XILINX®

Application Note

Foundation Express
Application Note Supplement

January 1998

0401721

Foundation Express

Contents

Chapter 1

Introduction

Foundation™ Express is a synthesis tool that utilizes the Synopsys FPGA Express® technology. This application note will help you to use Foundation Express in conjunction with the Foundation Series 1.4 design entry and implementation tools.

The FND-BSX and FND-EXP packages contain the new Foundation Express software. If you are upgrading from the previous 1.3 packages, then you need to update your license.dat file. Xilinx provides a template file with the new package definitions. Refer to the "Foundation Express Installation and Security" chapter in the *Foundation Express User Guide* for details.

Foundation Express is delivered in the enclosed CD-ROM.

Contents

The Foundation Express package contains the following items:

- *Foundation Express Application Note Supplement*
- *FPGA Express Users Guide*
- Foundation Express cover letter and hot sheet
- Foundation Express 2.0 CD

Features

Foundation Express allows Foundation customers to use Xilinx's new HDL synthesis technology (provided by Synopsys).

The major new features of Foundation Express include the following:

- high-quality synthesis for all Xilinx device families
  a) XC3000A / L, XC3100A / L
  b) XC4000E / EX / XL / XV
  c) XC5200
  d) Spartan
  e) XC9500
- VHDL and Verilog support
  a) improved Verilog black box support

     To instantiate a module in Verilog, you simply define an empty module containing the names and directions for the module's I / O pins and then instantiate the module. The instances of the module are automatically stored in the final netlist.

  b) updated faster HDL analyzers

     The latest (V)HDL analyzer from Synopsys Design Compiler version 1997.08 contains bug fixes and additional construct support.
- graphical constraint entry
  a) customizable timing groups
  b) improved overall system performance requirements
  c) You can also specify timing constraints on arbitrary timing paths, including multi-cycle paths. You can create a subpath by right-clicking on a path in the pre-optimized implementation.

- new Interactive Graphical Timing Analyzer

  Foundation Express includes a timing analyzer for timing report and debugging, which is built on the Paths and Ports constraint tables. After optimization, the timing verifier displays the following:

  a) The timing for both path groups and subpaths (Paths page). Double click on the path icon to display or hide the subpaths of a timing path. Clicking a path or subpath shows the worst delays to all endpoints in the To group. Clicking an endpoint shows the complete critical path from the start to endpoints.

  b) The input-to-clock and clock-to-output timing (Ports page) (Timing analyzer support is not available for the Xilinx XC9500.)

- automatic finite state machine (FSM) optimization

  Automatic FSM encoding is supported for enumerated types (VHDL); use the VHDL template to design your FSM, then choose One Hot or Binary encoding under **`Synthesis`** → **`Options`** → **`Project`**.

- improved graphical user interface

  The graphical user interface (GUI) has a new look and feel. In addition to the Design Sources window and the Chip window, the Error / Warnings window is always displayed. Several activities, such as entering constraints or viewing results, are started by clicking the right mouse button. Double clicking on icons expands or contracts the file or design hierarchy. For VHDL, libraries other than WORK can be created by clicking the right mouse button on the project icon and selecting New Library.

- built-in source editor

  The editor interacts with analysis errors and lets you edit source files. You start the editor and open the file by selecting the design file icon, clicking the right mouse button, and selecting Edit File. You can use a menu of editing options by clicking the right mouse button.

- M1/XACT option

  You can target either M1 or XACT. Targeting M1 turns off the HBLKNM setting. HBLKNM is used on some look-up tables to improve the quality of result when using XACT. The switch to control this feature is under the Xilinx Options spreadsheet of the pre-optimized implementation.

- choice of optimization parameters

  You can synthesize a design for high speed or minimum area. The switch to control this feature is under the Create Implementation window. Also with the Low/High effort switch, you can control the CPU effort for the optimization engine. Low effort runs faster, though High effort provides better quality of results. The switch to control this feature is also in the Create Implementation window.

- GSR mapping for designs including black boxes

  Foundation Express can now infer the global set reset (GSR) even for designs that contain black boxes (when allowed by the selected device). The switch, which is called Ignore Unlinked Cells During GSR Mapping, can be found by selecting **`Synthesis`** → **`Edit Constraints`** → **`Xilinx Options`** of the pre-optimized implementation.

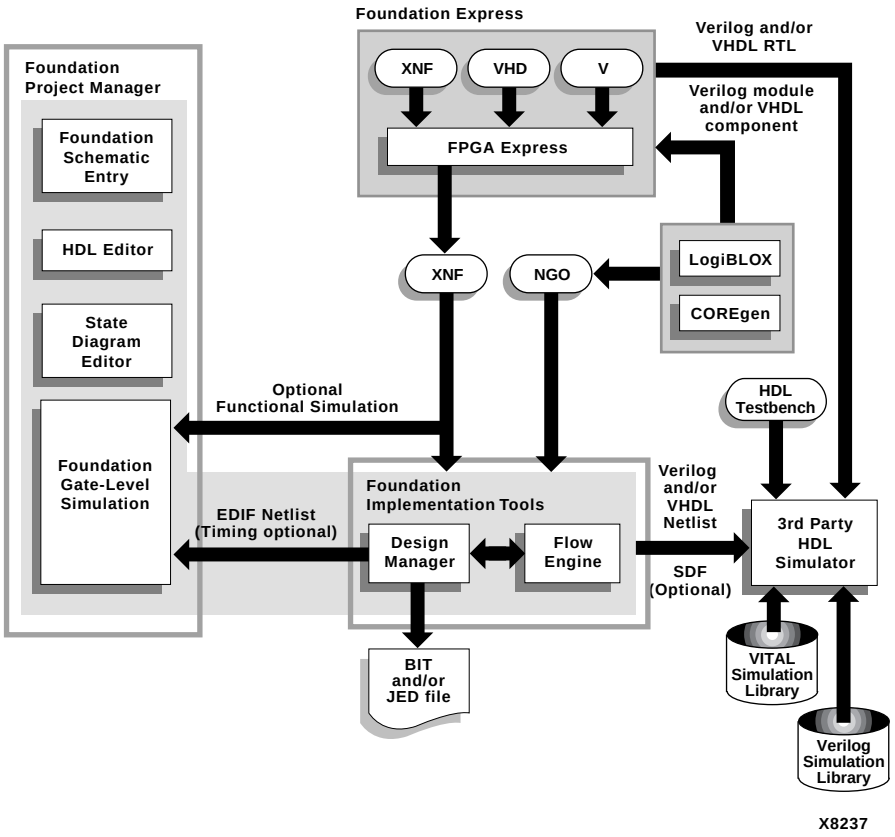
- enhanced report file

  The report file contains much more detailed information on the pre-optimized as well as the post-optimized design. Detailed information includes the design's hierarchy, the inferred operators, the cell count, the timing constraints, and the clock speed estimates.

If Foundation Express is installed, the HDL Editor provides color-coding and language assistance for the Verilog language. However, you cannot synthesize your designs within the HDL Editor. These designs must be synthesized using Foundation Express.

Design Flow

Following is an overall flow chart of the design process.



Chapter 2

Black Box Instantiation

This chapter describes how to create modular designs using Foundation 1.4 with Foundation Express.

LogiBLOX RAM Modular Design Using VHDL

This section explains how to instantiate a LogiBLOX module into a VHDL design using Foundation 1.4 with Foundation Express.

1. Using LogiBLOX, create a RAM module. LogiBLOX for Foundation Express must be started outside of Foundation so that you can set the vendor. To start LogiBLOX, select **`Start`** → **`Programs`** → **`Xilinx Foundation Series`** → **`LogiBLOX`**. When setting up LogiBLOX in the Setup window, set the Vendor to Synopsys and the Bus Notation as B<I>, as shown in the following figure. You must also define the Project Directory or LogiBLOX will build modules in *default_directory*\bin\nt.

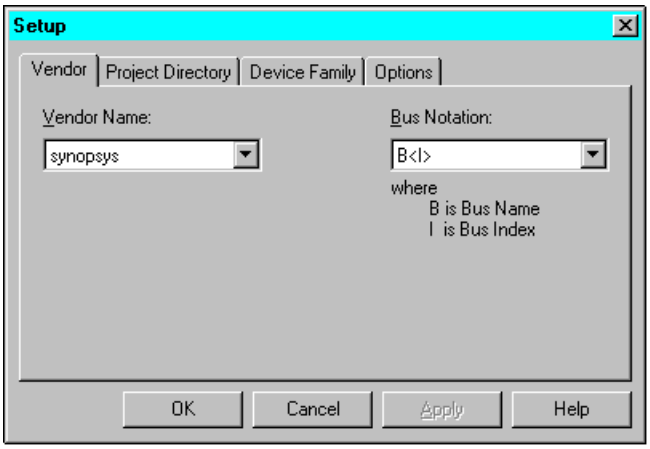


Figure 2-1 LogiBLOX Vendor Setup

In this example, a RAM48X4S is created. Within the Options tab in the Setup window, select VHDL template and NGO file, as shown in the following figure.

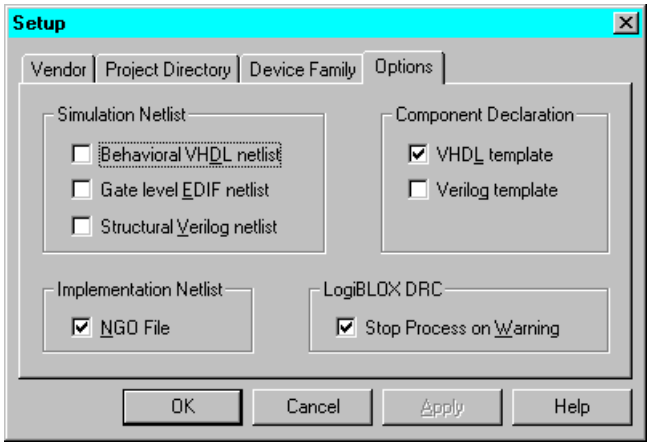


Figure 2-2 LogiBLOX Options Setup

2. Next, define the type of LogiBLOX module and its attributes. The Module Name specified for the LogiBLOX module is used as the name of the instantiation in the VHDL code.

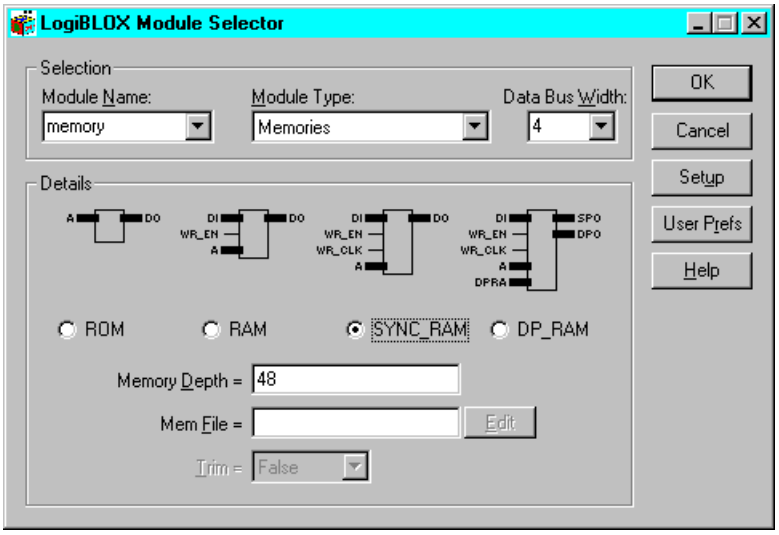


Figure 2-3 LogiBLOX Module Selector GUI

When the OK button is pressed, the LogiBLOX module is created. This module is a collection of several files, including an .ngo file, (the file merged by NGDBuild) and a .vhi file (used as an instantiation reference). Make sure the .ngo file is located in your Xilinx project directory.

```
------------------------------------------------------------
-- LogiBLOX SYNC_RAM Module "Memory"
-- Created by LogiBLOX version M1.4.12
--     on Fri Dec 19 11:50:38 1997
-- Attributes
--     MODTYPE = SYNC_RAM
--     BUS_WIDTH = 4
--     DEPTH = 48
------------------------------------------------------------
------------------------------------------------------------
-- Component Declaration
------------------------------------------------------------
component memory
  PORT(
    A:  IN std_logic_vector (5 DOWNTO 0);
    DI:  IN std_logic_vector (3 DOWNTO 0);
    WR_EN:  IN std_logic;
    WR_CLK:   IN std_logic;
    DO:  OUT std_logic_vector (3 DOWNTO 0));
 end component;
 ------------------------------------------------------------
 -- Component Instantiation
 ------------------------------------------------------------
 instance_name  : memory port map
 (A => ,
 DI => ,
 WR_EN => ,
 WR_CLK => ,
 DO => );
```

Figure 2-4 .VHI File Created by LogiBLOX

The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vhi file.

3. Using the .vhi file as a reference, write your VHDL code to instantiate the LogiBLOX RAM module. See the following figure as an example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


entity top is
port (           D: in STD_LOGIC; CE: in STD_LOGIC;
               CLK: in STD_LOGIC; Q: out STD_LOGIC;
              Atop: in STD_LOGIC_VECTOR (5 downto 0);
             DOtop: out STD_LOGIC_VECTOR (3 downto 0);
             DItop: in STD_LOGIC_VECTOR (3 downto 0);
          WR_ENtop: in STD_LOGIC;
         WR_CLKtop: in STD_LOGIC);
end top;

architecture inside of top is

component userff
port (    D: in STD_LOGIC; CE: in STD_LOGIC;
        CLK: in STD_LOGIC; Q:  out STD_LOGIC);^M
end component;

component memory
port (     A:  in STD_LOGIC_VECTOR (5 downto 0);
           DI:  in STD_LOGIC_VECTOR (3 downto 0);
           WR_EN:  in STD_LOGIC;
           WR_CLK:   in STD_LOGIC;
           DO:  out STD_LOGIC_VECTOR (3 downto 0));
end component;

begin

U0:  userff port map (D=>D, CE=>CE, CLK=>CLK, Q=>Q);

U1: memory port map (A=>Atop, DI=>DItop, WR_EN=>WR_ENtop,
                WR_CLK=>WR_CLKtop, DO=>DOtop);
end inside;
```

Figure 2-5 VHDL File With LogiBLOX Instantiation

For each .ngo file from LogiBLOX, you may have one or more VHDL files with the .ngo file instantiated. In this example, there is only one black box instantiation of memory, but multiple calls to the same module may be done.

4. Read, update (**`Synthesis`** → **`Update`**), and then implement (**`Synthesis`** → **`Create Implementation`**) the design in Foundation Express.

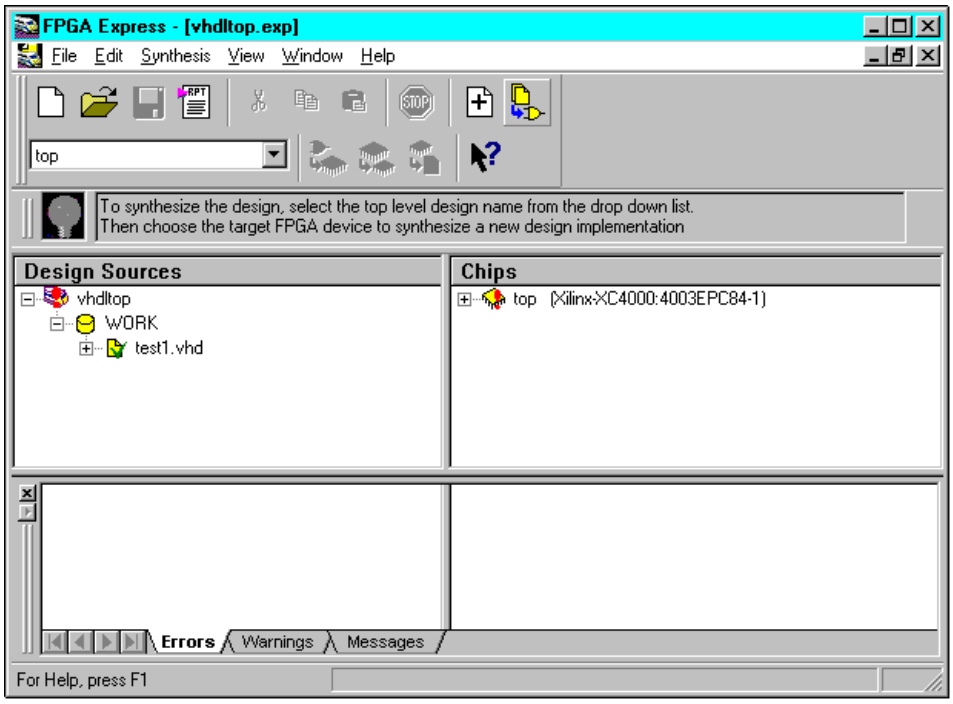


Figure 2-6 Design Implemented Within Foundation Express

5. (Optional) After implementing the design, select the implemented design from the Chips window and then select **`Synthesis`** → **`Edit Constraints`**. Set all constraints for the design within the listed tabs. Close the Implementation window. For more information about constraints, refer to the "Timing Constraints with Foundation Express" chapter.

6. Optimize the design by selecting **`Synthesis`** → **`Optimize Chip`**.

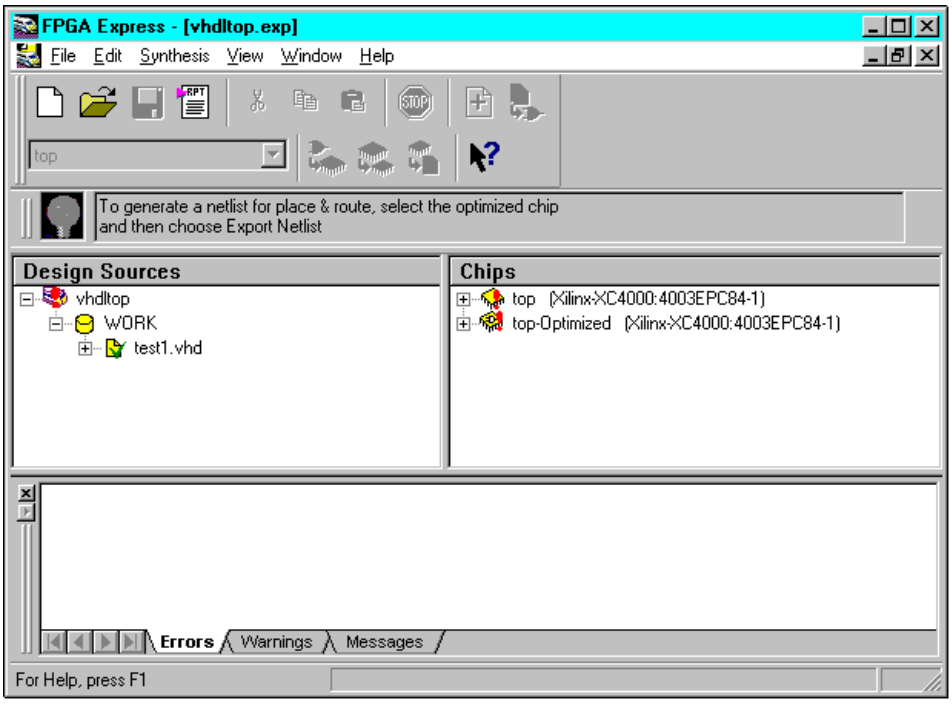


Figure 2-7 Design Optimized Within Foundation Express

7. With the optimized design selected, write out the XNF file by selecting **`Synthesis`** → **`Export Netlist`**. Express does not write out an XNF file for the instantiated LogiBLOX component.

8. Take the XNF files written by Express and the .ngo files written by LogiBLOX and process the design through the Xilinx Design Implementation software.

LogiBLOX RAM Modular Design Using Verilog

This section explains how to instantiate a LogiBLOX module into a Verilog design using Foundation 1.4 with Foundation Express.

1. Using LogiBLOX, create a RAM module. LogiBLOX for Foundation Express must be started outside of Foundation so that you can set the vendor. To start LogiBLOX, select **`Start`** → **`Programs`** → **`Xilinx Foundation Series`** → **`LogiBLOX`**. When setting up LogiBLOX in the Setup window, set the Vendor to Synopsys and the Bus Notation as B<I>, as shown in the following figure. You must also define the Project Directory or LogiBLOX will build modules in *default_directory*\bin\nt.

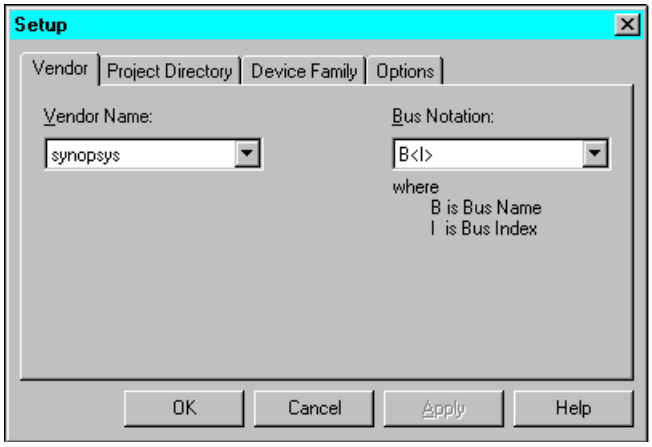


Figure 2-8 LogiBLOX Vendor Setup

2. In this example, a RAM48X4S is created. Within the Options tab in the Setup window, select Verilog template and NGO file, as shown in the following figure.

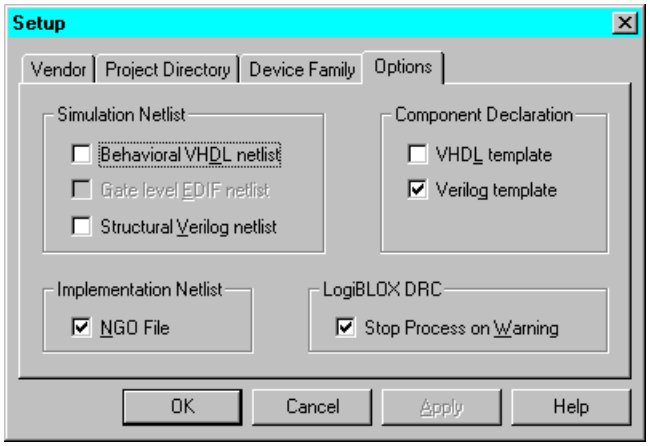


Figure 2-9 LogiBLOX Options Setup

3. Next, define the type of LogiBLOX module and its attributes. The Module Name specified for the LogiBLOX module is used as the name of the instantiation in the Verilog code.

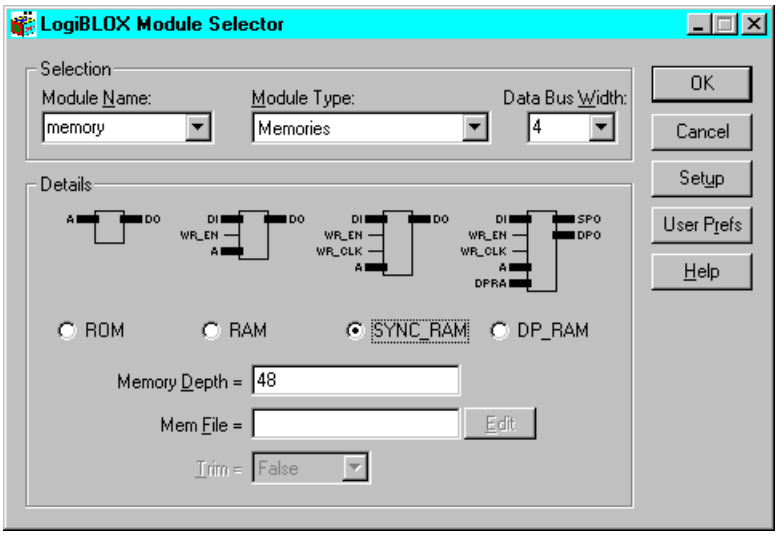


Figure 2-10 LogiBLOX Module Selector GUI

When the OK button is selected, the LogiBLOX module is created. This process creates a number of files, including an .ngo file, which is a file merged by NGDBuild, and a .vei file, which is used as an instantiation reference.

```
//-----------------------------------------------------
// LogiBLOX SYNC_RAM Module "memory"
// Created by LogiBLOX version M1.4.12
//    on Fri Dec 19 14:56:42 1997
// Attributes
//     MODTYPES = SYNC_RAM
//     BUS_WIDTH = 4
//     DEPTH = 48
//-----------------------------------------------------
memory instance_name
( .A () ,
 .DO () ,
.DI () ,
.WR_EN () ,
.WRCLK () );

module memory (A, DO, DI, WR_EN, WR_CLK);
input  [5:0] A;
output  [3:0] DO;
input  WR_EN;
input WR_CLK;
endmodule
```

Figure 2-11 .VEI File Created by LogiBLOX

4. Using the .vei file as a reference, write your Verilog code to instantiate the LogiBLOX RAM module. Refer to the "Verilog File With LogiBLOX Instantiation" figure.

   The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vei file.

```
module top (D,CE,CLK,Q,
            Atop, DOtop, DItop, WR_ENtop, WR_CLKtop);

input D;
input CE;
input CLK;
output Q;

input  [5:0] Atop;
output  [3:0] DOtop;
input  [3:0] DItop;
input WR_ENtop;
input WR_CLKtop;

userff U0  (.D(D),.CE(CE),.CLK(CLK),.Q(Q));

memory U1  (      .A(Atop),
              .DO (DOtop),
              .DI (DItop),
              .WR_EN (WR_ENtop),
              .WR_CLK (WR_CLKtop));
endmodule
```

Figure 2-12 Verilog File With LogiBLOX Instantiation

You now have an .ngo file from LogiBLOX, and one or more Verilog files with the .ngo file instantiated. In this example, there is only one instantiation of “memory”, but multiple calls to the same module may be done.

5. Using the .vei file as a reference, create an empty Verilog module to represent the black box module. See the following figure.

```
//-----------------------------------------------
// LogiBLOX SYNC_RAM Module "MEMORY"
// Created by LogiBLOX version M1.4.12
//    on Fri Dec 19 14:56:42 1997
// Attributes
//     MODTYPE = SYNC_RAM
//     BUS_WIDTH = 4
//     DEPTH = 48
//-----------------------------------------------
module MRMORY (A, DO, DI, WR_EN, WR_CLK);
 input  [5:0] A;
 output  [3:0] DO;
 input [3:0] DI;
 input  WR_EN;
 input WR_CLK;
endmodule
```

Figure 2-13 Empty Verilog Module Representing Black Box Module

6. Read, update (**Synthesis** → **Update**), and then implement (**Synthesis** → **Create Implementation**) the design in Foundation Express.

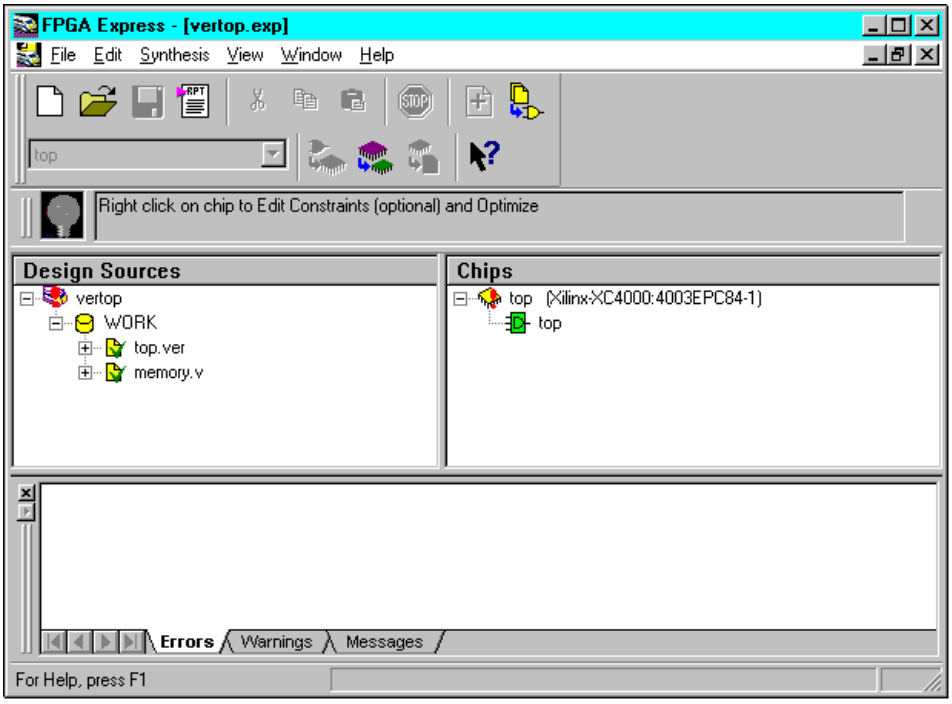


Figure 2-14 Design Implemented Within Foundation Express

7. (Optional) After implementing the design, select the implemented design from the Chips window and then select **Synthesis** → **Edit Constraints**. Set all constraints for the design within the listed tabs. Close the Implementation window. For more information about constraints, refer to the "Timing Constraints with Foundation Express" chapter.

8. Optimize the design by selecting **Synthesis** → **Optimize Chip**.

9. Write out the XNF file by selecting **Synthesis** → **Export Netlist**. Express does not write out an XNF file for the instantiated LogiBLOX component.

10. Take the XNF files written by Express and the .ngo files written by LogiBLOX and process the design through the Xilinx Design Implementation software.

Instantiating an XNF file in VHDL or Verilog

This section explains how to instantiate an XNF file into a VHDL or Verilog design using Foundation 1.4 and Foundation Express. This procedure only works for Unified Library XNF files.

1. Open the XNF file with a text editor. Search for the string LCANET.

   The LCANET line must be either LCANET, 5 or LCANET, 6. If the search for LCANET turns up an LCANET, 4 or earlier, this XNF file cannot be used in the M1 flow. Please see the *M1 Conversion Guide* on the Xilinx website for details on handling pre-Unified XNF files (LCANET, 4 or earlier).

   The name of the XNF file must be the name of the "component" instantiation in the VHDL code or the name of the "module" instantiation in the Verilog code.

2. To attach the XNF module in the VHDL / Verilog code, use the nets named in the PIN records and / or SIG records in the XNF file as the port names of the component instantiation. The following figure shows an example XNF file with PIN and SIG records.

```
SYM, current_state_reg<4>, DFF, LIBVER=2.0.0
PIN, D, I, next_state<4>, ,
PIN, C, I, N10, ,
PIN, Q, O, current_state<4>, ,
END
SIG, current_state<4>
SIG, CLK, I, ,
SIG, DATA, I, ,
SIG, SYNCFLG, O, ,
```

Figure 2-15 Portion of XNF File

To reference buses in the instantiation of XNF modules, the nets named in PIN records and / or SIG records must be of the form.

netname<number>

This designation allows the bus to be referenced in the VHDL component as a vector data type.

3. Using the filename of the XNF file as the name of the component and the name of nets in the XNF file as port names, instantiate the XNF file in the VHDL / Verilog code.
4. Take all VHDL / Verilog design files, read, update (**Synthesis** → **Update**), and then implement (**Synthesis** → **Create Implementation**) the design in Foundation Express.
5. (Optional) After implementing the design, select the implemented design from the Chips window and then select **Synthesis** → **Edit Constraints**. Set all constraints for the design within the listed tabs. Close the Implementation window. For more information about constraints, refer to the "Timing Constraints with Foundation Express" chapter.
6. Optimize the design by selecting **Synthesis** → **Optimize Chip**.
7. With the optimized design selected, write out the XNF file by selecting **Synthesis** → **Export Netlist**. Express does not write out an XNF file for the instantiated LogiBLOX component.
8. Take the XNF file from Express and XNF file instantiated in the VHDL / Verilog code and process with the Xilinx Design Implementation software.

Instantiating an EDIF file into an HDL Design

This procedure shows how to instantiate a purely schematic design from Foundation into Foundation Express. This procedure uses Foundation Express as the top level design with the Foundation Schematic Editor as the module generator.

1. Create a project in the Foundation Project Manager. Name the project you want to use for instantiating the schematic in the HDL code. If the project name is "big", then in Verilog the module name for the instantiated schematic is "big".
2. Create a schematic in the Foundation Schematic Editor. This schematic must not have I / O library components like IBUF, OBUF, OBUFT. In the schematic, use I / O terminals in place of I / O components. The name of the I / O terminals will be the name of the pins for the instantiation in the HDL code. The schematic may contain hierarchy.

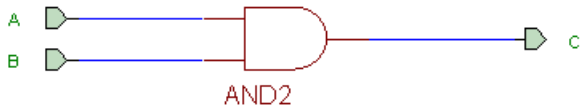


Figure 2-16 test1 Design with I/O Terminals

3. After creating the schematics for the design, create an EDIF file by selecting **`Options`** → **`Export Netlist`**.

   This step creates an EDIF file in the Foundation Project Manager. The EDIF file will have the extension .edn, and the project name will be the name of the EDIF file.

4. Instantiate the Foundation schematic.

   a) For Verilog, instantiate the Foundation schematic, using the project name as the instantiated module name and the name of the I/O terminals as the name of the pin. In this example, the EDIF file was created in a project called test1.

```
module test (FIRST,SECOND,THIRD);


input FIRST;
input SECOND;
output THIRD;


test1 U1 (.A(FIRST),.B(SECOND),.C(THIRD));


endmodule
```

Figure 2-17 Verilog Instantiation Example

   b) For VHDL, instantiate the Foundation schematic, using the project name as the instantiated module name and the name of the I/O terminals as the name of the pin. In this example, the EDIF file was created in a project called test1.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;


entity test is
port (   FIRST: in STD_LOGIC;
        SECOND: in STD_LOGIC;
         THIRD: out STD_LOGIC);
end test;

architecture inside of test is

component test1
port (A: in STD_LOGIC; B: in STD_LOGIC; C: out STD_LOGIC);
end component;

begin

U1: test1 port map(A=>FIRST,B=>SECOND,C=>THIRD);

end inside;
```

Figure 2-18 VHDL Instantiation Example

5. Take all VHDL / Verilog design files, read, update (**Synthesis** → **Update**), and then implement (**Synthesis** → **Create Implementation**) the design in Foundation Express.
6. After implementing the design, select the implemented design from the Chips window and then select **Synthesis** → **Edit Constraints**.
7. Set all constraints for the design within the listed tabs. Close the Implementation window. For more information about constraints, refer to the "Timing Constraints with Foundation Express" chapter.
8. Optimize the design by selecting **Synthesis** → **Optimize Chip**.
9. Write out the XNF file by selecting **Synthesis** → **Export Netlist**.
10. Take XNF files from Express and the EDN file from Foundation and process with the Xilinx Design Implementation software. All files must be located in the same directory before processing.

Chapter 3

Timing Constraints with Foundation Express

This chapter discusses the following:

- Constraints that Foundation Express can directly apply
- An overview of how to use M1 constraints with Foundation Express as the top-level design tool or as a module generator for Foundation schematics
- Information on how to reference instance names in XNF files from Foundation; when constraints are needed that Foundation Express cannot apply, a UCF file referencing instance names in the XNF file must be made.
- Information on how to reference the instances inside LogiBLOX RAM/ROM modules for creating UCF constraints

Applying Constraints with the Express GUI

After a design has been implemented in Foundation Express, M1 constraints can be applied to a design through three tabs in the Implementation window. With the implemented design highlighted in the Chips window, select **`Synthesis`** → **`Edit Constraints`**. See the next two figures.

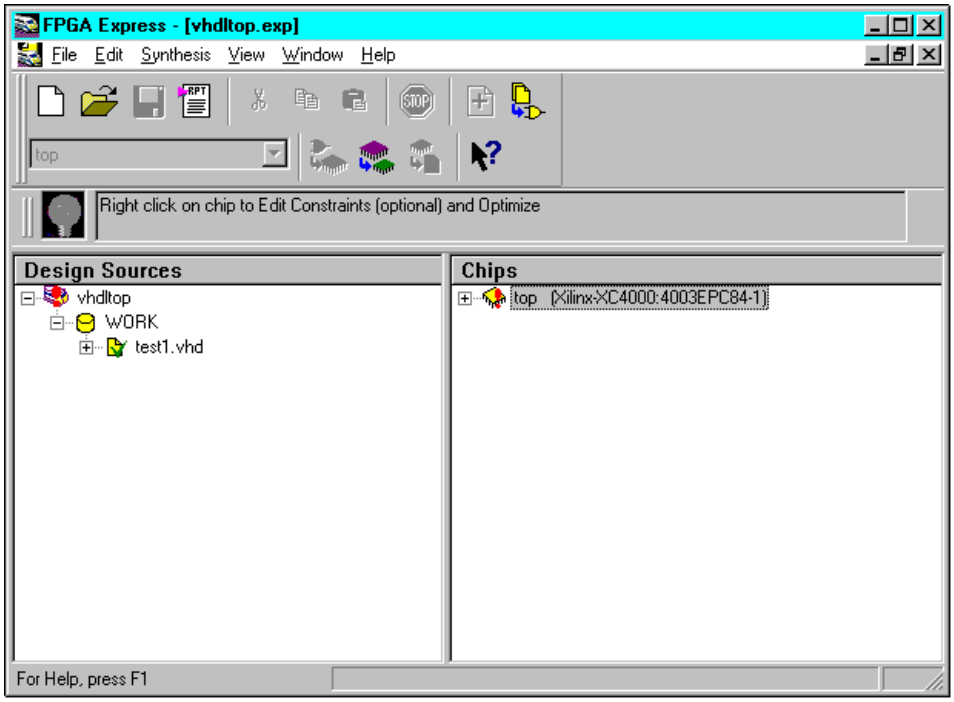


Figure 3-1 Implemented Design

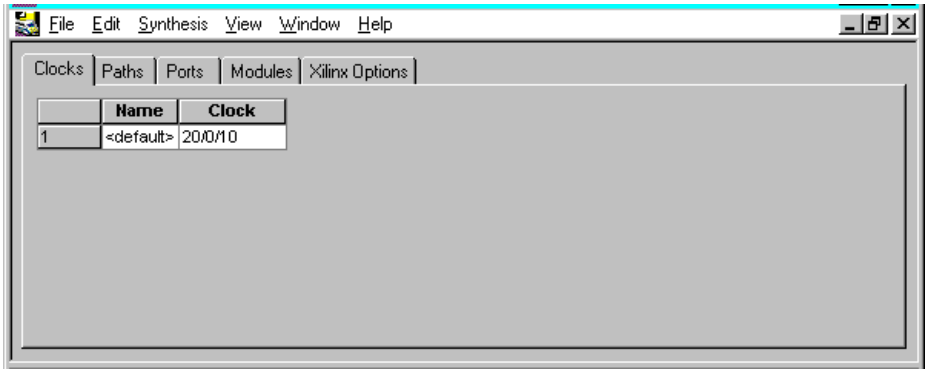


Figure 3-2 Constraints Spreadsheets

The Implementation window shown in the preceding figure has three different tabs where M1 constraints can be applied: Clock, Paths, and Ports.

- The Clocks tab allows you to specify overall speeds for the clocks in a design.
- The Paths tab allows you precise control of point-to-point timing in a design.
- The Ports tab allows OFFSETS, pullups/pulldowns, and pin locations to be specified in a design.

The timing constraints specified in the Implementation window are translated into FROM:TO timespecs and placed in the XNF file Express output. Following is an example:

```
SYM, TS0, TIMESPEC, TS0=from:pads:to:tgrp_0_DFF=40ns, LIBVER=2.0.0
END
```

Currently, Express cannot apply all Xilinx M1 constraints to a netlist.

Constraints that Express can apply:

- FROM:TO timespecs which use FFS, LATCHES, and PADS
- Pin location constraints
- Slew rate
- Pullup / Pulldown

Constraints that Express cannot apply:

- TPSYNC
- TPTHRU
- TIG
- OFFSET:IN:BEFORE
- OFFSET:OUT:AFTER
- user-RLOCs, RLOC_ORIGIN, RLOC_RANGE
- non-I/O LOCs
- KEEP
- U_SET,H_SET,HU_SET
- user-BLKNM and user-HBLKNM
- PROHIBIT

Express can create its own timegroups by grouping logic with common clocks and clock enables. In addition, you can form user-created timing subgroups by right clicking on an existing timing path and choosing New Sub Path.

Xilinx Logical Constraints

In the M1 design flow, constraints or attributes that can be applied within a schematic, netlist, or UCF file are known as logical constraints. For constraints that cannot be applied using the constraint GUI, a UCF file can be used to specify logical constraints. Logical constraints restrict the placement or timing of logic in an FPGA or CPLD design. In order to use a logical constraint correctly, the "instance" name of the logic in a design must be used. Instance names are XNF SYM record names, XNF SIG record names, XNF net names, and EXT record names. For examples of reading these instance names out of a XNF file from Express, refer to the following figure.

```
SYM, current_state_reg<4>, DFF, LIBVER=2.0.0
PIN, D, I, next_state<4>, ,
PIN, C, I, N10, ,
PIN, Q, O, current_state<4>, ,
END
SIG, current_state<4>
EXT, CLK, I, ,
EXT, DATA, I, ,
EXT, SYNCFLG, O, ,
```

Figure 3-3 XNF example

In the preceding figure, the SYM record name can be referenced by a logical constraint by using the instance name, current_state_reg<4>. A net called N10 or current_state<4> can also be used in a logical constraint. EXT records correspond to pins used on a package. The EXT records named CLK, DATA, and SYNCFLG can be referenced in a pin locking constraint.

For more information on M1 constraints, refer to the “Attributes, Constraints, and Carry Logic” chapter in the *Libraries Guide.*

Reading Instance Names from an XNF file for UCF Constraints

In M1, UCF constraints are applied by referencing instance names that are found in the XNF file. Instance names for logic in a design can be found by reading the XNF file. Refer to XNF syntax in the "XNF example" figure for the examples in this section. The following examples illustrate valid entries within a UCF file.

- A TNM constraint can be applied to an FF by using the instance name from the XNF file. Similarly, a LOC / RLOC can be applied:

  ```
  INST "current_state_reg<4>" TNM=group1;
  INST "current_state_reg<4>" LOC=CLB_R5C5;
  ```

  By attaching a TNM to this flip-flop instance name, this flip-flop can be referenced in a FROM:TO timing specification. Any symbol that can have an M1 constraint applied is referenced by using the string following the keyword: SYM.

- A pin on a device may be locked to a package-specific position by referencing the EXT record name and adding the .PAD string:

  ```
  INST "DATA.PAD" LOC=P124;
  ```

- An attribute which can be placed on a net, like KEEP or TNM, can be referenced by referencing the netname on the PIN record or SIG record:

  ```
  NET "current_state<4>" KEEP;
  NET "current_state<4>" TNM=group2;
  ```

A final note on referencing instance names from a XNF file: match the case. M1 is case-sensitive. If the case of names in the XNF file is not followed exactly, the M1 implementation software may not be able to find (or may incorrectly find) an instance name for a constraint.

Instance Names for LogiBLOX RAM/ROM

In the Foundation Express methodology, whenever large blocks of RAM/ROM are needed, LogiBLOX RAM/ROM modules should be instantiated by the user in the HDL code. With LogiBLOX RAM/ROM modules instantiated in the HDL code, timing and/or placement constraints on these RAM/ROM modules and the RAM/ROM primitives that comprise these modules, are specified in a .ucf file.

To create timing and/or placement constraints for RAM/ROM LogiBLOX modules, you must know how many primitives are used and how the primitives inside the RAM/ROM LogiBLOX modules are named.

Calculating Primitives for a LogiBLOX RAM/ROM Module

When a RAM/ROM is specified with LogiBLOX, the RAM/ROM depth and width are specified. If the RAM/ROM depth is divisible by 32, then 32x1 primitives are used. If the RAM/ROM depth is not divisible by 32, then 16x1 primitives are used instead. In the case of dual-port RAMs, 16x1 primitives are always used. Based on whether 32x1 or 16x1 primitives are used, the number of RAM/ROMs primitives can be calculated.

For example, if a RAM48x4 was required for a design, RAM16x1 primitives would be used. Based on the width, there would be four banks of RAM16x1's. Based on the depth, each bank would have three RAM16x1's.

Naming Primitives in LogiBLOX RAM/ROM Modules

Using the example of a RAM48x4, the RAM primitives inside the LogiBLOX would be named as follows:

```
MEM0_0    MEM1_0    MEM2_0    MEM3_0
MEM0_1    MEM1_1    MEM2_1    MEM3_1
MEM0_2    MEM1_2    MEM2_2    MEM3_2
```

Each primitive in a LogiBLOX RAM/ROM module has an instance name of MEMx_y, where y represents the primitive position in the bank of memory, and where x represents the bit position of the RAM/ROM output.

Referencing LogiBLOX Entities

This section is written in terms of the Verilog example, using the files illustrated in Figures 3-4 through 3-7. This section also applies to the VHDL example in Figures 3-8 through 3-11. For information on compiling these examples, see the "Black Box Instantiation" chapter.

LogiBLOX RAM/ROM modules in the M1 Foundation Express flow are constrained using a UCF file.

LogiBLOX RAM/ROM modules instantiated in the HDL code can be referenced by the complete hierarchical instance name. If a LogiBLOX RAM/ROM module is at the top-level of the HDL code, then the instance name of the LogiBLOX RAM/ROM module is just the instantiated instance name. In the case of a LogiBLOX RAM/ROM that is instantiated within the hierarchy of the design, the instance name of the LogiBLOX RAM/ROM module is the full hierarchical path to the LogiBLOX RAM/ROM. The hierarchy level names are listed from the top level down and are separated by a "_".

In the Verilog example, the RAM32X1S is named "memory". The memory module is instantiated in the Verilog module "inside" with an instance name "U1". "inside" is instantiated in the top-level module "test" with an instance name "U0". Therefore, the RAM32X1S can be referenced in a UCF file as "U0_U1". For example, to attach a TNM to this block of RAM, the following line could be used in the UCF file:

```
INST "U0_U1" TNM=block1;
```

Since U0_U1 is composed of two RAM primitives, a timegroup called block1 is created; the block1 TNM can be used throughout the UCF file as a timespec end/start point, and/or U0_U1 could have a LOC area constraint applied to it. If the RAM32X1S has been instantiated in the top-level file and the instance name used in the instantiation is U1, then this block of RAM can just be referenced by U1.

Sometimes it is necessary to apply constraints to the primitives that compose the LogiBLOX RAM/ROM module. For example, if you choose a floorplanning strategy to implement your design, it may be necessary to apply LOC constraints to one or more primitives inside a LogiBLOX RAM/ROM module. Consider the RAM32X2S example. Suppose that each of the RAM primitives needs to be constrained to a particular CLB location.

Based on the rules for determining the MEMx_y instance names, using the example from above, each of the RAM primitives can be referenced by concatenating the full-hierarchical name to each of the MEMx_y names. The RAM32x2S created by LogiBLOX will have primitives named MEM0_0 and MEM1_0. So, CLB constraints in a .ucf file for each of these two items would be:

```
INST “U0_U1/MEM0_0” LOC=CLB_R10C10;
INST “U0_U1/MEM0_1” LOC=CLB_R11C11;
```

In the following figure, the LogiBLOX module is contained in the “inside UO” component.

test.v:

```
module test(DATA,DATAOUT,ADDR,C,ENB);
input   [3:0] DATA;
output  [3:0] DATAOUT;
input   [5:0] ADDR;
input   C;
input ENB;
wire [3:0] dataoutreg;
reg [3:0] datareg;
reg [3:0] DATAOUT;
reg [5:0] addrreg;

inside UO (.MDATA(datareg),.MDATAOUT(dataoutreg),.MADDR(addrreg),.C(C),.WE(ENB));

always@(posedge C)
        datareg = DATA;

always@(posedge C)
               DATAOUT = dataoutreg;

always@(posedge C)
        addrreg = ADDR;
endmodule
```

Figure 3-4 Top-level Verilog File

The following figure illustrates the instantiated LogiBLOX module, “memory U1”.

```
inside.v:

module inside(MDATA,MDATAOUT,MADDR,C,WE);

input   [3:0] MDATA;
output  [3:0] MDATAOUT;
input   [5:0] MADDR;
input   C;
input WE;

memory  U1
( .A(MADDR),
 .DO(MDATAOUT),
 .DI(MDATA),
 .WR_EN(WE),
 .WR_CLK(C));

endmodule
```

Figure 3-5 Verilog File with Instantiated LogiBLOX Module

When the LogiBLOX module is created, a .vei file is created, which is used as an instantiation reference.

```
//-----------------------------------------------------
// LogiBLOX SYNC_RAM Module "memory"
// Created by LogiBLOX version M1.4.12
//    on Fri Dec 19 14:56:42 1997
// Attributes
//     MODTYPES = SYNC_RAM
//     BUS_WIDTH = 4
//     DEPTH = 48
//-----------------------------------------------------
memory instance_name
( .A () ,
 .DO () ,
.DI () ,
.WR_EN () ,
.WRCLK () );

module memory (A, DO, DI, WR_EN, WR_CLK);
input  [5:0] A;
output  [3:0] DO;
input  WR_EN;
input WR_CLK;
endmodule
```

Figure 3-6 VEI File Created by LogiBLOX

```
test.ucf:

INST UO_U1 TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST UO_U1/mem0_0 LOC=CLB_R7C2;
```

Figure 3-7 UCF File for Verilog Example

```
test.vhd:

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity test is
        port(   DATA: in STD_LOGIC_VECTOR(3 downto 0);
                DATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
                ADDR: in STD_LOGIC_VECTOR(5 downto 0);
                C, ENB: in STD_LOGIC);
end test;

architecture details of test is
signal dataoutreg,datareg: STD_LOGIC_VECTOR(3 downto 0);
signal addrreg: STD_LOGIC_VECTOR(5 downto 0);

component inside
port(   MDATA: in STD_LOGIC_VECTOR(3 downto 0);
        MDATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
        MADDR: in STD_LOGIC_VECTOR(5 downto 0);
        C,WE: in STD_LOGIC);
end component;
begin
U0: inside port
map(MDATA=>datareg.,MDATAOUT=>dataoutreg.,MADDR=>addrreg,C=>C,WE=>ENB); map

process( C )
begin
        if(C'event and C='1') then
        datareg <= DATA;
        end if;
end process;

process( C )
begin
        if(C'event and C='1') then
        DATAOUT <= dataoutreg;
        end if;
end process;

process( C )
begin
        if(C'event and C='1') then
        addrreg <= ADDR;
        end if;
end process;

end details;
```

Figure 3-8 Top-level VHDL Example File

```
inside.vhd:

entity inside is
        port(   MDATA: in STD_LOGIC_VECTOR(3 downto 0);
                MDATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
                MADDR: in STD_LOGIC_VECTOR(5 downto 0);
                C,WE: in STD_LOGIC);
end inside;

architecture details of inside is
component memory
port(   A: in STD_LOGIC_VECTOR(5 downto 0);
        DO: out STD_LOGIC_VECTOR(3 downto 0);
        DI: in STD_LOGIC_VECTOR(3 downto 0);
        WR_EN,WR_CLK: in STD_LOGIC);
end component;

begin
U1: memory port map(A=>MADDR,DO=>MDATAOUT,DI=>MDATA,WR_EN=>WE,WR_CLK=>C);
end details;
```

Figure 3-9 VHDL File with Instantiated LogiBLOX Module

```
------------------------------------------------------
-- LogiBLOX SYNC_RAM Module "Memory"
-- Created by LogiBLOX version M1.4.12
--     on Fri Dec 19 11:50:38 1997
-- Attributes
--     MODTYPE = SYNC_RAM
--     BUS_WIDTH = 4
--     DEPTH = 48
------------------------------------------------------
------------------------------------------------------
-- Component Declaration
------------------------------------------------------
component memory
  PORT(
    A:  IN std_logic_vector (5 DOWNTO 0);
    DI:  IN std_logic_vector (3 DOWNTO 0);
    WR_EN:  IN std_logic;
    WR_CLK:   IN std_logic;
    DO:  OUT std_logic_vector (3 DOWNTO 0));
 end component;
 ------------------------------------------------------
 -- Component Instantiation
 ------------------------------------------------------
 instance_name  : memory port map
 (A => ,
 DI => ,
 WR_EN => ,
 WR_CLK => ,
 DO => );
```

Figure 3-10 .VHI File Created By LogiBLOX

```
test.ucf:

INST U0_U1 TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST U0_U1/mem0_0 LOC=CLB_R7C2;
```

Figure 3-11 UCF File for VHDL Example

Chapter 4

Simulation With Foundation Express Netlists

Foundation Express is a synthesis tool only. This chapter outlines the four simulation flows available when using Foundation Express as the top-level design tool. This is not a chapter describing the details of HDL simulation. It is assumed that readers of this chapter already have HDL simulation knowledge. For more information about simulation, refer to the *Foundation Express User Guide*. Also consult the Logic Simulator manual from the Foundation Project Manager (**`Help`** → **`Foundation Help Contents`** → **`Logic Simulator`**).

Simulation Flows in 1.4

In the M1 design methodology, there are four possible simulation flows:

- Pre-synthesis RTL simulation (functional simulation)
- Post-synthesis pre-route simulation (post-translate simulation)
- Post-map pre-route simulation (post-map simulation)
- Post-par simulation (timing simulation).

Each of these simulation flows adds greater accuracy the closer the design is to completion. By using the same testbench and comparing the results at each of the simulation stages, you can be confident that what was synthesized, and/or trimmed, matches your design specification.

Simulation of Modules/Black Box Designs

In the Foundation Express 1.4 design flow, you can instantiate black box designs if Express is the top-level tool. Black box modules instantiated in Express are LogiBLOX modules, LogiCORE, or unified XNF files. By definition, these instantiated modules cannot be simulated in the pre-synthesis RTL simulation flow, except for LogiBLOX instantiations. Pre-synthesis RTL simulation means that all behavior in the design is described in the HDL code. Instantiated LogiBLOX can be simulated in pre-synthesis RTL simulation, since the LogiBLOX tool can create behavioral HDL models which can be used for pre-synthesis RTL simulation. Instantiated black boxes without behavior, like an XNF file, can only be functionally simulated after NGDBuild has been run.

Simulation Flows for Foundation Express

The following subsections briefly describe each of the simulation flows.

Pre-Synthesis RTL Simulation

Pre-synthesis RTL simulation is traditionally known as functional simulation. The main purpose of HDL functional simulation is to determine if the behavior of the HDL is what is expected.

In the existing functional simulation flow with Express, functional simulation is possible if the HDL code does not have instantiated FFS, OBUF, and OBUFTs. Instantiated FFS, OBUF, and OBUFTs have extra pins for simulation of the GSR and GTS. Instantiated combinatorial logic, RAM/ROM primitives, or IO combinatorial logic can be simulated. Instantiated LogiBLOX can also be simulated by using the behavioral model produced by LogiBLOX. If there is no way to avoid instantiation of FFS, OBUF, and/or OBUFTs, then another type of simulation should be used: Post-synthesis pre-route simulation (post-NGDBuild).

1. Collect all HDL files for the design you wish to simulate.
2. Create a testbench file.
3. Read in HDL files and testbench files into the HDL simulation tool.

The specific procedures for this flow depend on the HDL simulator. Please consult your HDL simulator documentation for more information.

Post-Synthesis Pre-Route Simulation

Post-synthesis pre-route simulation is another simulation flow in M1. Like pre-synthesis RTL simulation, it is a functional simulation. However, unlike pre-synthesis RTL simulation, the behavior in this type of simulation is post-synthesis. With this type of flow, the synthesized logic behavior is evaluated. But unlike pre-synthesis RTL simulation, it does not matter if there are instantiated FFS, OBUF, and / or OBUFTs since NGDBuild (Translate) converts all logic in the design to simulation primitives (SIMPRIMS). Logic converted to SIMPRIMS can be simulated by the Xilinx Verilog or VITAL simulation libraries. The logic simulated in this flow has not been trimmed. It is always possible that trimming of a design could change behavior. In the M1 flow, there is no flow to verify that logical trimming has not changed the design behavior.

1. Synthesize HDL code in Foundation Express to an XNF file.
2. Open the Xilinx Design Manager by selecting **Start** → **Programs** → **Xilinx Foundation Series** → **Design Manager**. Do *not* open the Design Manager via the Foundation Project Manager.
3. Open the XNF file in the Xilinx Design Manager.

   Perform the following steps:

   a) After the Design Manager main window opens, create a new project directory by selecting **File** → **New Project**.

   b) In the New project dialog box, select the Browse button for the input design.

   c) Select XNF files from the Files of Type list box. Locate the XNF from the Look in list box. Click OK.

   d) Select **Design** → **New Version**. Enter a new version and click OK.

   e) Select **Design** → **New Revision**. Enter a new revision and click OK.

f) If you have any instantiated NGO, EDIF or XNF files in your design, make sure these files are located in the project directory.

4. Run Translate from the Design Manager to create an .ngd file. Translate produces an .ngd file with the device family name (for example, xc4000e.ngd) that is located in the version directory. Copy this file into your project directory—the one that contains the XNF file and any instantiated NGO, EDIF, XNF files.
5. Open a DOS window.
6. From the DOS command line. Run NGD2VHDL or NGD2VER on the .ngd file produced by Translate. Use the appropriate command line options for NGD2VHDL or NGD2VER appropriate to your HDL simulator. The basic syntax without any options is as follows:

   ngd2ver *input_file*.**ngd** *output_file*.**v**

   ngd2vhdl *input_file*.**ngd** *output_file*.**vhd**

   Make sure that you specify an *output_file* name that does not overwrite an existing file.

   For more information on NGD2VER and NGD2VHDL, see the "NGD2VER" chapter and the "NGD2VHDL" chapter of the *Development System Reference Guide.*
7. Combine the behavioral .v file or .vhd file with the testbench file and simulate.

Post-Map Pre-Route Simulation (FPGAs Only)

A third type of simulation is post-map pre-route simulation. The M1 MAP tool trims redundant logic from a design and maps logic to the appropriate technology. Simulating after a design is mapped allows you to verify that the trimmed and mapped design behavior is still consistent with pre-synthesis RTL simulation and post-synthesis pre-route simulation.

Like post-synthesis pre-route simulation, instantiated FFS, OBUF, and OBUFTs do not make a difference, since all the logic in the design has already been transformed into SIMPRIMS.

1. Synthesize HDL code in Foundation Express to create an XNF file.
2. Open the Xilinx Design Manager by selecting **Start** → **Programs** → **Xilinx Foundation Series** → **Design Manager**. Do *not* open the Design Manager via the Foundation Project Manager.
3. Open the XNF file in the Xilinx Design Manager.

   Perform the following steps:

   a) After the Design Manager main window opens, create a new project directory by selecting **File** → **New Project**.

   b) In the New project dialog box, select the Browse button for the input design.

   c) Select XNF files from the Files of Type list box. Locate the XNF from the Look in list box. Click OK.

   d) Select **Design** → **New Version**. Enter a new version and click OK.

   e) Select **Design** → **New Revision**. Enter a new revision and click OK.

   f) If you have any instantiated NGO, EDIF, or XNF files in your design, make sure these files are located in the project directory along with the XNF file.
4. From the Design Manager, run the design through Translate and Map to create an .ncd file. Map produces an .ncd and an .ngm file from the .ngd file created by Translate. The .ncd file that is created is named map.ncd. The file is located in the revision directory. Copy this file into your project directory—the one that contains the XNF file and any instantiated NGO, EDIF, or XNF files.
5. Open a DOS window.
6. From the DOS command line, run NGDANNO command to create an .nga file. The basic syntax of the command is as follows:

   ngdanno -o *output_file*.**nga map.ncd map.ngm**

7. From the DOS command line, run NGD2VHDL or NGD2VER on the .nga file produced by NGDANNO. Use the appropriate command line options for NGD2VHDL or NGD2VER appropriate to your HDL simulator. The basic syntax without any options is as follows:

   `ngd2ver` *input_file*`.nga` *output_file*`.v`

   `ngd2vhdl` *input_file*`.nga` *output_file*`.vhd`

   Make sure that you specify an *output_file* name that does not overwrite an existing file.

   For more information on NGD2VER and NGD2VHDL, see the "NGD2VER" chapter and the "NGD2VHDL" chapter of the *Development System Reference Guide.*

8. Combine the behavioral .v file or .vhd file with the testbench file and simulate.

Post-PAR Simulation

The last type of simulation in the M1 flow is the traditional HDL timing simulation. In this simulation, all timing due to logic levels and routing is taken into account, along with behavior of the trimmed and mapped logic. Timing information for a design is back-annotated into an SDF file. Separation of the timing information into an SDF file has several advantages.

- Timing information separated into a SDF file allows the Xilinx design to be simulated in many third party HDL simulators of choice.
- Separation of timing into an SDF file allows for the increased speed of timing simulation. Consult your HDL simulator owner's manual for more information on SDF features.

1. Synthesize HDL code in Foundation Express to create an XNF file.
2. Open the Xilinx Design Manager by selecting **`Start`** → **`Programs`** → **`Xilinx Foundation Series`** → **`Design Manager`**. Do *not* open the Design Manager via the Foundation Project Manager.
3. Open the XNF file in the Xilinx Design Manager.

Perform the following steps:

a) After the Design Manager main window opens, create a new project directory by selecting **File** → **New Project**.

b) In the New project dialog box, select the Browse button for the input design.

c) Select XNF files from the Files of Type list box. Locate the XNF from the Look in list box. Click OK.

d) Select **Design** → **New Version**. Enter a new version and click OK.

e) Select **Design** → **New Revision**. Enter a new revision and click OK.

f) If you have any instantiated NGO, XNF, or EDIF files in your design, make sure these files are located in the project directory.

4. With the revision highlighted in the Design Manager main window, select **Tools** → **Flow Engine**.
5. Select **Setup** → **Options**. Select Produce Timing Simulation data.
6. To create VHDL or Verilog simulation data and back annotation, perform the following steps.

   a) Click Edit Template for the Implementation template.

   b) Within the Interface tab of the Implementation Options window, select VHDL or Verilog from the Format field.

   c) Select Correlate Simulation Data to Input Design.

   d) Click OK in the Implementation Options dialog box.

   e) Click OK in the Options dialog box.

7. From the Flow Engine window, select **Setup** → **Stop After**.
8. Select Timing from the Stop After list box and then click OK.
9. Run the design through the Design Manager Flow Engine by selecting **Flow** → **Run**. Depending on whether you created VHDL or Verilog simulation data, a *time_sim*.v or *time_sim*.vhd file is created in your project directory. In addition, a *time_sim*.sdf file with complete timing data is also produced.

10. Combine the behavioral .v file or .vhd file with the testbench file and simulate.

Simulating a Design With the Foundation Logic Simulator

If Foundation Express is used as the top-level design-entry tool, or if a Foundation Schematic is used as the top-level design with HDL macros in the schematic, the Foundation 1.4 gate-level simulator can be used for functional and timing simulation. Simulation using the gate simulator requires that a flat netlist be created before the simulator can be used. In the Foundation 1.4 flow, where Foundation Express is the top-level tool, there are six types of designs that can be simulated:

a) Pure HDL code

b) HDL code with instantiated LogiBLOX

c) HDL code with instantiated LogiCORE

d) HDL code with instantiated XNF

e) HDL code with instantiated EDIF from 1.4

f) HDL code which is a combination of one or more of b, c, d, and / or e.

Based on the six design types above and the four simulation flows available in Foundation 1.4, detailed instructions on simulation are described. Where possible, these instructions are GUI based. For the post-Translate and post-map flows, it may be necessary to run commands in a DOS shell.

Foundation 1.4 Project Structure Overview

In Foundation 1.4, when a project is defined in the Foundation Project Manager, you must specify the project name and the location of the project directory. Within the selected project location, a directory named after the project name is created along with a .pdf file (project directory file). The .pdf file contains project information regarding the design files and libraries in the project.

After defining the project, files can be added to the project by selecting **Document** → **Add**. Design files for a project can reside in any location. In the interest of organization, your design files should occupy the directory named after the project.

Foundation Express Project Structure Overview

Like the Project Manager, you must specify the project name and location. The project name becomes a directory inside the project location. Inside the directory named after the project, Foundation Express creates two items: a Workspace directory, which is used by Foundation Express during synthesis, and an .exp file, which contains project information. Design files for a Foundation Express project can reside in any location, although Xilinx recommends the project directory. Similarly, after synthesizing a design, you may place the resulting XNF file in any location, although Xilinx recommends that this file be placed within the Foundation project directory.

Creating the Foundation Express Project Structure

Follow these procedures to create a Foundation Express Project.

1. Start the Xilinx Foundation Project Manager.

Figure 4-1 Foundation Project Manager Icon

2. Select **File** → **New Project** to create a new project in the directory of your choice.

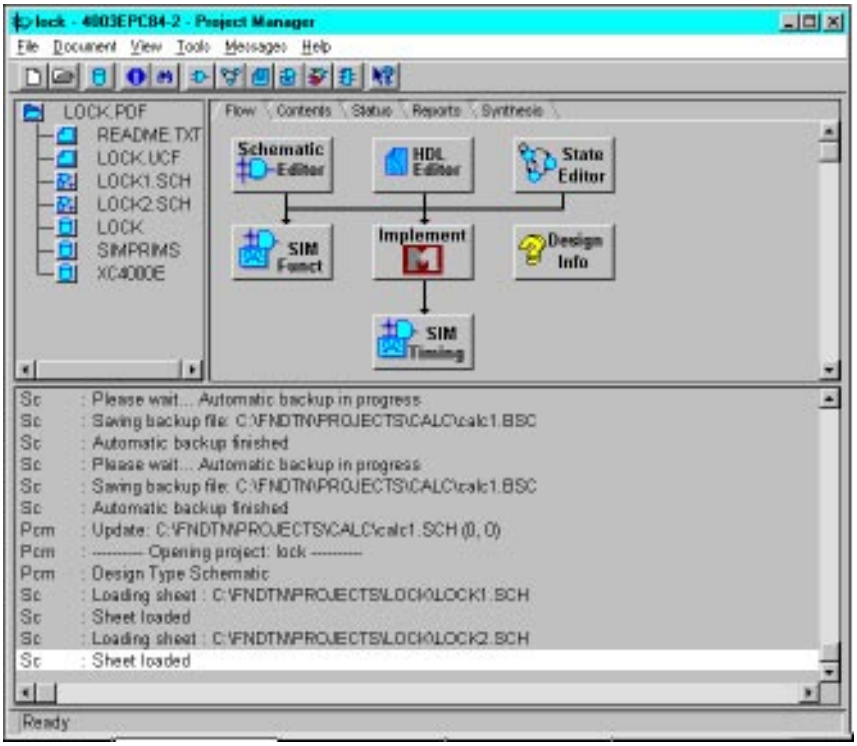


Figure 4-2 Defining a New Project in the Foundation Project Manager

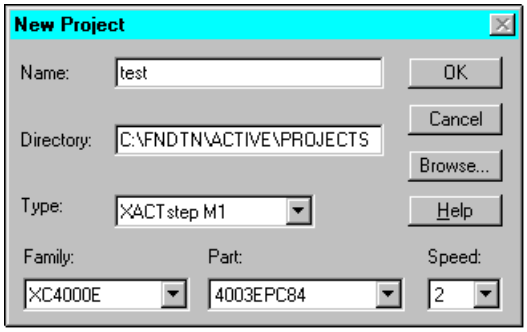


Figure 4-3 Specifying Project Name

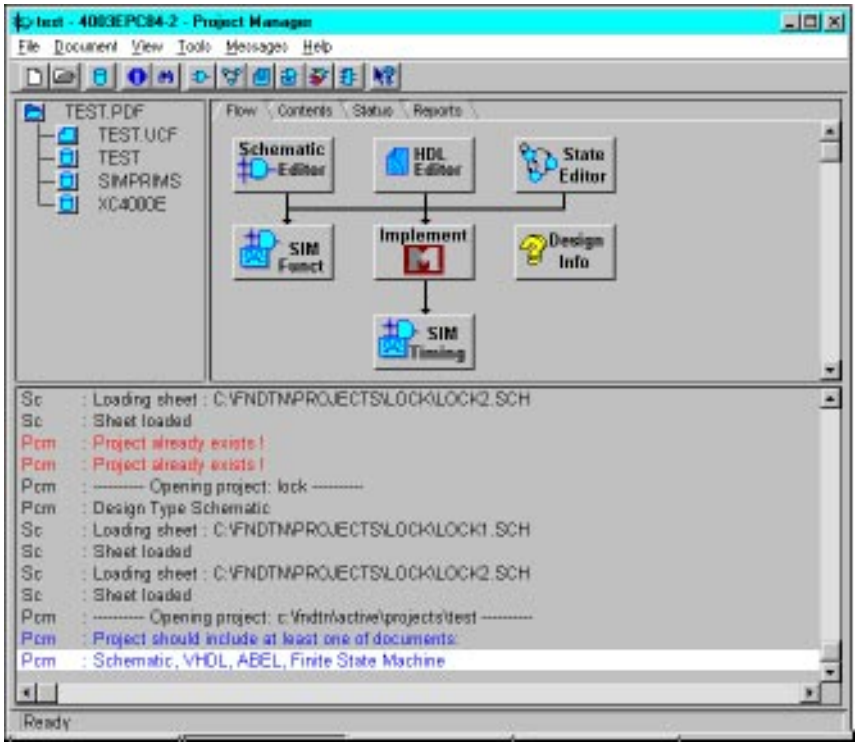


Figure 4-4 Foundation Project Manager Project Defined

3. Using the Explorer, create a subdirectory called “express” inside the directory where the Foundation project was created.
4. Using the Explorer, copy all HDL files into the express subdirectory. Create new HDL files for the Foundation Express project within this directory.
5. Start Foundation Express (**`Start`** → **`Programs`** → **`Xilinx Foundation Series`** → **`Foundation Express`**).
6. Create a Foundation Express project inside the express subdirectory (**`File`** → **`New`**).

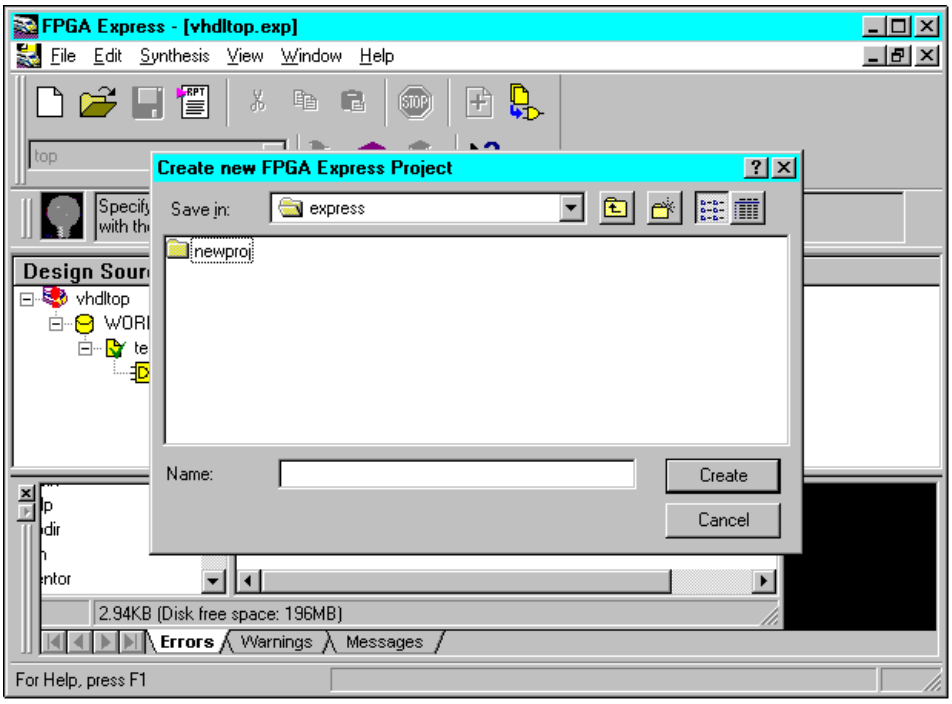


Figure 4-5 Making a New Foundation Express Project

7. Add files to the Foundation Express project from the express subdirectory.

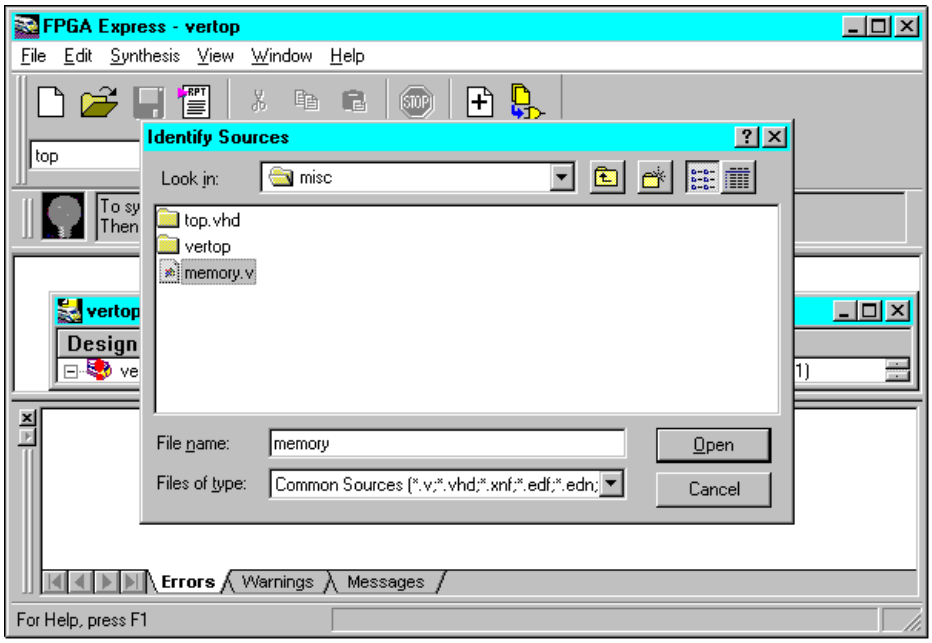


Figure 4-6 Adding HDL files to a Foundation Express Project

Using the Logic Simulator with Express Designs

If you are not familiar with the basic operations of the simulator, refer to the "Functional Simulation" chapter in the *Foundation Series User Guide*.

1. Create a Foundation Express project structure. See the "Foundation Express Project Structure Overview" section for details.
2. When generating the XNF file from Foundation Express, be sure that this file is saved into the Foundation project directory. In this example, this directory is c:\fndtn\active\projects\test.

When using the Foundation 1.4 Logic Simulator, the flow varies slightly depending on whether the top-level design file is the HDL file from Express or a Schematic from Foundation.

To simulate a Foundation Schematic with instantiated XNF modules from Foundation Express, perform the following steps:

1. Refer to the "Schematic Based Design and Foundation Express" chapter for instructions on importing the XNF files from Express into the Foundation Schematic.
2. To functionally simulate the design, simply invoke the Foundation Simulator by clicking the SIM Funct button in the Foundation Project Manager and simulate as with any other Foundation design.

To simulate a top-level HDL Express design using the Foundation 1.4 Logic Simulator:

1. Invoke the Xilinx M1 Design Manager from the Xilinx program group (**`Start`** → **`Programs`** → **`Xilinx Foundation Series`** → **`Design Manager`**).
2. Create a new project, using the XNF file from Express as the input file.
3. Create a new version and then a new revision, by selecting **`Design`** → **`New Version`**, then **`Design`** → **`New Revision`** from the Design Manager.

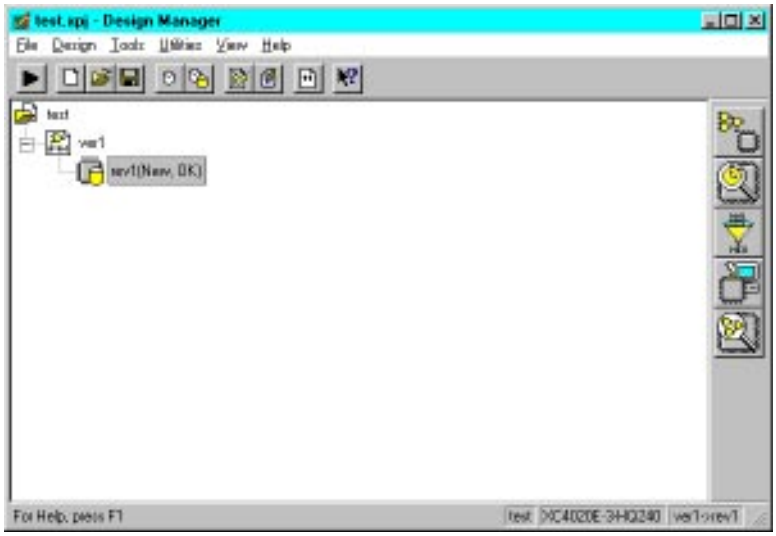


Figure 4-7 New Version and Revision Created

4. Run the Flow Engine and stop after Translate (**`Tools`** → **`Flow Engine`** and then select **`Setup`** → **`Stop After`**).

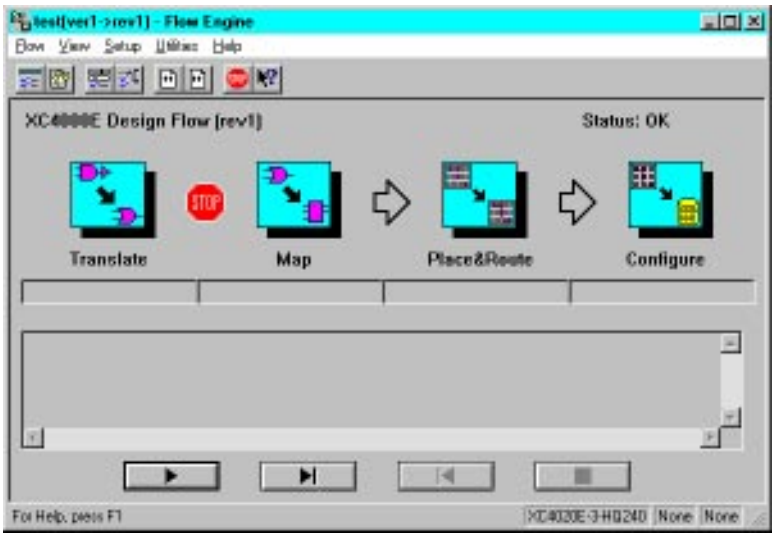


Figure 4-8 Flow Engine Set to Stop after Translate

5. Now, select **`Flow`** → **`Run`** in the Flow Engine.
6. When the Flow Engine is finished, return to the Foundation Project Manager and select **`Tools`** → **`Checkpoint Simulation`**.

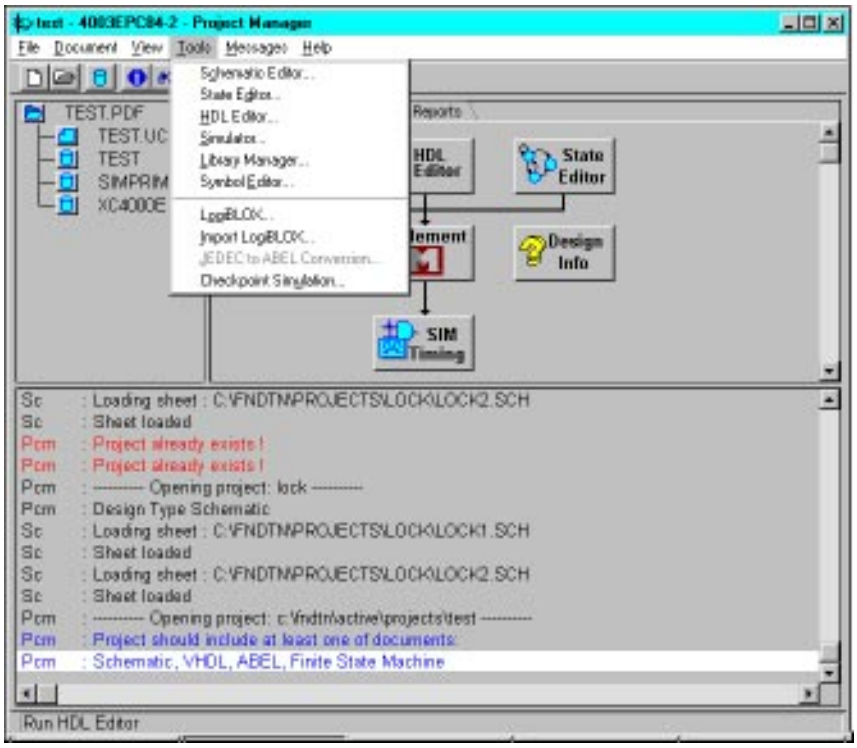


Figure 4-9 Checkpoint Simulation

An .ngd file displays in the Checkpoint Simulation window. The file name has the same name as the device family of the project. The extension is .ngd. In this example, since the project's device family is the 4000EX, the .ngd file created is XC4000EX.ngd.

7. Select the design listed in the Checkpoint Simulation window and click OK.

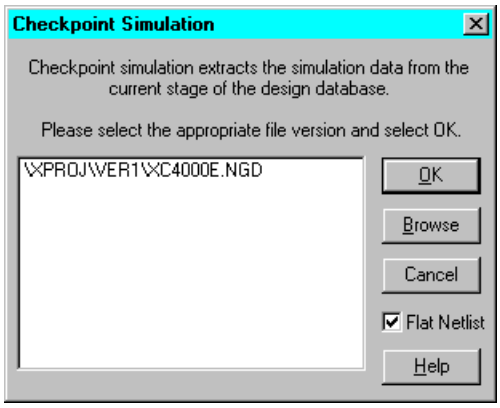


Figure 4-10 Select OK to Convert .ngd File to EDIF

8. After selecting OK in the Checkpoint Simulation window, the Project Manager indicates that NGD2EDIF is running. When NGD2EDIF is finished, the Foundation simulator automatically starts. Proceed with simulation by selecting the signals to stimulate for the **`Signals`** → **`Add Signals`**. The signals listed correspond to the top-level entity ports in VHDL or top-level module ports in Verilog. For more information on using the Foundation Simulator, please consult the Foundation Online Help.

Timing Simulation using the Logic Simulator

This section explains how to perform timing simulation for top-level schematic and HDL designs.

Top-level Schematics

1. Create a Foundation Express project. See the “Creating the Foundation Express Project Structure” section for more details.
2. Place and route the design. In the Flow Engine, make sure that the Produce Timing Simulation Data box is selected. If it is not, select **`Setup`** → **`Options`** and check this option.

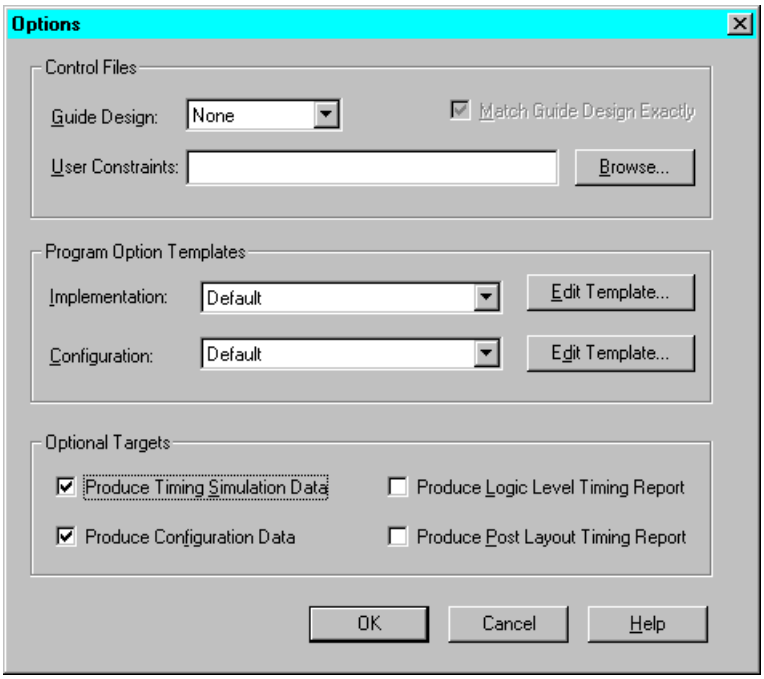


Figure 4-11 Options Dialog Box

3. After implementing the design in the Design Manager, return to the Foundation Project Manager and click the SIM Timing button to start timing simulation.

   Foundation automatically translates this back-annotated timing netlist to an EDIF file and loads the simulator.

Top-level HDL Designs

1. Make sure the Xilinx project is located within the Foundation project directory.
2. Place and route the design in the Design Manager. In the Flow Engine, make sure that the Produce Timing Simulation Data box is selected. If it is not, select **`Setup`** → **`Options`** and check this option.
3. From the Foundation Project manager, select **`Tools`** → **`Checkpoint Simulation`**. Select the *design_name*.nga file and click OK.

Chapter 5

Schematic Based Design and Foundation Express

This chapter describes the process for creating an HDL macro with Foundation Express and then creating a symbol for this macro for placement within a Foundation schematic. Foundation Express is the macro generator, and Foundation is the top-level tool.

Creating HDL Macros with Foundation Express

This procedure describes the process for creating an HDL macro with Foundation Express, and then creating a symbol for this macro for placement within a Foundation schematic. In this procedure, Foundation Express is the macro generator and Foundation is the top-level tool.

Creating the Foundation Project

The Foundation project should be created first. If the project has not yet been created, select **`File`** → **`New Project`** from the Foundation Project Manager and choose the appropriate family and desired project name. For more information on creating and working with Foundation projects, please refer to the *Foundation Series Quick Start Guide 1.4* and Online Help.

Compiling HDL Code in Foundation Express

Create a project in Foundation Express, if the project does not already exist, and synthesize the HDL design as described in the Foundation Express Documentation. When performing the Create Implementation step, be sure that the Do not insert I/O pads box is checked.

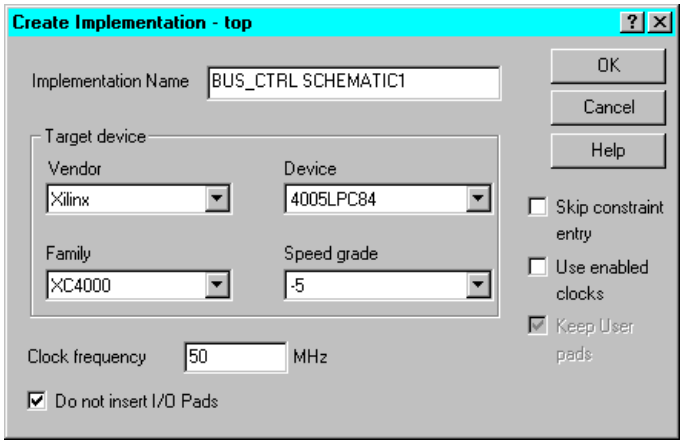


Figure 5-1 Do not insert I/O Pads

When performing the Export Netlist step in Foundation Express, browse to the Foundation project directory created above and save the netlist into this directory.

Importing the Netlist into Foundation Schematic

1. Open the Foundation project in the Foundation Project Manager.
2. To import the XNF netlist from Foundation Express into the Foundation Schematic, select **`Hierarchy`** → **`Create Macro Symbol from Netlist`** from the Schematic Editor.

Note: Foundation Express also creates an XSF file in addition to the XNF file. Foundation uses the XSF file to create bus pins. If you move the XNF file to the Foundation project, make sure you also move the XSF file.

This selection imports the netlist into the Foundation project and creates an associated symbol for placement on the schematic. The symbol is automatically added to the Foundation project library.

3. To bring up the SC Symbols list, click the SC Symbols icon on the vertical toolbar.

4. To place the symbol on the schematic, browse through the SC Symbols list of library components to find the module. The name of the module will be the same as the name of the imported XNF netlist.

Figure 5-2 SC Symbol Icon

5. Place the symbol on the schematic.

Chapter 6

Xilinx Customer Support Information

For registration, authorization codes, update information, warranty status, shipping, product issues, and technical support, call Monday through Friday, 8 a.m. to 5 p.m. Pacific time.

Registration, Authorization, and Customer Service

- United States and Canada (1-800-624-4782)
- Europe (Contact your local Distributor)
- Japan (81-33-297-9912)
- Southeast Asia / All Other Countries (852-2424-5200)
- Facsimile Transmission (1-408-559-0115)

Technical Support

Hotline Access and Hours

Location	Telephone	Electronic Mail	Hours
U.S. and Canada	1-800-255-7778	hotline@xilinx.com	Mon, Tues, Wed, Fri: 6:30 a.m. – 5:00 p.m Thurs: 6:30 a.m. – 4:00 p.m Pacific Standard Time
Japan	81-33-297-9163	jhotline@xilinx.com	Mon, Tues, Thurs, Fri: 9:00 a.m. – 5:00 p.m Wed: 9:00 a.m. – 4:00 p.m

Location	Telephone	Electronic Mail	Hours
France	33-1-3463-0100	frhelp@xlinx.com	Mon – Fri: 9:30 a.m. – 12:30 p.m 2:00 p.m. – 5:30 p.m
Germany	49-89-9915-4930	dlhelp@xilinx.com	Mon – Thurs: 8:00 a.m. – 12:00 p.m 1:00 p.m. – 5:00 p.m Fri: 8:00 a.m. – 12:00 p.m 1:00 p.m. – 3:00 p.m
United Kingdom	44-1-932-820821	ukhelp@xilinx.com	Mon – Thurs: 9:00 a.m. – 12:00 p.m 1:00 p.m. – 5:30 p.m Fri: 9:00 a.m. – 12:00 p.m 1:00 p.m. – 3:30 p.m

- Technical Support FAX (24 hours / 7 days) (1-408-879-4442)
- Internet E-mail Address (24 hours / 7 days) (hotline@xilinx.com)
- Xilinx Worldwide Web Site (http: / / www.xilinx.com)
- Xilinx Student Edition Users (For all technical support and further information, see http: / / www.xilinx.com / programs / univ.htm.)

Training

- Xilinx Training Administrator (1-408-879-5090)
- International customers, contact your local sales representative or distributor.

Index

I

L

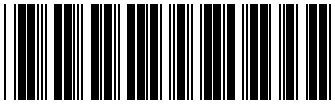
N

O

P

S

The Programmable Logic Company℠



0401721

Printed in U.S.A.