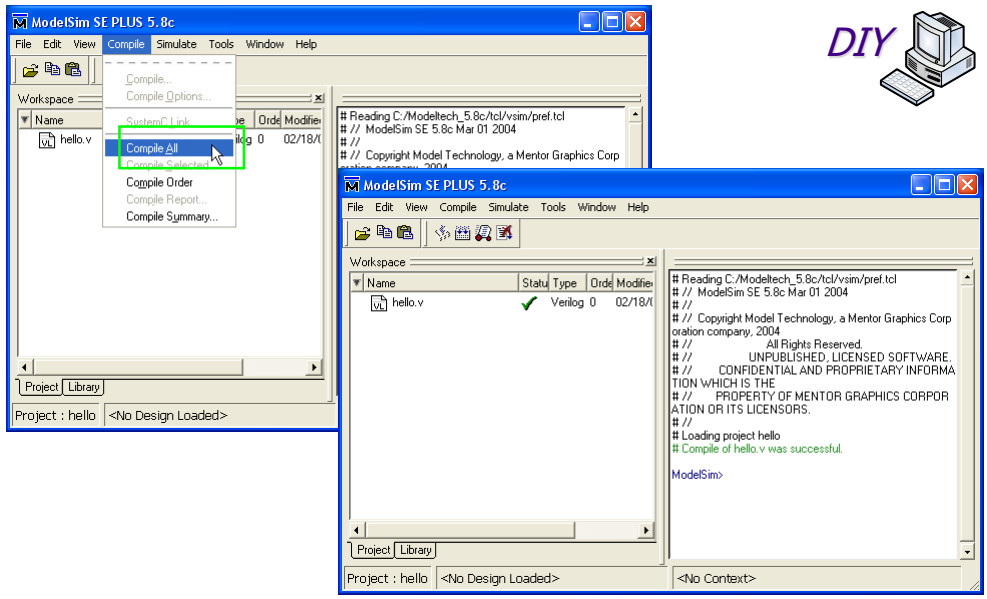


Add existing file

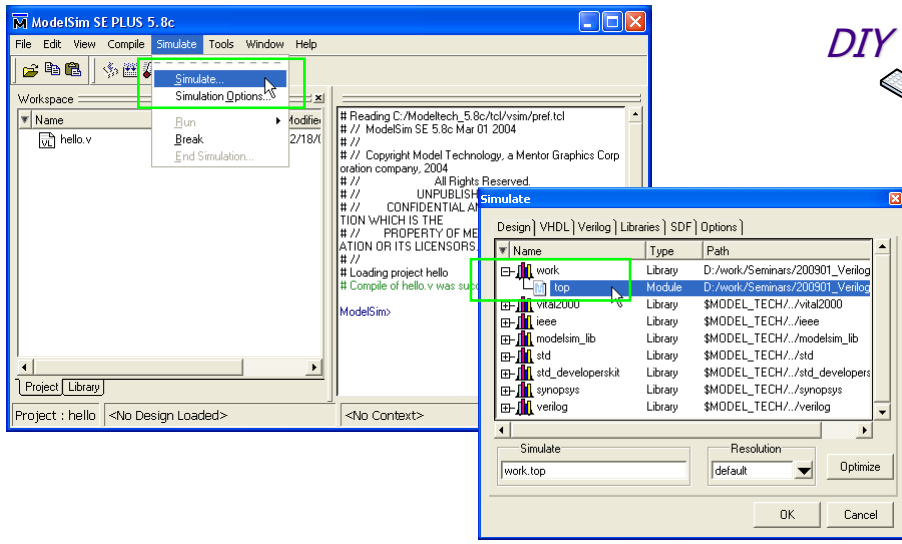
- Add the Verilog design file



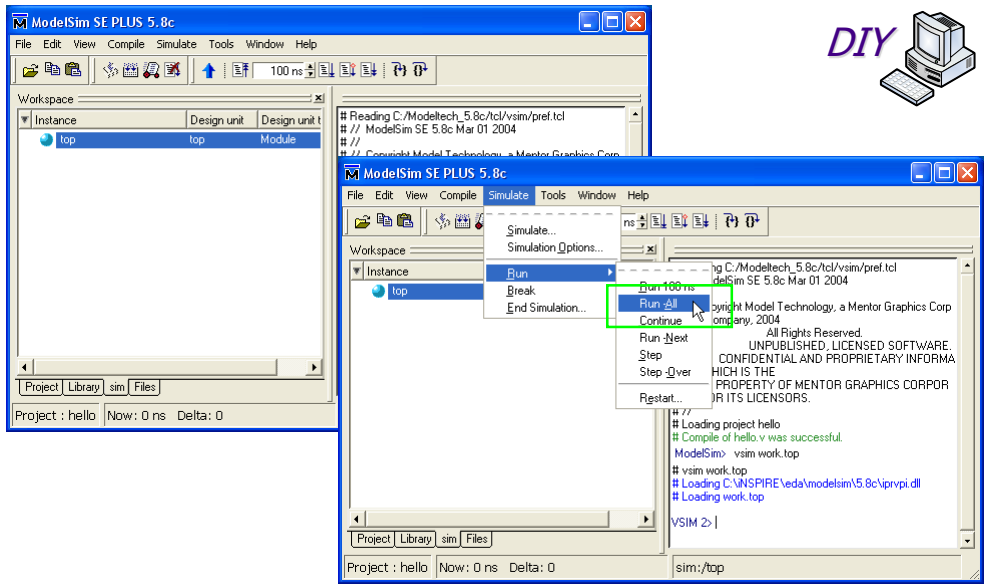
Compile



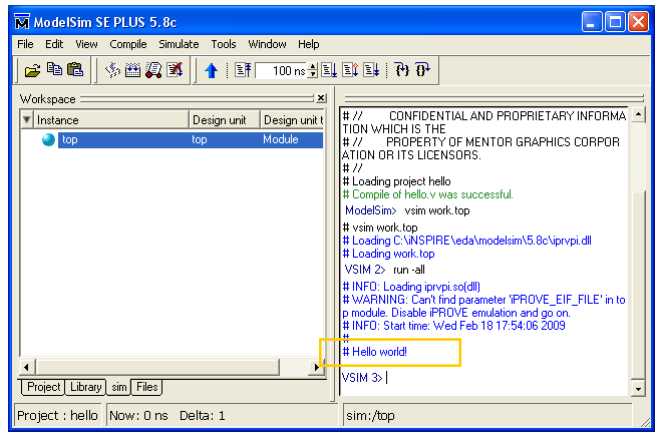
Compile



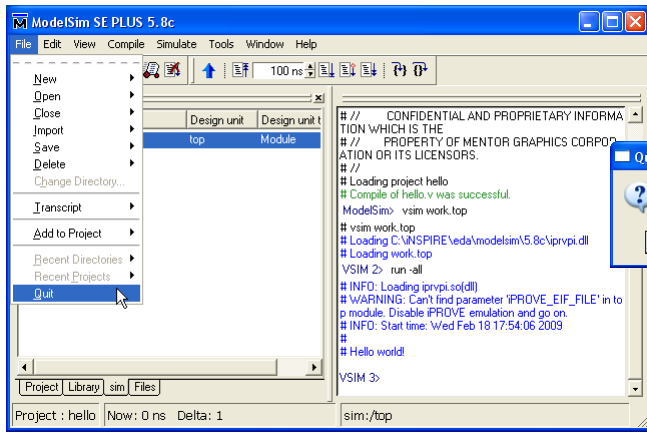
Compile



Simulation



Quit



- There should be 'hello.mpf', which is ModelSim project file.

Contents

Command based simulation

```
vlib   work
vlog  hello.v
vsim -c -do "run -all; quit"  work.top
```

- 'vlib' specifies where compiled library locates.
- 'vlog' compiles Verilog code.
- 'vsim' simulate the design with a given command.
- You can use any script language such as MS-DOS, shell and so on.

The **vlib** command creates a design library.

```
vlib  [options] <name>
```

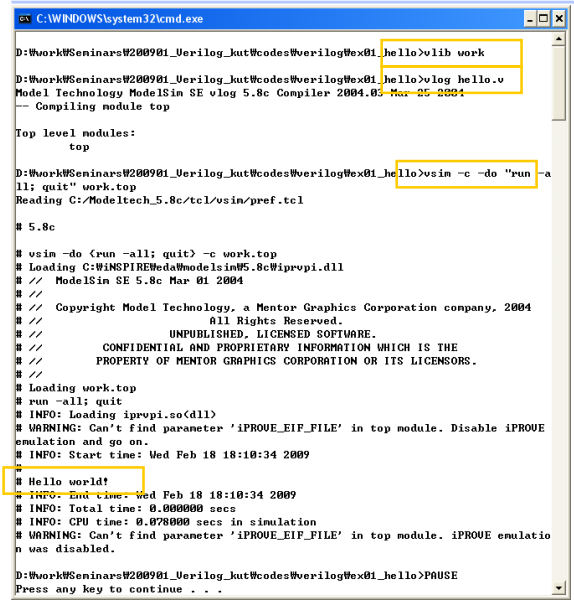
The **vlog** command compiles Verilog source code into a specified working library (or to the **work** library by default).

```
vlog  [options] <filename>
```

The **vsim** command is used to invoke the VSIM simulator.

```
vsim [-c] [-do "<command_string>"]  <library_name>.<design_unit>
```


Command based simulation result

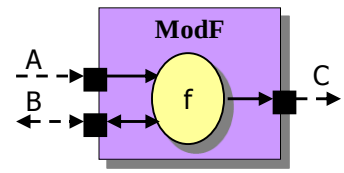


Contents

- Design flow overview
- Hello world
  - GUI based
  - Command based
- Module
  - Declaration
  - Instantiation
  - Port
  - Test-bench
- Adder example
  - Example design
  - Simulation
  - Synthesis
  - Gate-level simulation

Module declaration

- Module is the basic design unit.
- Module must be declared. → *prepare*
- Module can be instantiated. → *use*
- A module definition shall be enclosed between the keywords 'module' and 'endmodule'.



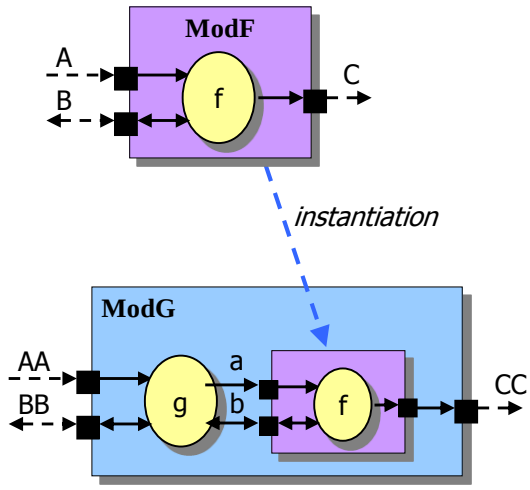

```
module ModF ( input wire A
            , input wire [7:0] B
            , output wire [7:0] C);
  // declarations
  // description of 'f'
endmodule
```

- Module syntax

```
module module_name (list of ports);
  // in, out, inout port declarations
  // signal/wire/reg declarations
  // data variable declarations
  // sub-module instantiation and connection
  // functional blocks: initial, always, function, task
endmodule
```

```
module ModF (A, B, C);
  input       A;
  inout  [7:0] B;
  output [7:0] C;
  // declarations
  // description of 'f'
endmodule
```

Module instantiation



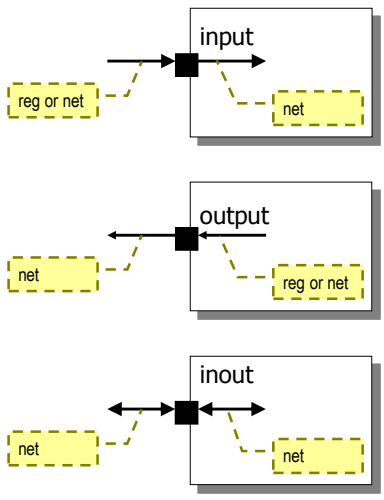

```
module ModG (AA, BB, CC);
   input        AA;
   inout  [7:0] BB;
   output [7:0] CC;
   wire         a;
   wire [7:0] b;
   // description of 'g'
   ModF Umodgf(.A(a), .B(b), .C(CC));
endmodule
```

Port connection

Instance name

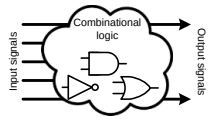
Module name

Connection through port



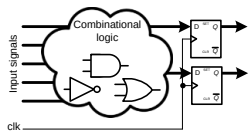
- Port connection rules
  - Input port must be net (i.e. wire).
  - Output port can be either net or reg.
  - Inout port must be net.

Coding



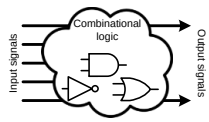

```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d = sig_in_a & sig_in_b;
assign sig_out_c = sig_in_a | sig_in_b;
```

Combinational logic inference by continuous assignment (note wire and assign)




```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ (posedge clk) begin
   sig_out_d <= sig_in_a & sig_in_b;
   sig_out_c <= sig_in_a | sig_in_b;
end
```

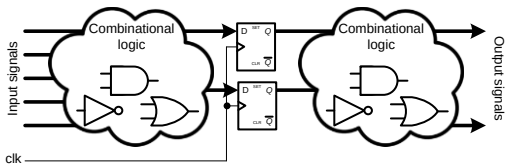
Register or Flip-Flop inference by always block (note “posedge clk”)




```
wire sig_in_a;
wire sig_in_b;
reg sig_out_c;
reg sig_out_d;

always @ ( * ) begin
   sig_out_d <= sig_in_a & sig_in_b;
   sig_out_c <= sig_in_a | sig_in_b;
end
```

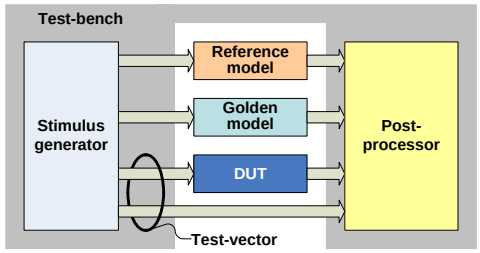
Combinational logic inference by always block (note event list or sensitivity list)

Coding



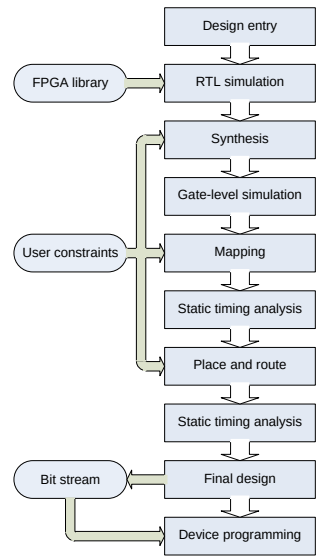

```
wire sig_in_a;
wire sig_in_b;
wire sig_out_c;
wire sig_out_d;
reg sig_reg_x;
reg sig_reg_y

always @ (posedge clk) begin
   sig_reg_x <= sig_in_a & sig_in_b;
   sig_reg_y <= sig_in_a | sig_in_b;
end

assign sig_out_c = sig_reg_x & sig_reg_y;
assign sig_out_d = sig_reg_x | sig_reg_y;
```

Test-bench



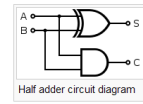
- A test-bench is a layer of code that is created to apply input patterns (stimulus) to the DUT (design under test) and to determine whether the DUT produces the outputs expected.
- A test-vector is a set of values for all the expected input ports (stimuli) and expected values for the output ports of a module under test.
- A test-bench that is created to apply inputs, sample the outputs of the DUT, and compare the outputs with the expected (golden) results is called a self-checking test-bench.

Contents

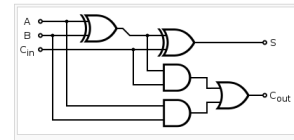


What is adder

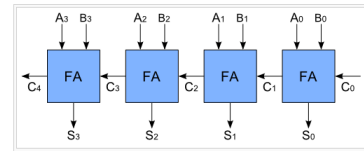
- An adder is a digital circuit that performs addition of numbers.
  - The half adder adds two one-bit binary numbers (A and B). The output is the sum of the two bits (S) and the carry (C).
  - The full-adder circuit adds three one-bit binary numbers (C, A and B) and outputs two one-bit binary numbers, a sum (S) and a carry (C).
- Multi-bit adder
  - Ripple carry adder
  - Carry look-ahead adders



Half adder circuit diagram



Full adder circuit diagram
Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

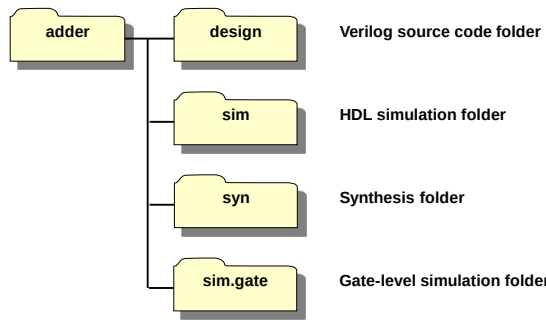


4-bit ripple carry adder circuit diagram

Picture has been adopted from Wikipedia (http://en.wikipedia.org/wiki/Half_adder).

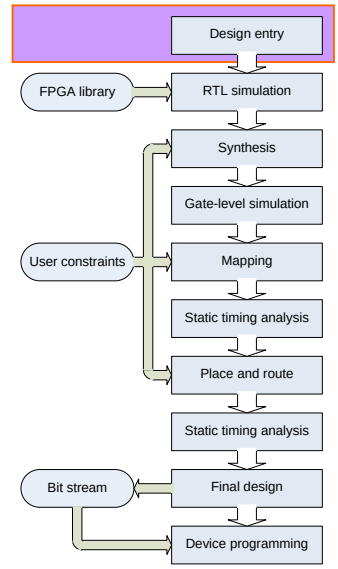
Directory structure

- Directory structure: $(PROJECT)/codes/ex_verilog/tutorial



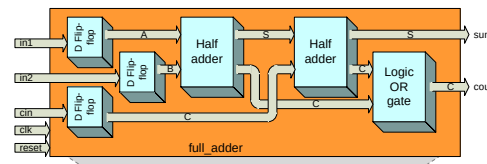
Contents

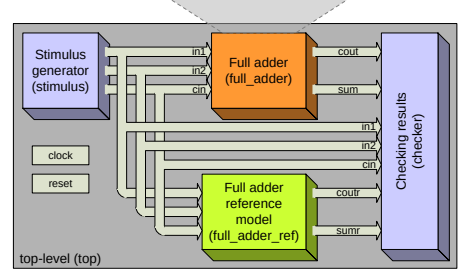
- Design flow overview
- Hello world
  - GUI based
  - Command based
- Module
  - Declaration
  - Instantiation
  - Port
  - Test-bench
- Adder example
  - What is adder
  - Directory structure
  - Example design
  - Simulation
  - Synthesis
  - Gate-level simulation

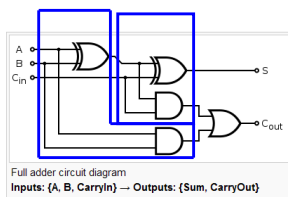


Example design

- See the 'design' directory
  - top.v
  - full_adder.v
  - half_adder_gate.v
  - half_adder_rtl.v
  - stimulus.v
  - full_adder_ref.v
  - checker.v



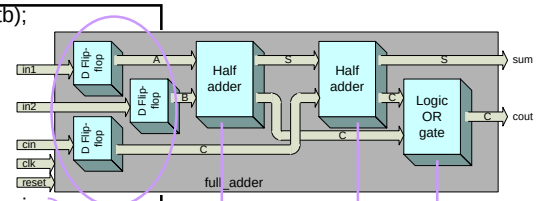




Full adder circuit diagram
Inputs: {A, B, CarryIn} → Outputs: {Sum, CarryOut}

Full adder (1/3)

```
module full_adder(sum,cout,in1,in2,cin,clk,resetb);
   output sum, cout;
   input  in1, in2, cin;
   input  clk, resetb;

   wire   sum, cout;
   reg    rin1, rin2, rcin;
   wire   s1, c1, s2, c2;

   always @ (posedge clk or negedge resetb) begin
      if (resetb==1'b0) begin
        rin1 <= 1'b0;  rin2 <= 1'b0;   rcin <= 1'b0;
      end else begin
        rin1 <= in1;    rin2 <= in2;     rcin <= cin;
      end
   end

   half_adder_gate ha1 (.S(s1), .C(c1), .A(rin1), .B(rin2));
   half_adder_rtl     ha2 (.S(s2), .C(c2), .A(s1),   .B(rcin));

   assign sum = s2;
   assign cout = c1|c2;
endmodule
```



Full adder (2/3)

```
module full_adder(sum,cout,in1,in2,cin,clk,resetb);
   output sum, cout;
   input  in1, in2, cin;
   input  clk, resetb;

   wire   sum, cout;
   reg    rin1, rin2, rcin;
   wire   s1, c1; wire   s2, c2;

   always @ (posedge clk or negedge resetb) begin
      if (resetb==1'b0) begin
        rin1 <= 1'b0;  rin2 <= 1'b0;  rcin <= 1'b0;
      end else begin
        rin1 <= in1;  rin2 <= in2; rcin <= cin;
      end
   end

   half_adder_gate ha1 (.S(s1), .C(c1), .A(rin1), .B(rin2));
   half_adder_rtl     ha2 (.S(s2), .C(c2), .A(s1),   .B(rcin));

   assign sum = s2;
   assign cout = c1|c2;
endmodule
```

- module name
- ports
- port directions
- internal signals
- internal design description
- sub module instantiation
- internal design description

Full adder (3/3)

```
module full_adder(sum,cout,in1,in2,cin,clk,resetb);
   output sum, cout;
   input  in1, in2, cin;
   input  clk, resetb;

   wire   sum, cout;
   reg    rin1, rin2, rcin;
   wire   s1, c1; wire   s2, c2;

   always @ (posedge clk or negedge resetb) begin
      if (resetb==1'b0) begin
         rin1 <= 1'b0;  rin2 <= 1'b0;  rcin <= 1'b0;
      end else begin
         rin1 <= in1;  rin2 <= in2; rcin <= cin;
      end
   end

   half_adder_gate ha1 (.S(s1), .C(c1), .A(rin1), .B(rin2));
   half_adder_rtl     ha2 (.S(s2), .C(c2), .A(s1),   .B(rcin));

   assign sum = s2;
   assign cout = c1|c2;
endmodule
```

Verilog data types: net and variable
- net: wire
- variable: reg, integer, real ...

This always block runs when the following condition occurs.
positive edge of 'clk'
negative edge of 'resetb'

Procedural assignment places values to variables within always, initial and so on.

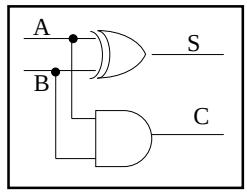
==: equality check operator
<=: non-blocking assignment operator

Sub-module instantiation with named port connection; refer to positional pot connection.

Continuous assignment places values to nets whenever the value of the right-hand side changes.

|: bit-wize OR operator

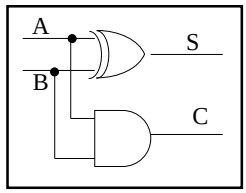
Half adder structural model (gate level)




```
module half_adder_gate (S, C, A, B);
        output S, C;
        input   A, B;
        and UAND (C, A, B);
        xor UXOR (S, A, B);
endmodule
```

Structural model: instantiation of primitives and modules.

Half adder data-flow model (RTL)




```
module half_adder_rtl (S, C, A, B);
        output S, C;
        input  A, B;
        wire   S, C;
        assign C = A & B;
        assign S = A ^ B;
endmodule
```

Data-flow model: continuous assignments.

Test-bench: stimulus

```
module stimulus(out1,out2,out3,clk,resetb);
   output out1,out2,out3;
   input clk,resetb;

   reg out1,out2,out3;

   initial begin
      out1 <=0; out2 <=0; out3 <=0;

      wait (resetb==1'b0);
      wait (resetb==1'b1);
      @ (posedge clk);

      out1=1; out2=0; out3=0; @ (posedge clk);
      out1=0; out2=1; out3=0; @ (posedge clk);
      out1=1; out2=1; out3=0; @ (posedge clk);
      out1=0; out2=0; out3=1; @ (posedge clk);
      out1=1; out2=0; out3=1; @ (posedge clk);
      out1=0; out2=1; out3=1; @ (posedge clk);
      out1=1; out2=1; out3=1; @ (posedge clk);

      repeat (3) @ (posedge clk);

      $finish;
   end
endmodule
```

- Initial construct is executed at the start of simulation.
- <=: non-blocking assignment operator
- 'wait': It blocks until the condition becomes true.
- '@' waits until the specified event occurs.
- =: blocking assignment operator
- 'repeat': Executes a statement a fixed number of times.
- The finish system task simply makes the simulator exit and pass control back to the host computer operating system.

Test-bench: checker

```
module checker(in1,in2,cin,sum,cout,sumr,coutr,clk,resetb);
   input in1,in2,cin,sum,cout,sumr,coutr,clk,resetb;
   always @ (clk) begin
      if ({cout,sum}=={coutr,sumr})
            $display($time,,"correct");
      else $display($time,,"error result=%b expect=%b",  {cout, sum}, {coutr,sumr});
   end
endmodule
```

Display task puts information on the terminal.

display_task ::= $display (list_of_arguments);

Time system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.

time_function ::= $time ;

Test-bench: full_adder_ref

```
module full_adder_ref(sum,cout,in1,in2,cin,clk,resetb);
   output sum, cout;
   input  in1, in2, cin;
   input  clk, resetb;

   wire   sum, cout;
   reg    rin1, rin2, rcin;
   wire   s1, c1;
   wire   s2, c2;

   always @ (posedge clk or negedge resetb) begin
       if (resetb==1'b0) begin
          rin1 <= 1'b0;
          rin2 <= 1'b0;
          rcin <= 1'b0;
       end else begin
          rin1 <= in1;
          rin2 <= in2;
          rcin <= cin;
       end
   end

   assign {cout, sum} = rin1+rin2+rcin;

endmodule
```

'assign': Continuous assignment places values to nets whenever the value of the right-hand side changes.

Concatenation operator ({, }) combines two or more in order to form wider bits.

Test-bench: top

```
module top;
  wire sum, cout, in1, in2, cin;
  reg  clk;
  full_adder fa (.sum(sum), .cout(cout), .in1(in1), .in2(in2), .cin(cin));
  stimulus   st (.out1(in1), .out2(in2), .out3(cin), .clk(clk));
  checker    ck (.in1(in1), .in2(in2), .cin(cin), .sum(sum), .cout(cout), .clk(clk));
 initial begin
       clk = 0;
       forever #5 clk = ~clk;
 end
 initial begin
       $dumpfile("wave.vcd");
       $dumpvars(1)
 end
endmodule
```

Top-level module contains whole design, which does not have port.

Forever statement continuously executes a statemet.

forever_statement ::= **forever** statement

Initial construct is executed at the start of simulation.

initial_construct ::= **initial** statement

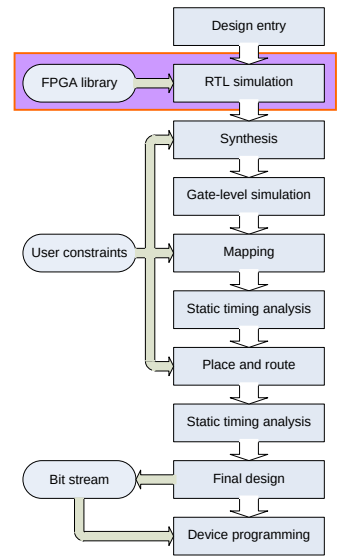
Dumpfile task specify the name of the VCD file.

dumpfile_task ::= $dumpfile(filename) ;

Dumpvars task puts information on the terminal.

dumpvars_task ::= **$dumpvars** (level, [list_of_mod_or_var);

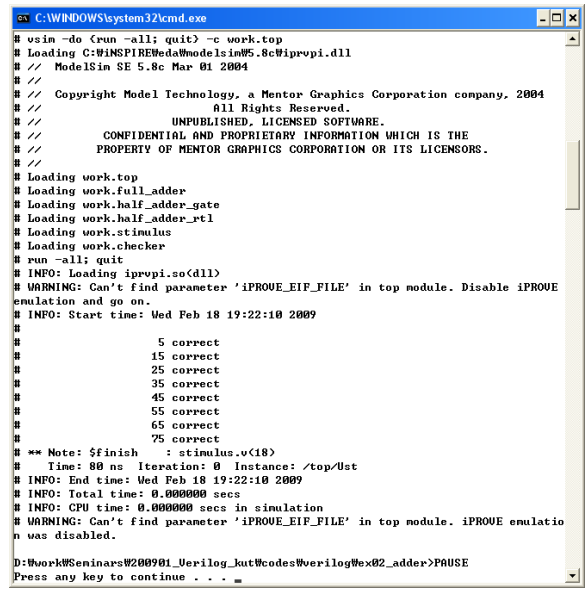
Contents



Command based simulation

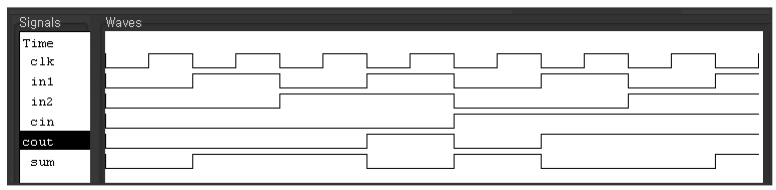
- See ‘sim’ directory

```
vlib work
vlog ../design/top.v
vlog ../design/full_adder.v
vlog ../design/half_adder_gate.v
vlog ../design/half_adder_rtl.v
vlog ../design/stimulus.v
vlog ../design/full_adder_ref.v
vlog ../design/checker.v
vsim -c -do "run -all; quit"  work.top
```



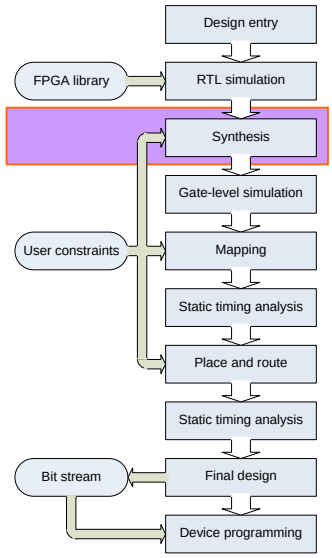
Simulation result with VCD

- Run ‘GTKwave’ to see wave form



Contents

- Design flow overview
- Hello world
  - GUI based
  - Command based
- Module
  - Declaration
  - Instantiation
  - Port
  - Test-bench
- Adder example
  - What is adder
  - Directory structure
  - Example design
  - Simulation
  - Synthesis
  - Gate-level simulation



Synthesis with command-line

- See the 'syn' directory

Windows batch command file "RunMe.bat"

```
set top=full_adder
xst -ifn xst_option.xst -ofn %top%.log
ngc2edif -bd angle -w %top%.ngc %top%.edif
netgen –w –ofmt verilog %top%.ngc %top%_gate.v
```

Generate gate-level design

"xst_option.xst"

```
set -tmpdir .
run
-ifmt MIXED
-top $top
-p $device
-ifn $design
-vlgincdir $inc_dir
-ofn $top
-ofmt NGC
……
-iobuf NO
-fsm_encoding User
……
-iob auto
```

"xst_list.xst"

```
verilog xst "../design/full_adder..v"
verilog xst "../design/half_adder_gate.v"
verilog xst "../design/half_adder_rtl.v"
```

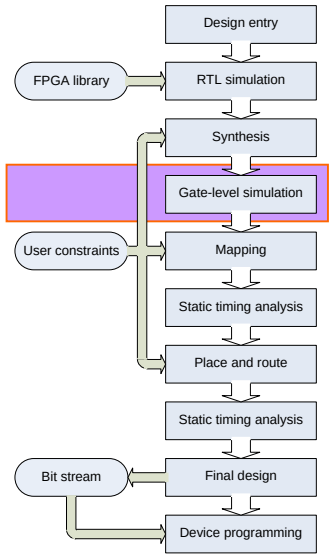
Important notice:
iNCITE does not need IO pads.

FSM encoding can be changed without this option

DIY

Contents

- Design flow overview
- Hello world
  - GUI based
  - Command based
- Module
  - Declaration
  - Instantiation
  - Port
  - Test-bench
- Adder example
  - What is adder
  - Directory structure
  - Example design
  - Simulation
  - Synthesis
  - Gate-level simulation



Command based gate-level simulation

- See ‘sim.gate’ directory

```
vlib work
vlog ../design/top.v
vlog ../design/stimulus.v
vlog ../design/checker.v
vlog ../syn/full_adder_gate.v^
        +libext+.v^
        -y %XILINX%/verilog/src/simprims^
        -y %XILINX%/verilog/src/unisims
vsim -c -do "run -all; quit"  work.top   work.glbl
```

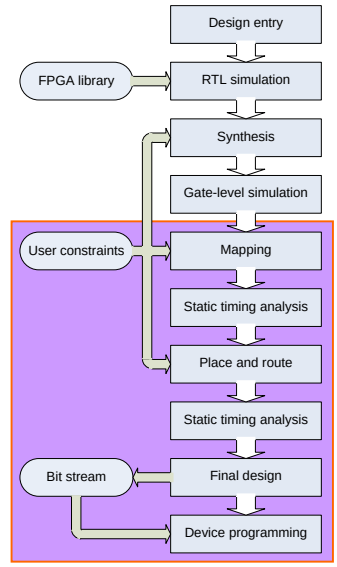
Generated gate-level design while logic synthesis

Gate-level library for Xilinx

Contents

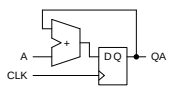
- Design flow overview
- Hello world
  - GUI based
  - Command based
- Module
  - Declaration
  - Instantiation
  - Port
  - Test-bench
- Adder example
  - What is adder
  - Directory structure
  - Example design
  - Simulation
  - Synthesis
  - Gate-level simulation
  - Map & PnR
  - Static timing analysis
  - FPGA-based co-simulation

will cover at the later module

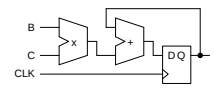


Home work

- Accumulator



- MAC (multiply-accumulate)
  - a ← a + b * c



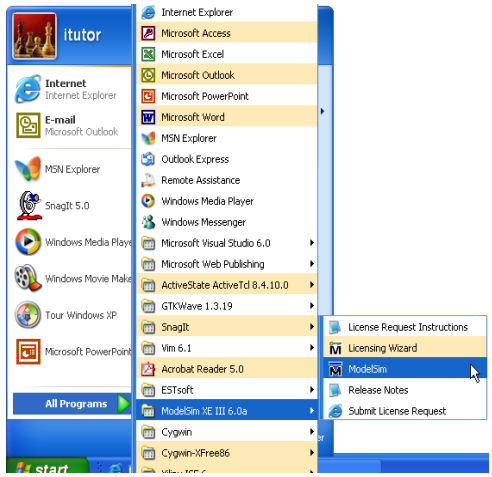
References

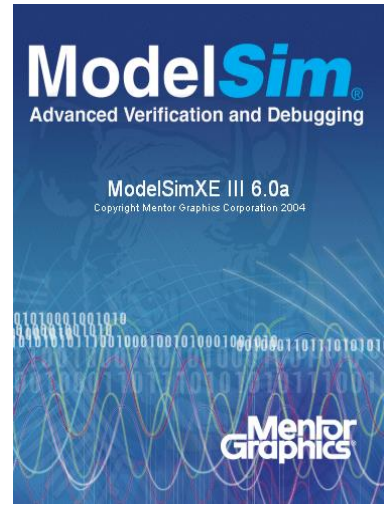
- ModelSim Tutorial, Mentor Graphics.
- ModeSim Reference Manual, Mentor Graphics.
- ModelSim User's Manual, Mentor Graphics.
- ModelSim Command Reference, Mentor Graphics.

Appendix: Simulation with ModelSim GUI

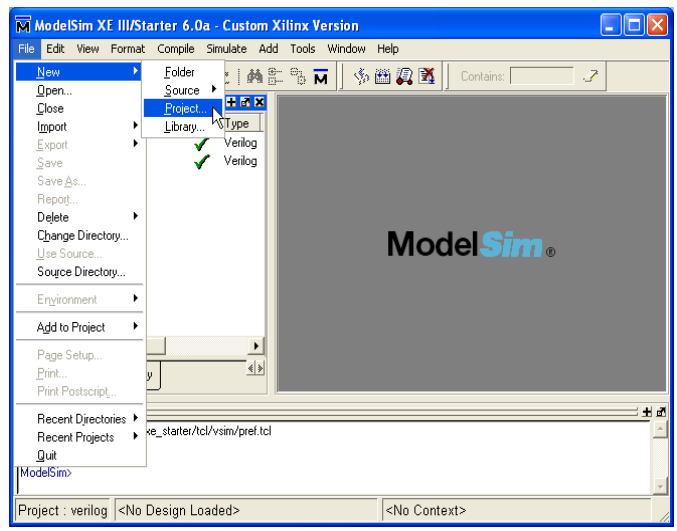
- Invoking ModelSim form start menu
- Create new project
- Add design files
- Compile
- Wave setting
- Simulation
- Invoking ModeSim project

Invoking ModelSim from start menu

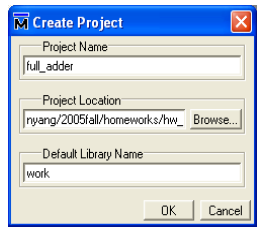


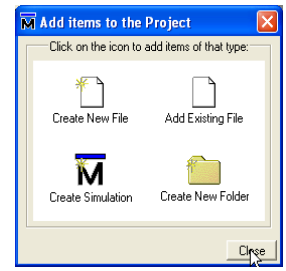


File->New->Project

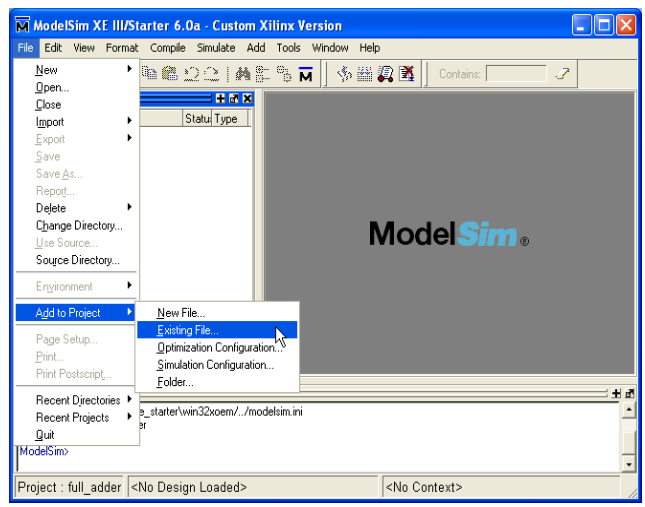


Specify project name and location

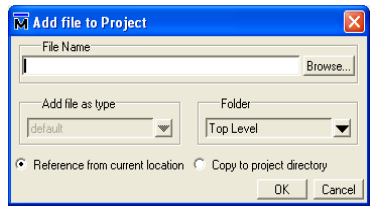


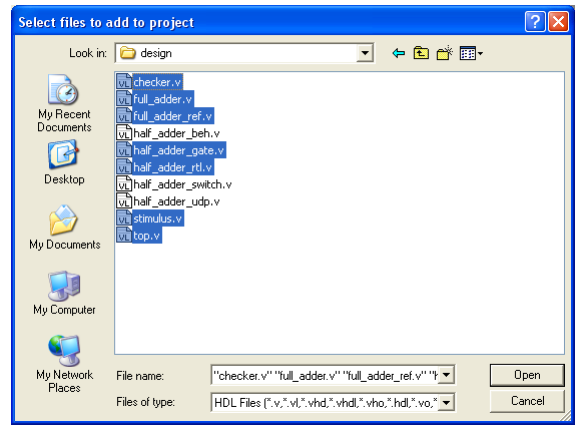


File->Add to Project->Existing File

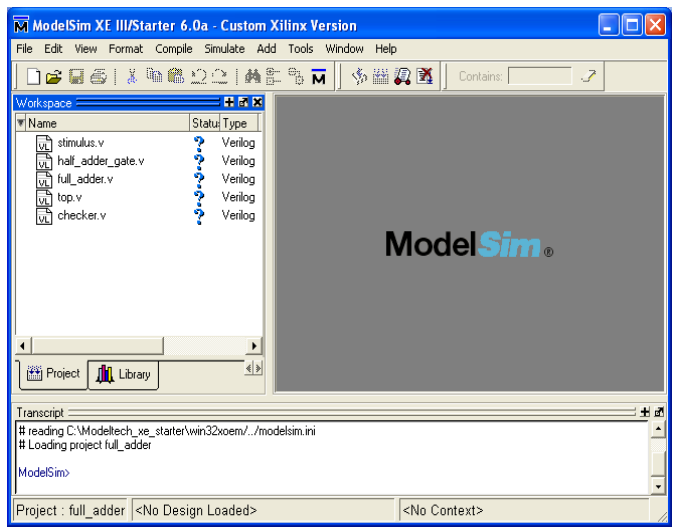


Add files

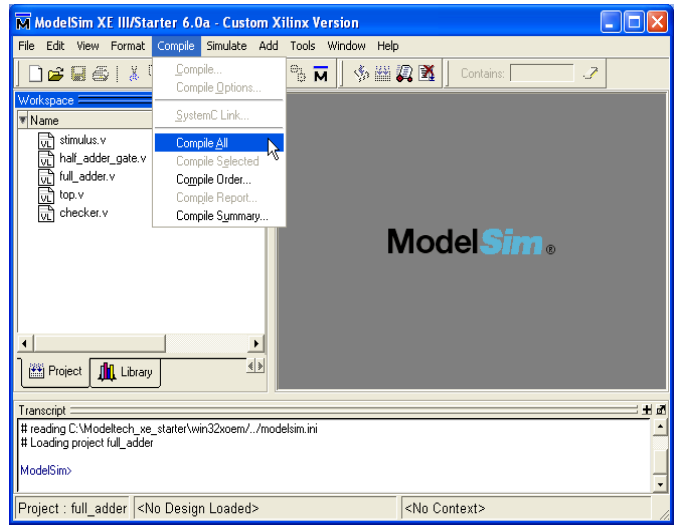




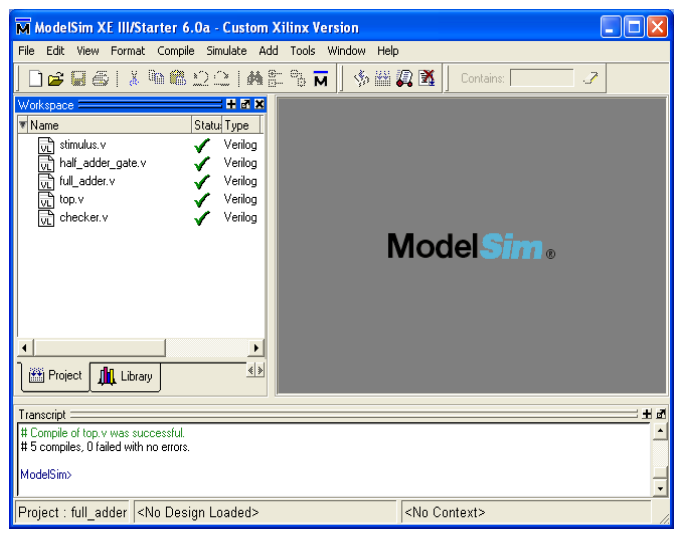
After adding files



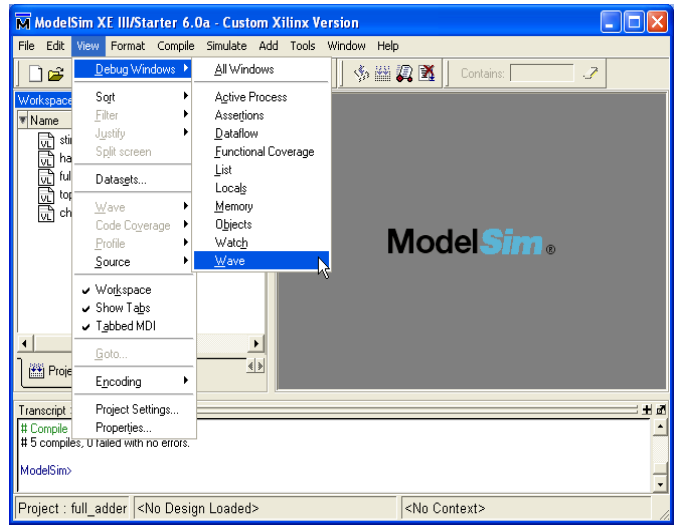
Compile->Compile All



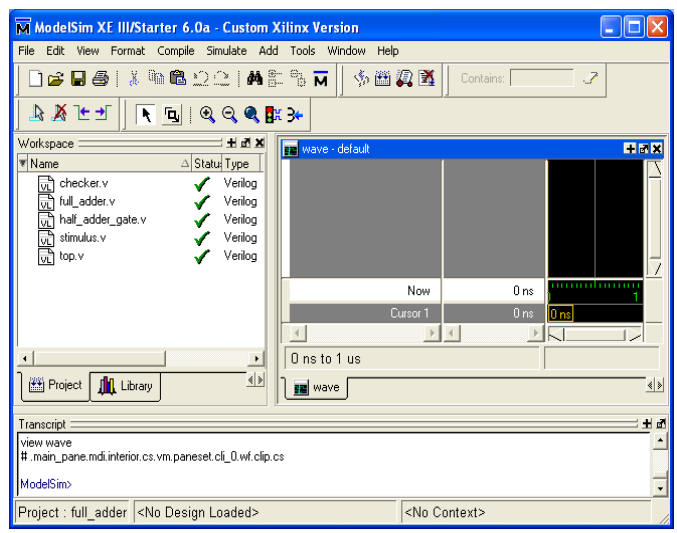
After compilation



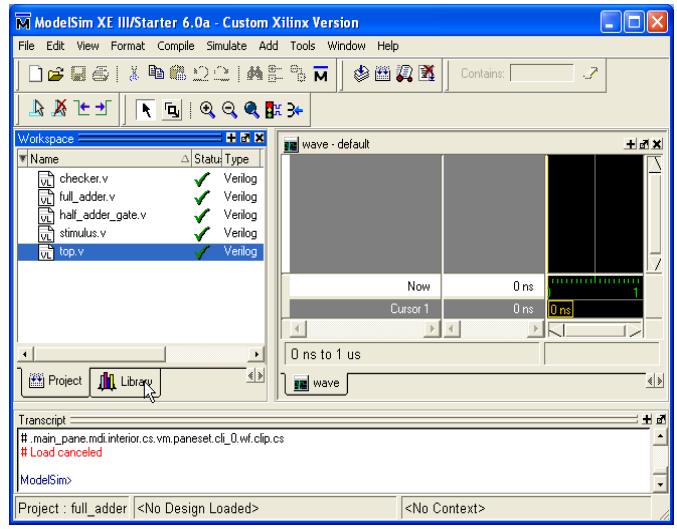
View->Debug Windows->Wave



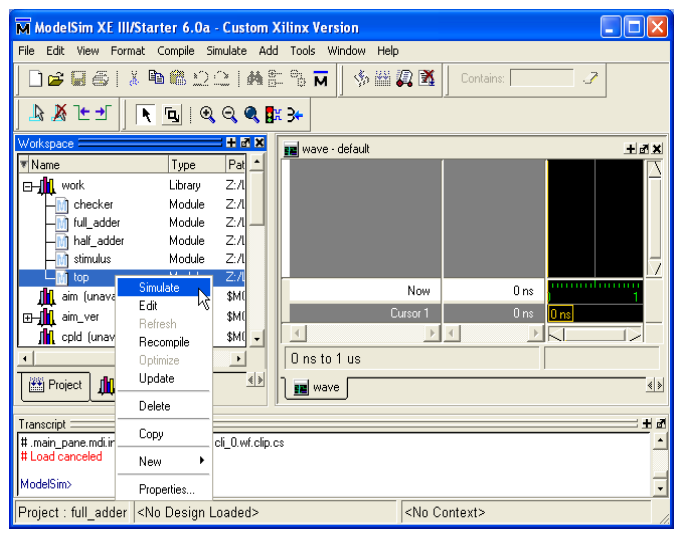
After adding wave window



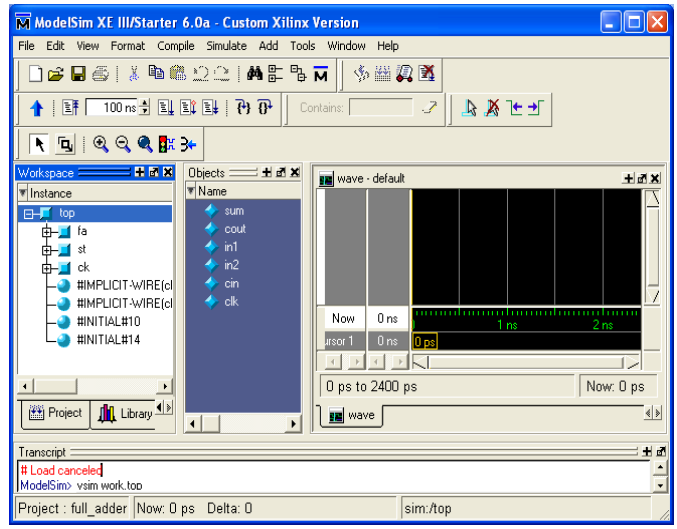
Select Library tab



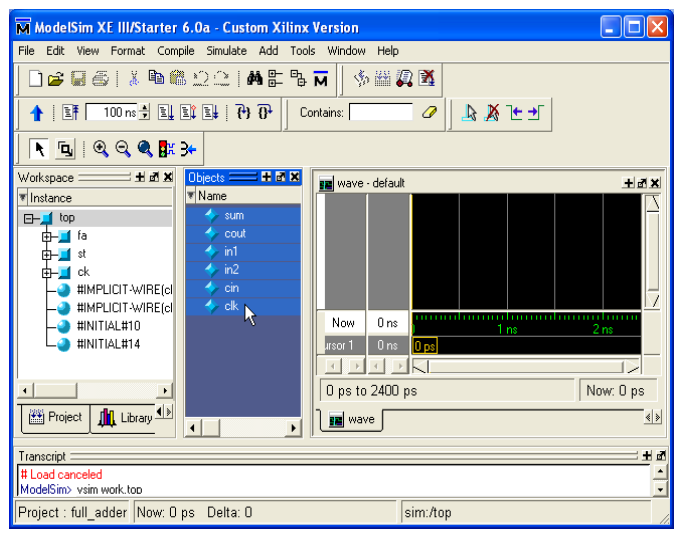
Run simulation with top-level



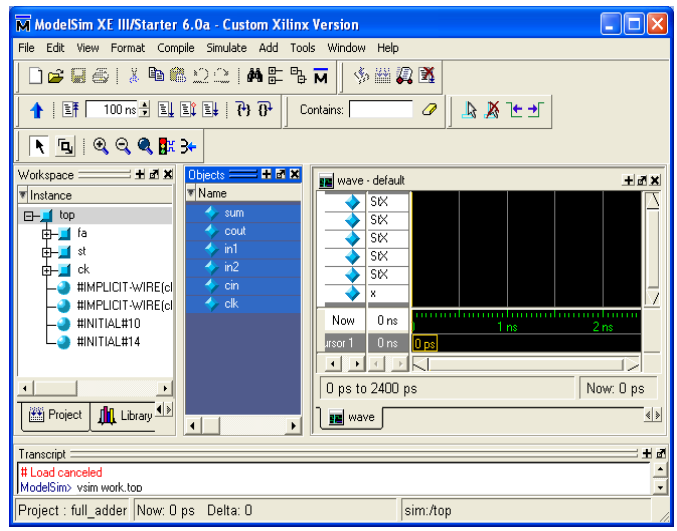
After simulation



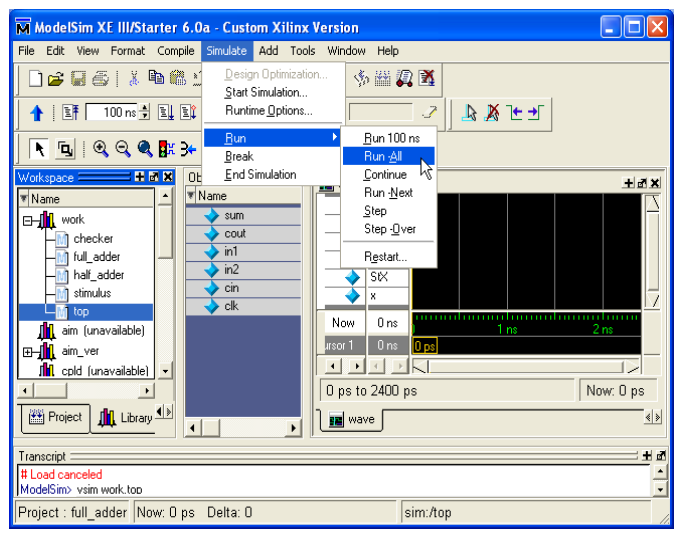
Selecting signals to be view



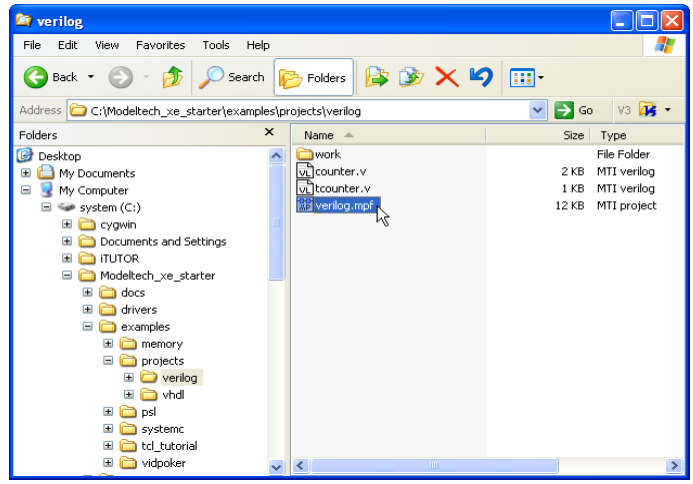
After selection



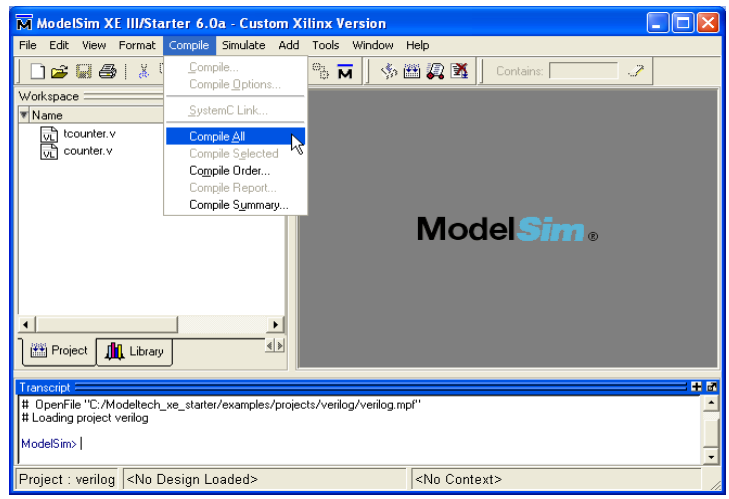
Run-All



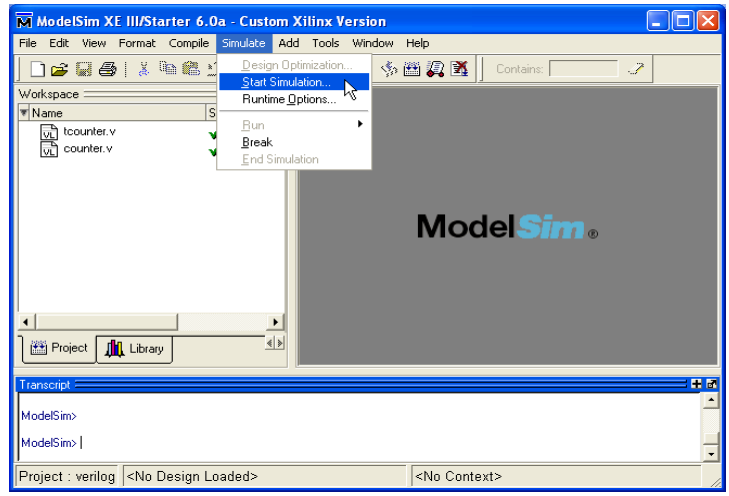
Invoke ModelSim project (1/8)



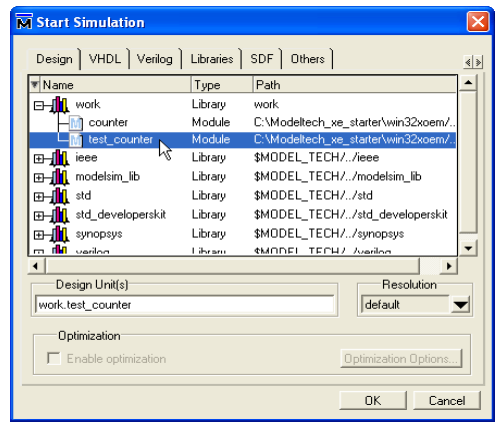
Compile (2/8)



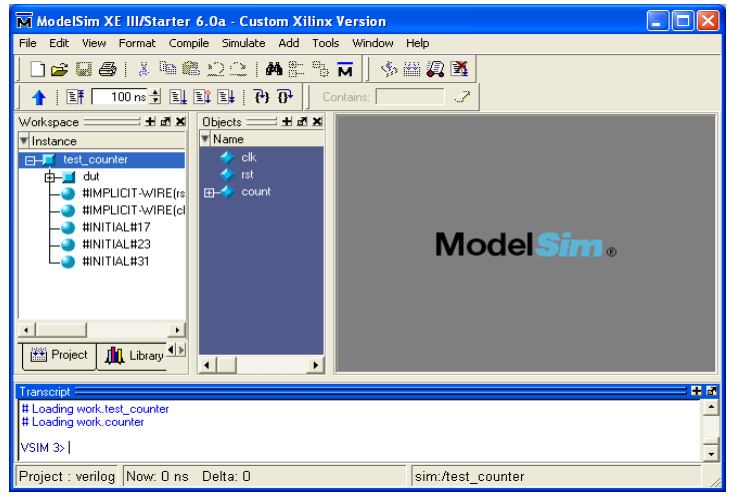
Start simulation (3/8)



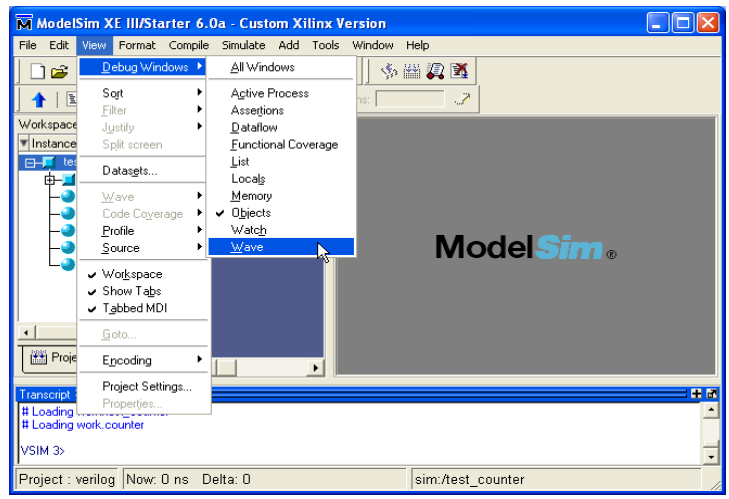
Select top-level (4/8)



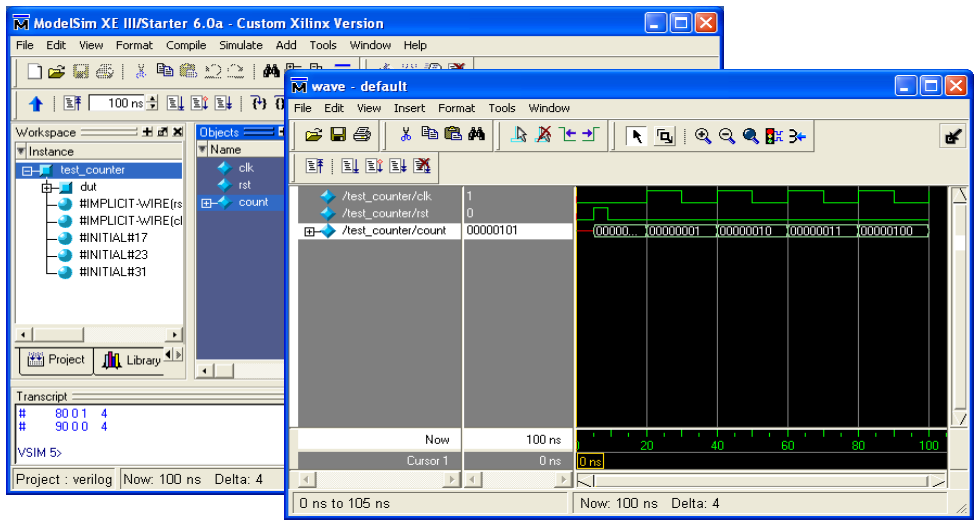
After simulation (5/8)



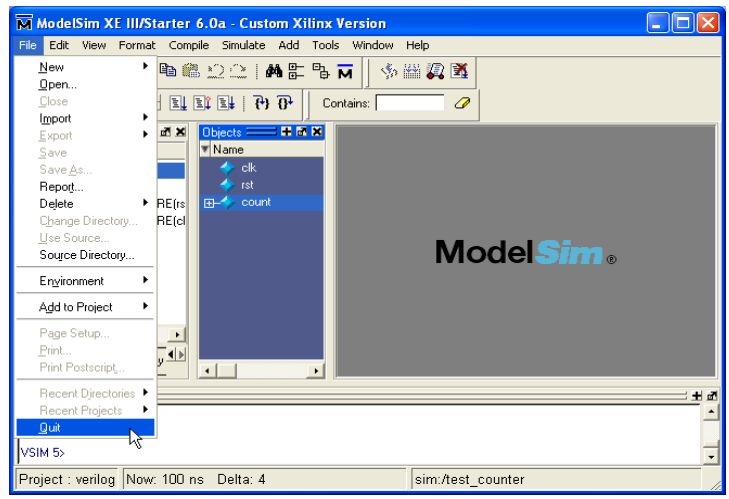
Activate waveform window (6/8)



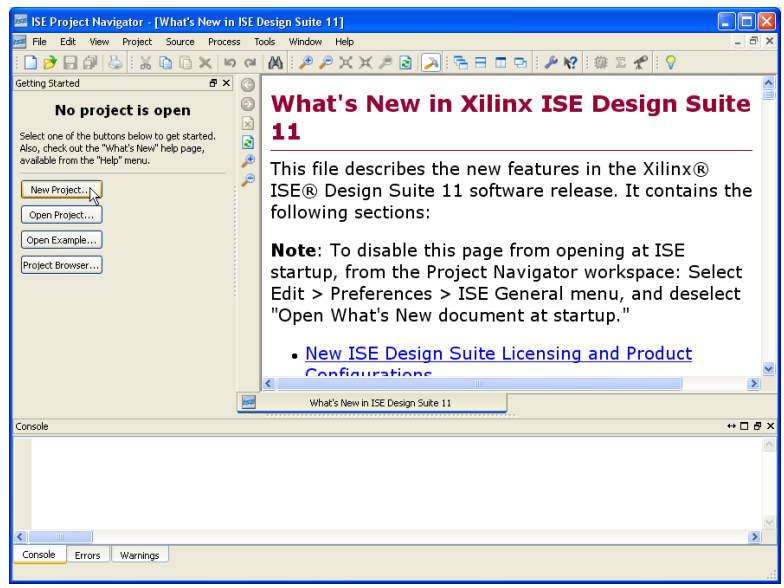
Display waveform (7/8)

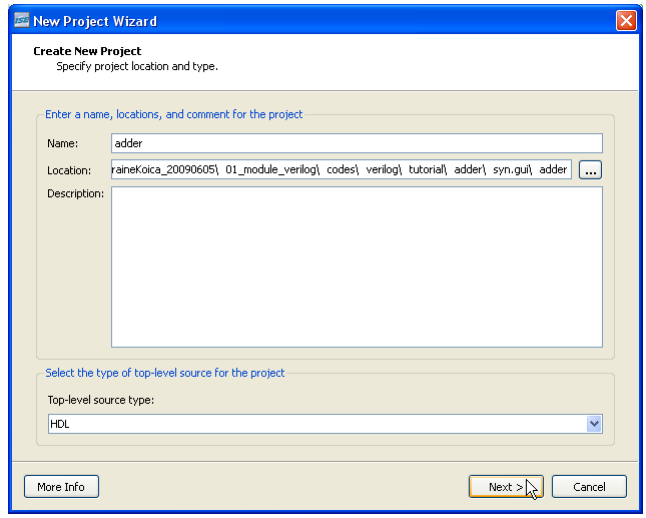


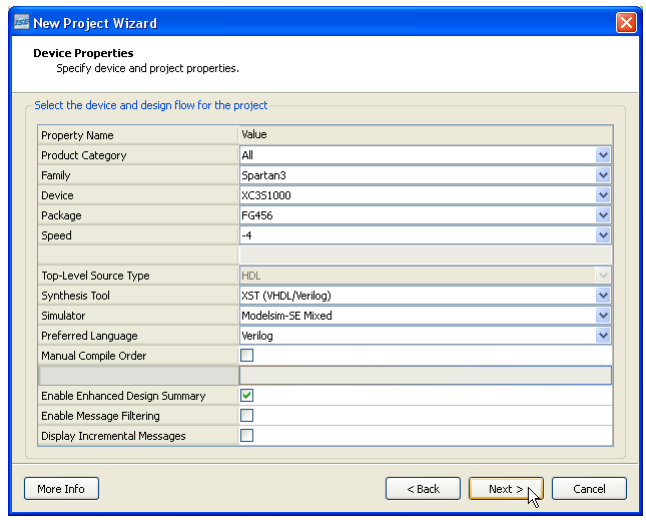
Exiting ModelSim (8/8)

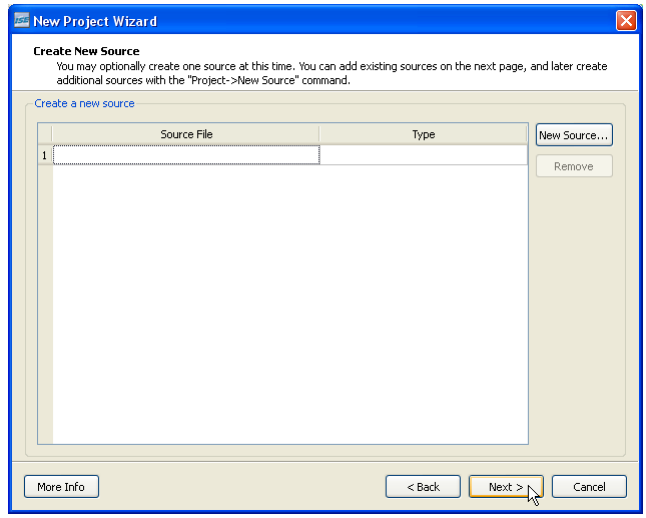


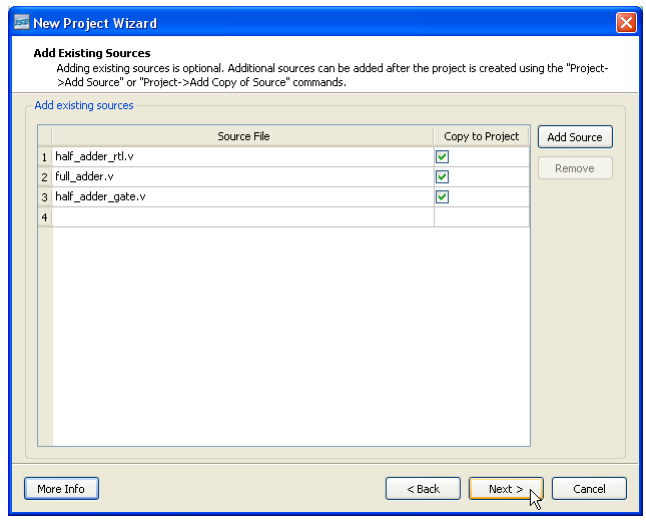
Appendix: Synthesis with ISE GUI

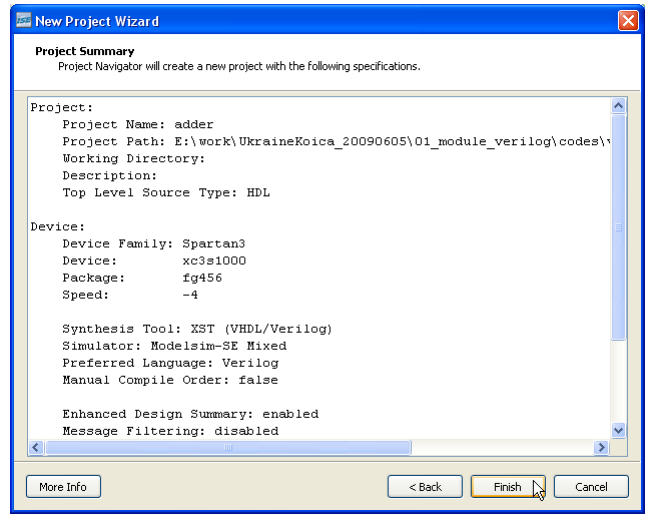


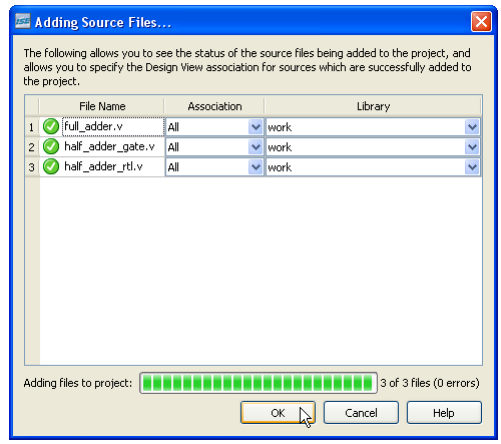









New Project Wizard
Project Summary
Project Navigator will create a new project with the following specifications.
Project:
Project Name: adder
Project Path: E:\work\UkraineKoica_20090605\01_module_verilog\codes\
Working Directory:
Description:
Top Level Source Type: HDL
Device:
Device Family: Spartan3
Device: xc3s1000
Package: fg456
Speed: -4
Synthesis Tool: XST (VHDL/Verilog)
Simulator: Modelsim-SE Mixed
Preferred Language: Verilog
Manual Compile Order: false
Enhanced Design Summary: enabled
Message Filtering: disabled
More Info
< Back
Finish
Cancel


Adding Source Files...
The following allows you to see the status of the source files being added to the project, and allows you to specify the Design View association for sources which are successfully added to the project.
File Name
Association
Library
1 full_adder.v All work
2 half_adder_gate.v All work
3 half_adder_rtl.v All work
Adding files to project:
3 of 3 files (0 errors)
OK
Cancel
Help