

# Accessing Xilinx Device Features Using Foundation VHDL

Rev. 2.1  
February 1997

This document discusses how to take advantage of Xilinx-specific device features through the Foundation VHDL tool. You should be running Xilinx Foundation Series software version 6.0.2.

## Contents

- |   |  |
|---|--|
| 1. I/O Pin Assignment                           | 8. Instantiating Xilinx Unified Library Components |
| 2. I/O Flip-Flops                               | 9. XBLOX   |
| 3. Bidirectional I/O                            | 10. RAM and ROM                                    |
| 4. Pull-ups, Pull-downs, and Wide-Edge Decoders | 11. Latches  |
| 5. Global Buffers                               | 12. Timespecs                                      |
| 6. Global Set/Reset and STARTUP                 | 13. Boundary Scan                                  |
| 7. Output Slew Rate                             |  |

## 1. I/O Pin Assignment

To lock down I/O signals to specific pins on the target device, use the 'PINNUM' attribute in the VHDL code as shown below. Either declare the pinnum attribute in the entity, or declare the Metamor library, in which the pinnum attribute is declared.

```
attribute pinnum of <port_name>: signal is "<pin_number>";
```

### >>Example of using PINNUM attribute:

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

library METAMOR;
--Package attributes contains declarations of the Metamor specific synthesis attributes.
use METAMOR.attributes.all;

entity flop is
  port (CLK, DIN, RESET: in std_logic;
        INBUS: in std_logic_vector (3 downto 0);
        DOUT: out std_logic;
        OUTBUS: out std_logic_vector (3 downto 0));

  attribute pinnum of DIN: signal is "p20"; -- lock DIN to p20
  attribute pinnum of INBUS: signal is "p16, p17, p18, p19";
                                     -- lock INBUS3 to p16 ... INBUS0 to p19.
end flop;

architecture LOCTEST of flop is
begin
  process (CLK, RESET)
  begin
    if RESET='1' then      --asynchronous RESET active High
      DOUT <= '0';
    end if;
  end process;
end architecture;
```

```

        elsif (CLK'event and CLK='1') then --CLK rising edge
            DOUT <= DIN;
        end if;
    end process;

    OUTBUS <= not INBUS;

end LOCTEST;

```

## **2. I/O Flip-Flops**

The X VHDL compiler will infer an input or output flip flop if all of the following criteria are met:

1. The design is compiled as a “chip”.
2. The input or output to the flip flop is also declared as a port in the entity.
3. The flip flop does not use the Preset or Clear pin, *unless* the Xilinx\_GSR attribute is used with that preset/clr signal. (See “Global Set/Reset,” Section 6, for more on this attribute).
4. For all families EXCEPT the XC4000E, the flip-flop does not use the CE pin.  
In an XC4000E design, a design which meets criteria 1-3, and uses the CE pin will infer an I/O flip-flop, as the XC4000E includes this architectural feature.

The following is an example where input flip-flops will be inferred by X VHDL. Note that since the flip-flops use the CE pin, only an XC4000E design will infer input flip-flops in this case.

### **>>Example of I/O flip-flop inference**

```

--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity iob4ke is
    port (IN1, IN2, CLK, EN: in std_logic; OUT1: out std_logic);
end iob4ke;

architecture IOFF_CE of iob4ke is
    signal A: std_logic;
    signal B: std_logic;
begin
    process (CLK, EN)
    begin
        if CLK'event and CLK='1' then --CLK rising edge
            if EN='1' then -- clock enable
                A <= in1;
            end if;
        end if;
    end process;

    process (CLK, EN)
    begin
        if CLK'event and CLK='1' then -- CLK rising edge
            if EN='1' then -- clock enable
                B <= in2;
            end if;
        end if;
    end process;

    OUT1 <= A and B;

end IOFF_CE;

```

### **3. Bidirectional I/O**

Bidirectional I/O signals should be described behaviorally in the VHDL code using an 'inout' port in the entity and with the output described as tri-state in the architecture. X VHDL will infer the appropriate types of I/O buffers.

The following example shows a behavioral description of a bidirectional I/O. Note that in this example the output is both tri-state and registered. This flip-flop will be pulled into the IOB as an output flip-flop.

#### **>> Example of behavioral description of bidirectional I/O**

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity bidirectional is
  port (DATA: inout std_logic;
        NOT_DATA: out std_logic;
        CLK, A, B: in std_logic;
        ENABLE: in std_logic);
end bidirectional;

architecture INVERT of bidirectional is
  signal IN1: std_logic;
  signal OUT1: std_logic;
begin
  IN1 <= A and B;

  process (CLK)
  begin
    if CLK'event and CLK='1' then --CLK rising edge
      OUT1 <= IN1;
    end if;
  end process;

  DATA <= (OUT1) when ENABLE='1' else 'Z'; -- describes tri-state buffer
  NOT_DATA <= not DATA;
end INVERT;
```

### **4. Using Pull-ups, Pull-downs and Wide-Edge Decoders**

The use of pull-ups, pull-downs and wide-edge decoders is not currently supported with Foundation VHDL. To take advantage of these architectural features, use a top-level schematic to place those components. The VHDL portion of the design can be made into a macro which can be placed on the top-level schematic. To create the macro, select 'Create Macro' from the Project menu in the HDL Editor.

### **5. Global Buffers**

Xilinx Devices feature global buffers which provide low skew and high drive for clock signals and other control signals with high fan-out. Consult the *XACTStep Libraries Guide* for more information on the various types of global buffers available with the various FPGA families.

There are 3 ways global buffers may be used through Foundation VHDL.

1. Automatic Insertion
2. Xilinx\_BUFG Attribute
3. Instantiation

### ***1. Automatic Insertion***

XVHDL automatically attaches a BUFG to any input port signal which directly drives a clock pin. Depending on which Xilinx device family is selected in the Foundation Project Manager, a limit is set as to the maximum number of BUFGs which can be automatically inserted into the synthesized XNF file, based on the architectural capabilities of the part.

When XVHDL inserts a BUFG on a clock net as described above, the BUFG takes the place of an IBUF. Therefore, a dedicated global buffer input pad on the device is used.

### ***2. Xilinx\_BUFG Attribute***

It is also possible for XVHDL to insert a BUFG on any specified input port signal, regardless of whether it is a clock signal or not. Use the 'Xilinx\_BUFG' attribute as shown below to access this feature. Again, this BUFG replaces the IBUF, and thus uses a dedicated global buffer pad. This attribute will not attach a BUFG to an internal signal. Only apply this attribute to top-level port signals.

```
attribute Xilinx_BUFG: boolean;  
attribute Xilinx_BUFG of <port_name>: signal is true;
```

### ***3. Instantiation***

Global buffers may also be instantiated. Use this method to drive a global buffer with an internally generated signal, or to control the specific type of global buffer used. If, for instance, in an XC4000 design, you wish to specify the use of a BUFPG rather than the generic BUFG, you may instantiate the BUFPG, as shown in the example below.

If you wish to use the dedicated input pad for the instantiated global buffer, you must use the 'inhibit\_buf' attribute on the input signal port to prevent XVHDL from inserting an IBUF at the input. This IBUF would prevent the global buffer from using the dedicated input pad. If you wish to use an input pad other than the dedicated global buffer input pad, do not use the 'inhibit\_buf' attribute. XVHDL will treat the signal like an ordinary input and insert an IBUF between the pad and the global buffer.

### **>>Example of using Xilinx\_BUFG attribute and instantiating global buffer**

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.  
library IEEE;  
use IEEE.std_logic_1164.all;  
  
entity dffs is  
  port (CLK1, CLK2, RESET, IN1, IN2, CE: in std_logic;  
        OUT1, OUT2, OUT3: out std_logic);  
  
  attribute xilinx_bufg: boolean;  
  attribute xilinx_bufg of CE: signal is true;  
    -- CE signal is driven through global buffer  
  attribute inhibit_buf: boolean;  
  attribute inhibit_buf of CLK2: signal is true;  
    -- prevents IBUF from being attached to this port,  
    -- as we are instantiating a BUFPG which will use the dedicated input pad.  
  
end dffs;  
  
architecture CLK_TEST of dffs is  
  signal BUFPG_CLK: std_logic;  
  component BUFPG  
    port (I: in std_logic; O: out std_logic);  
  end component;  
begin  
  process (CLK1, RESET, CE)  
  begin
```

```

    if RESET='1' then
        out1 <= '0';
    elsif (CLK1'event and CLK1='1') then
        if CE='1' then
            OUT1 <= IN1;
        end if;
    end if;
end process;

U1: BUFGP port map (I=>CLK2, O=>BUFGP_CLK);

process (BUFGP_CLK, RESET)
begin
    if RESET='1' then
        OUT2 <= '0';
    elsif (BUFGP_CLK'event and BUFGP_CLK='1') then
        OUT2 <= IN2;
    end if;
end process;

end CLK_TEST;

```

## **6. Global Set/Reset and STARTUP**

Xilinx FPGAs have a dedicated Global Reset net which, when asserted, will initialize all flip-flops in the device. (Set or Reset for the xc4000; Reset for the xc3000, xc5200)

The XC3x00/A devices have a dedicated RESET pin, which is connected to this global reset net.

The XC4000/A/D/H/E and XC5200 devices allow any signal, either external or internal, to serve as the global reset signal. The STARTUP symbol from the Xilinx Unified Libraries allows you to access this global reset net. The user Reset signal is connected to the GSR pin (for xc4000) or the GR pin (for xc5200) on the STARTUP symbol in order to access the global reset net. When this user Reset signal is asserted, all flip-flops in the device will be Set or Reset appropriately.

### ***XC4000/XC5200***

To use the dedicated global reset resource with Foundation VHDL, the STARTUP must be instantiated in the design, with the user Reset signal connected to the GSR pin on the STARTUP. See example below. It is not necessary to tie the Reset signal to every individual flip-flop in the design. Any signal that is tied to the Set/Reset pin of a flip-flop will use regular interconnect for the routing of the signal, and NOT the dedicated global resource network, causing unnecessary routing congestion. If you have written the VHDL code and included the Reset signal in individual flip-flop descriptions, this Reset net will be removed from the netlist when it is connected to the STARTUP block. This means that no regular routing resources will be taken up by the Reset signal—all flip-flops will now be reset with the dedicated Global Reset net. The STARTUP block is simulatable.

### ***XC3000***

With the XC3000 family, the global reset net is accessed through a dedicated pin on the device. For simulation purposes, the signal 'SimGlobalReset' in the simulation netlist serves as the global reset net. Toggle this to simulate the global reset.

If you have written the VHDL code and included the RESET signal in individual flip-flop descriptions, the 'Xilinx\_GSR' attribute may be used to strip out this signal from the resulting XNF netlist. This will enable use of the Global Reset signal instead, either through use of the STARTUP symbol, or just by using the dedicated pin on the XC3000 device. This will save valuable routing resources. The syntax for the Xilinx\_GSR attribute is as follows:

```
attribute Xilinx_GSR : boolean;
attribute Xilinx_GSR of <global_sig_name>: signal is true;
```

### >> Example of instantiating STARTUP and using Xilinx\_GSR attribute (XC4000 design)

**\*\*Note:** Assume that initially this design was coded such that RESET was the asynchronous Reset signal for many flip-flops, and it was later decided to use the Global Set/Reset rather than having RESET routed to many flip-flops through general routing resources. The STARTUP declares the signal RESET as the Global Set/Reset signal, and will also strip out the signal RESET from the rest of the XNF file.

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity use_gsr is
  port (RESET, IN1, CLK: in std_logic; OUT1 : out std_logic);
end use_gsr;

architecture TEST of use_gsr is
  component STARTUP
    port (GSR,GTS,CLK: in std_logic := '0';
          Q2,Q3,Q1Q4,DONEIN: out std_logic);
  end component;
begin
  U1: STARTUP port map (GSR=>RESET);
  process (CLK,RESET)
  begin
    if RESET='1' then
      OUT1 <= '0';
    elsif (CLK'event and CLK='1') then
      OUT1 <= IN1;
    end if;
  end process;
end TEST;
```

### ***XC4000 -- Using GSR, how do I specify some flops as Set, and some as Reset?***

The XC4000 family has the capability to either Set or Reset flip-flops upon assertion of the GSR signal. Unless otherwise specified, all flip-flops in the device will be asynchronously Reset, output low, when GSR is asserted. To specify asynchronous Set for various flip-flops, either:

1. Describe the 'Set' flip-flop behaviorally, then use Xilinx\_GSR attribute to remove the Preset signal.
2. Instantiate a flip-flop with asynchronous Set, such as FDP.

## **7. Specifying Output Slew Rate**

Slew rate may be set on a pad-by-pad basis for the XC4000, XC5200, XC9500 families. The default slew rate is 'SLOW'. To change the slew rate to 'FAST', use the following syntax:

```
attribute FAST: boolean;
attribute FAST of DOUT: signal is true;
```

### >>Example of specifying FAST slew rate:

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity flop is
  port (CLK, DIN, RESET: in std_logic; DOUT: out std_logic);
  attribute FAST: boolean;
  attribute FAST of DOUT: signal is true;  --Assigns FAST slew rate to DOUT pin
```

```

end flop;

architecture FASTSLEW of flop is
begin
  process (CLK, RESET)
  begin
    if RESET='1' then      --asynchronous RESET active High
      DOUT <= '0';
    elsif (CLK'event and CLK='1') then --CLK rising edge
      DOUT <= DIN;
    end if;
  end process;
end FASTSLEW;

```

## **8. Instantiating Xilinx Unified Library Components**

Many Xilinx Unified Libraries components may be instantiated in the VHDL code. Please refer to the README.TXT file in the <active>\VHDL\XLNX\_LIB directory for a Component Description Table of components which may be instantiated.

The syntax for declaring a component is as follows, and should be placed between the Architecture and Begin statements in the VHDL file:

```

component <component_name>
  port (<port_name_1> : <direction> <data_type>;
    ...
    <port_name_n> : <direction> <data_type>);
end component;

```

The syntax for instantiating the component is as follows, and should be placed after the Begin statement in the VHDL file:

```

<instance_name> : <component_name>
  port map (<port_name_1> => <signal_1>,
    ...
    <port_name_n> => <signal_n>);

```

If you instantiate any components marked with a “\*” in the Component Description Table, you must add the following line to the XDM.PRO file before compiling the design with the XACTStep Design Manager:

```
Options XNFMERGE -D <active>\VHDL\XLNX_LIB\<family>\
```

where <active> is the Foundation installation directory (typically C:\ACTIVE), and <family> is one of the following:

```

XC3000 : XC3x00/A/L designs
XC4000 : XC4000/A/H designs
XC4000E : XC4000E designs
XC5200 : XC5200 designs

```

## **9. XBLOX**

There are 2 ways in which XBLOX can be used in a design through Foundation VHDL:

1. Inference by XVHDL
2. Instantiation by the user.

### ***1. Inference by XVHDL***

XVHDL will automatically infer the following XBLOX components:

```
ADD_SUB
```

## COMPARE COUNTER ACCUM

The XBLOX synthesis compile option must be selected, and operands must be VHDL signals or variables for Xblox inference to occur. The following example shows code which will infer an XBLOX COUNTER component. This simple counter design can be found in the Active\vhdl\samples\prep7 directory.

### >>Example of inference of Xblox COUNTER

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;
-----
-- Benchmark circuit #7: 16-bit Counter (one instance)
--
entity prep7 is
    port(CLK,RST,LD,CE: boolean; D: integer range 0 to 65535;
          Q: buffer integer range 0 to 65535);
end prep7;

architecture BEHAVIOR of prep7 is
begin
    process(RST,CLK)
    begin
        if (RST) then                -- Async Reset
            Q <= 0;
        else
            if (CLK and CLK'event) then -- Clock (edge triggered)
                if CE or ld then        -- use register clk enable
                    if LD then          -- Sync Load
                        Q <= D;
                    else
                        Q <= Q + 1;
                    end if;
                end if;
            end if;
        end if;
    end process;
end BEHAVIOR;
```

### 2. *Instantiation of XBLOX*

XBLOX may also be instantiated in the VHDL code. Since the XBLOX library is a library of “parameterizable” macrocells, the same component may be declared once with ports of variable width, and then instantiated multiple times with various portwidths and parameters, if necessary. As well as declaring ports, generics may also be declared and mapped to the instantiated component to attach parameters to the component. XBLOX components are declared in the VHDL library XBLOX.VHD. Therefore, use this library in the VHDL code to be able to easily instantiate XBLOX components. Refer to the library file XBLOX.VHD in the ACTIVE\VHDL\VHDL\_LIB directory for more information regarding the component declarations and the available generics.

It is necessary to use the attribute ‘macrocell’ when declaring the XBLOX component so that the compiler understands that the functionality of the component will be described by another tool later in the implementation. The syntax for this attribute is as follows:

```
attribute macrocell of <xblox_component>: component is true;
```

This attribute is declared in the Metamor library.



## >> Example of instantiating the COUNTER component from the XBLOX library.

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

library METAMOR;
--Package attributes contains declarations of the Metamor specific synthesis attributes.
use METAMOR.attributes.all;

library XBLOX;
--This package contains component declarations for Xblox macrocells.
use XBLOX.macros.all;

entity count is
  port (DATA: in std_logic_vector (5 downto 0);
        CLK, LD: in std_logic;
        TC: out std_logic;
        QOUT: out std_logic_vector (5 downto 0));
end count;

architecture XBLOX of count is

  attribute MACROCELL of COUNTER: component is true;

begin
  U1: COUNTER generic map (STYLE => "JOHNSON") --defines Style as Johnson
    port map (D_IN=>DATA, LOAD=>LD, CLOCK=>CLK,
              TERM_CNT=>TC, Q_OUT=>QOUT);
end XBLOX;
```

## **10. Using RAM and ROM in XC4000 devices**

The XC4000 family has the capability to implement on-chip RAM and ROM efficiently using CLB function generators.

### **Using RAM**

There are 2 ways to implement RAM using Foundation VHDL.

1. Use Memory Generator to generate an XNF file for the memory, and then instantiate the XNF file.
2. Instantiate 16x1 and 32x1 RAM primitives.

\*Note: DO NOT describe RAM behaviorally in VHDL, as this results in combinatorial loops.

#### ***1. Using Memory Generator and instantiation***

- 1.) From the Foundation Project Manager, open 'Memory Generator' from the 'Applications' menu.
- 2.) Choose the type and size of RAM you want, give it a name, and push 'Generate'.
- 3.) This will create an XNF file for the RAM component.
- 4.) Now you must instantiate the XNF file in the VHDL code.

The names of the pins on the RAM follow the same naming convention as the RAM components in the *Xilinx Libraries Guide*, so you really only need to be concerned with the number of inputs, addresses, and outputs, which is dependant on the size of the RAM, to be sure that the port names on the RAM component match those in the XNF file. If there is any doubt as to the names of the ports, the names of the pins on the RAM component can be found in the XNF file which was produced by the Memory Generator.

The following is an example of this method. Note also the use of the 'macrocell' attribute. This attribute allows for the use of vector pins as opposed to single-bit pins. If this attribute is not used, the pin expansion

of the vectors will be incompatible, and the resulting XNF netlist will not compile correctly. Declare the Metamor library when using this attribute. The second example of instantiating RAM uses single-bit pins, and thus the 'macrocell' attribute is not necessary.

### >>Example of Instantiating XNF file Created with Memory Generator

```
--This is an example of a 16x2 Dual-Port RAM
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

library METAMOR;
--Package attributes contains declarations of the Metamor specific synthesis attributes.
use METAMOR.attributes.all;

entity dual_port is
    port(ADD, DPRADD: in std_logic_vector (3 downto 0);
          WR_EN, WR_CLK, DATA0, DATA1: in std_logic;
          SPOUT, DPOUT: out std_logic_vector (1 downto 0));
end dual_port;

architecture FROM_MEMGEN of dual_port is
    component MEMGEN_M --memgen_m.xnf is the XNF file created by Memory Generator
        port(A,DPR: in std_logic_vector (3 downto 0);
              WE,WCLK,D0,D1: in std_logic;
              SPO,DPO: out std_logic_vector (1 downto 0));
    end component;
    attribute macrocell of memgen_m: component is true;
begin
    U1: MEMGEN_M port map (A=>ADD, DPR=>DPRADD,
                           WE=>WR_EN, WCLK=>WR_CLK, D0=>DATA0, D1=>DATA1,
                           SPO=>SPOUT, DPO=>DPOUT);
end FROM_MEMGEN;
```

## 2. *Instantiating RAM primitive.*

RAM 16x1 and 32x1 primitives from the XC4000 Unified Library may be instantiated directly into a VHDL file as shown below.

### >>Example of instantiating a RAM16X1D primitive.

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity dp_ram is
    port (WR, DATA, CLK: in std_logic;
          ADDR0, ADDR1, ADDR2, ADDR3: in std_logic;
          DPRA0, DPRA1, DPRA2, DPRA3: in std_logic;
          SPOUT, DPOUT: out std_logic);
end dp_ram;

architecture INST of dp_ram is
    component RAM16X1D
        port (D,WE,WCLK,A3,A2,A1,A0,DPRA3,DPRA2,DPRA1,DPRA0: in std_logic;
              SPO,DPO: out std_logic);
    end component ;
begin
    U1: RAM16X1D
        port map (
```

```

D=>DATA, WE=>WR, WCLK=>CLK,
A3=>ADDR3, A2=>ADDR2, A1=>ADDR1, A0=>ADDR0,
DPRA3=>DPRA3, DPRA2=>DPRA2, DPRA1=>DPRA1, DPRA0=>DPRA0,
SPO=>SPOUT, DPO=>DPOUT);
end INST;

```

\*Note: When compiling the design with the XACTStep Design Manager, you must copy the appropriate XNF file (in this case RAM16X1D.XNF) from the <Foundation\_directory>\vhdl\xlntx\_lib\<family>\ directory into your project directory, *or* add the following line to the XDM.PRO file found in your project directory: Options XNFMERGE -D <active>\VHDL\XLNX\_LIB\<family>\ where <active> is the location of the Foundation install (typically C:\ACTIVE) and <family> is either XC4000 or XC4000E.

## Using ROM

There are 3 ways to implement ROM using Foundation VHDL.

1. Use Memory Generator to generate an XNF file for the memory, and then instantiate the XNF file.
2. Instantiate ROM primitives.
3. Describe ROM behaviorally.

### *Option 1 follows the same procedure as described above for RAM.*

In this case be sure that there is a .MEM file in your project directory describing the contents of the ROM. More detail on the format of the .MEM file can be found in the 'MEMGEN' chapter of the *Xilinx Development Systems Reference Guide, Vol. 1*.

### **2. Instantiating ROM primitives.**

ROM 16x1 or 32x1 primitives may be instantiated using the following method. An INIT attribute must be added to the instantiated ROM component to describe the initial contents of the ROM. The INIT attribute must be set to a HEX value: 4 digits for a 16x1 ROM, and 8 digits for a 32x1 ROM. Note that the name of the component is just ROM, and the 'SCHNM' attribute determines whether it is a 16x1 ROM or a 32x1 ROM. As well, a 'DEF' attribute must be attached to the instantiated component, with the value 'ROM'. See example below.

### **>>Example of instantiating a ROM primitive**

```

--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

library METAMOR;
--Package attributes contains declarations of the Metamor specific synthesis attributes.
use METAMOR.attributes.all;

entity rom_inst is
  port (ADDR0, ADDR1, ADDR2, ADDR3: in std_logic;
        OUTPUT: out std_logic);
end rom_inst;

architecture INST of rom_inst is
  component ROM    --NOTE: component is declared as just ROM.
    port (A0, A1, A2, A3: in std_logic; O: out std_logic);
  end component;
  attribute INIT: string;
  attribute DEF: string;
  attribute SCHNM: string;
  attribute INIT of U1: label is "1111";
  attribute DEF of U1: label is "ROM";
  attribute SCHNM of U1: label is "ROM16X1"; --Determines whether component is ROM16x1 or ROM32x1
begin

```

```

    U1: ROM port map (A0=>ADDR0, A1=>ADDR1, A2=>ADDR2, A3=>ADDR3, O=>OUTPUT);
end inst;

```

\*Note: When compiling the design with the XACTStep Design Manager, you must copy the appropriate XNF file (in this case ROM16X1.XNF) from the <Foundation\_directory>\vhdl\xlxx\_lib\<family>\ directory into your project directory, *or* add the following line to the XDM.PRO file found in your project directory: Options XNFMERGE -D <active>\VHDL\XLNX\_LIB\<family>\ where <active> is the location of the Foundation install (typically C:\ACTIVE) and <family> is either XC4000 or XC4000E.

### 3. Describing a ROM behaviorally

#### >>Example of how to describe a 16x4 ROM behaviorally in VHDL.

```

--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity rom16x4_4k is
    port (ADDR: in integer range 0 to 15;
          DATA: out std_logic_vector (3 downto 0));
end rom16x4_4k;

architecture BEHAV of rom16x4_4k is
    subtype ROM_WORD is std_logic_vector (3 downto 0);
    type ROM_TABLE is array (0 to 15) of ROM_WORD;
    constant ROM: ROM_TABLE := ROM_TABLE'(
        ROM_WORD("0000"),
        ROM_WORD("0001"),
        ROM_WORD("0010"),
        ROM_WORD("0100"),
        ROM_WORD("1000"),
        ROM_WORD("1000"),
        ROM_WORD("1100"),
        ROM_WORD("1010"),
        ROM_WORD("1001"),
        ROM_WORD("1001"),
        ROM_WORD("1010"),
        ROM_WORD("1100"),
        ROM_WORD("1001"),
        ROM_WORD("1001"),
        ROM_WORD("1101"),
        ROM_WORD("1111"));
begin
    DATA <= ROM(ADDR);
end BEHAV;

```

## 11. Using Latches

### ***XC3000***

XC3000 devices do not have latches available in the CLBs. Attempts at implementing latches should be avoided, as the latch will be implemented with gates, and combinatorial loops will result. D flip-flops should be used instead.

### ***XC4000***

XC4000 devices, like XC3000 devices, do not have latches available in the CLBs, so latches would be implemented using gates. However, unlike the XC3000, the XC4000 also has the option to use RAM to

implement latches without combinatorial loops. XVHDL will actually infer RAM if a latch is described in the VHDL code.

### ***XC5200***

XC5200 devices do have latches available in the CLBs. Latches may either be inferred or instantiated with XC5200 designs.

If Improvex appears to be taking too long to compile, this can be an indication that XVHDL is inferring latches. If you believe that XVHDL is inferring latches where flip-flops were anticipated, check the XVHDL synthesis report to see if it reports any inferred latches or combinatorial feedback. If so, check the code to be sure that flip-flops are coded properly to infer flip-flops and not latches.

The following is an example showing how the same VHDL code which describes a latch will be implemented in different ways depending on the target architecture.

- \* XC3000 designs: XVHDL will implement the latch using combinatorial gates and feedback loops.
- \* XC4000 designs: XVHDL will implement the latch using RAM.
- \* XC5200 designs: XVHDL will implement the latch using the Latch primitive from the XC5200 library.

### **>>Example of a VHDL file describing a Transparent Data Latch**

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (GATE, A, B: in std_logic; Q: out std_logic);
end d_latch;

architecture BEHAV of d_latch is
  signal DATA: std_logic;
begin
  DATA <= A and B;
  process (GATE, DATA)
  begin
    if (GATE = '1') then
      Q <= DATA;
    end if;
  end process;
end BEHAV;
```

## **12. Using Timespecs**

XACT-Performance, part of the XACTStep implementation software, allows users to set timing requirements on various paths in the design. By attaching 'TNM' attributes to various instances in the design, and then setting point-to-point timespecs, timing-critical paths can be controlled. The process for using timespecs with a Foundation VHDL design is as follows:

1. Attach 'TNM' attributes to PADs, Flip-Flops, RAMs, Latches in the VHDL code.
2. Write a constraint file (.CST) which contains the Timespecs. This .CST file will be read into the Design Manager during the implementation.

Refer to the *Xilinx Development Systems Reference Guide Vol. 1, Chapter 4*, for more information about using Timespecs and XACT-Performance.

### ***1. Attaching a 'TNM' attribute from within the VHDL code:***

### Inferred Components

To attach a 'TNM' attribute to an inferred PAD, Flip-Flop or Latch, use the following syntax:

```
attribute TNM: string;  
attribute TNM of <signal_name>: signal is "<tnm_id>";
```

<Signal\_name> is the Q output of the flip-flop or latch, or the port name for the desired pad.

<Tnm\_id> is the TNM identifier name which you assign to the instance.

### Instantiated Components

To attach a 'TNM' attribute to an instantiated flip-flop, RAM or latch, use the following syntax:

```
attribute TNM: string;  
attribute TNM of <instance_name>: label is "<tnm_id>";
```

<Instance\_name> is the instance name assigned to the instantiated component in the VHDL code.

<Tnm\_id> is the TNM identifier name which you assign to the instance.

If a signal is declared as a top-level entity port, and is also the output of a flip-flop or latch, a TNM attribute attached to that signal will be passed to the OPAD, and not to the flip-flop. In order to attach a TNM attribute to a flip-flop whose output is also a top-level output port, a dummy signal must be created. See below for an example of this usage.

### >>Example of Attaching TNM Attributes to Inferred Flip-Flops and PADs.

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.  
library IEEE;  
use IEEE.std_logic_1164.all;  
  
library METAMOR;  
--Package attributes contains declarations of the Metamor specific synthesis attributes.  
use METAMOR.attributes.all;  
  
entity flops is  
  port (DIN1, CLK, RESET: in std_logic; DOUT1: out std_logic);  
end flops;  
  
architecture USE_TSPEC of flops is  
  signal X : std_logic;  
  signal Y : std_logic;  
  signal DUMMY: std_logic;  
  
  attribute TNM: string;  
  attribute TNM of DOUT1: signal is "outpad"; --TNM of 'outpad' on Output Pad  
  attribute TNM of DIN1: signal is "inpad"; --TNM of 'inpad' on Input Pad  
  attribute TNM of X: signal is "flopX"; --TNM of 'flopX' on 1st FF  
  attribute TNM of Y: signal is "flopY"; --TNM of 'flopY' on 2nd FF  
  attribute TNM of DUMMY: signal is "flopout"; --TNM of 'flopout' on 3rd FF  
begin  
  process (CLK, RESET)  
  begin  
    if RESET='1' then  
      X <= '0';  
    elsif (CLK'event and CLK='1') then  
      X <= DIN1;  
    end if;  
  end process;  
  
  process (CLK, RESET)  
  begin
```

```

    if RESET='1' then
        Y <= '0';
    elsif (CLK'event and CLK='1') then
        Y <= X;
    end if;
end process;

process (CLK, RESET)
begin
    if RESET='1' then
        DUMMY <= '0';
    elsif (CLK'event and CLK='1') then
        DUMMY <= Y;
    end if;
end process;

DOUT1 <= DUMMY; --Pass the output of the last FF to the OPAD.
                --This dummy signal is created to be able to put a TNM on the
                --last FF whose output is also the top-level port.
end USE_TSPEC;

```

## >>Example of Attaching TNM Attribute to Instantiated RAM

```

--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

library METAMOR;
use METAMOR.attributes.all;

entity DP_RAM is
    port (WR, DATA, CLK, RST: in STD_LOGIC; ADDR0, ADDR1, ADDR2, ADDR3: in std_logic;
          DPRA0, DPRA1, DPRA2, DPRA3: in std_logic;
          DPOUT, SPOUT, DOUT: out std_logic);
end dp_ram;

architecture INST of dp_ram is
    component RAM16X1D
        port (D,WE,WCLK,A3,A2,A1,A0,DPRA3,DPRA2,DPRA1,DPRA0: in std_logic;
              SPO,DPO: out std_logic);
    end component ;
    attribute TNM: string;
    attribute TNM of U1: label is "ramtnm"; -- Attach TNM of 'ramtnm' to the RAM instance U1.
begin
    U1: RAM16X1D
        port map (
            D=>DATA, WE=>WR, WCLK=>CLK,
            A3=>ADDR3, A2=>ADDR2, A1=>ADDR1, A0=>ADDR0,
            DPRA3=>DPRA3, DPRA2=>DPRA2, DPRA1=>DPRA1, DPRA0=>DPRA0,
            SPO=>SPOUT, DPO=>DPOUT);
end INST;

```

## 2. Writing a Constraint File with Timespecs to be used by XACTStep Design Manager

After attaching TNMs to the desired components in the VHDL file, you must now write a constraint file (.cst) which will be read by the XACTStep Design Manager during the Implementation phase. Below is an example of the correct syntax to use when writing the constraint file. This constraint file cooresponds to the first example code shown above, containing the three inferred flip-flops.

```

timespec="ts01=from:inpad:to:flopx=20ns";
timespec="ts02=from:flopx:to:flopy=8ns";
timespec="ts04=from:flopout:to:outpad=8ns";

```

```
timespec="ts05=from:flopy::to:flopout=10ns";
```

### **13. Using Boundary Scan**

XC4000 and XC5200 devices feature boundary scan logic which supports the IEEE Standard 1149.1. Refer to the *Development System User Guide* for more information on the Boundary Scan capabilities of the devices, and to the *Libraries Guide* for more information on the BSCAN symbol.

#### ***XC4000***

To be able to access the Boundary Scan logic after configuration, the BSCAN and TDI, TMS, TCK, and TDO Pads must be instantiated in the design. (To access boundary scan logic before configuration, nothing special need be added to the design.) These special I/O Pads connect directly to the BSCAN component pins. Do not declare the TDI, TMS, TCK and TDO signals as ports in the entity declaration, as the compiler would then insert regular I/O pads in addition to these special pads. See example below.

#### **>> Example of instantiating BSCAN in XC4000 design.**

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity jtag is
  port (A: in std_logic; B: out std_logic);
end jtag;

architecture TEST of jtag is
  signal TDI_P: std_logic;
  signal TMS_P: std_logic;
  signal TCK_P: std_logic;
  signal TDO_P: std_logic;

  component BSCAN
    port (TDI, TMS, TCK: in std_logic; TDO: out std_logic);
  end component;

  component TDI
    port (I: out std_logic);
  end component;

  component TCK
    port (I: out std_logic);
  end component;

  component TMS
    port (I: out std_logic);
  end component;

  component TDO
    port (O: in std_logic);
  end component;

begin
  U1: BSCAN port map (TDI=>TDI_P, TMS=>TMS_P, TCK=>TCK_P, TDO=>TDO_P);
  U2: TDI port map (I=>TDI_P);
  U3: TMS port map (I=>TMS_P);
  U4: TCK port map (I=>TCK_P);
  U5: TDO port map (O=>TDO_P);

  B <= not A;
end TEST;
```



### ***XC5200***

To be able to access the Boundary Scan logic after configuration, the BSCAN component, special I/O Pads, *as well as* IBUFs and OBUF(s) must be instantiated. Unlike with the XC4000, the TDI, TMS, TCK, and TDO pads *do not* connect directly to the BSCAN component pins. They must go through an IBUF or an OBUF, and thus these must also be instantiated. See example below.

#### **>> Example of instantiating BSCAN in XC5200 design**

```
--The IEEE standard 1164 package, declares std_logic, rising_edge(), etc.
library IEEE;
use IEEE.std_logic_1164.all;

entity jtag is
  port (A: in std_logic; B: out std_logic);
end jtag;

architecture TEST of jtag is
  signal TDI_P: std_logic;
  signal TMS_P: std_logic;
  signal TCK_P: std_logic;
  signal TDO_P: std_logic;
  signal TDI_BUF: std_logic;
  signal TMS_BUF: std_logic;
  signal TCK_BUF: std_logic;
  signal TDO_BUF: std_logic;

  component BSCAN
    port (TDI, TMS, TCK: in std_logic; TDO: out std_logic);
  end component;

  component TDI
    port (I: out std_logic);
  end component;

  component TCK
    port (I: out std_logic);
  end component;

  component TMS
    port (I: out std_logic);
  end component;

  component TDO
    port (O: in std_logic);
  end component;

  component IBUF
    port (I: in std_logic; O: out std_logic);
  end component;

  component OBUF
    port (I: in std_logic; O: out std_logic);
  end component;

begin
  U1: BSCAN port map (TDI=>TDI_BUF, TMS=>TMS_BUF, TCK=>TCK_BUF, TDO=>TDO_BUF);
  U2: TDI port map (I=>TDI_P);
  U3: TMS port map (I=>TMS_P);
  U4: TCK port map (I=>TCK_P);
  U5: TDO port map (O=>TDO_P);
  U6: IBUF port map (I=>TDI_P, O=>TDI_BUF);
```

```
U7: IBUF port map (I=>TMS_P, O=>TMS_BUF);  
U8: IBUF port map (I=>TCK_P, O=>TCK_BUF);  
U9: OBUF port map (I=>TDO_BUF, O=>TDO_P);
```

```
B <= not A;  
end TEST;
```