Application Note
January 2002

ORCA® Series 3 FPGAs Programmable I/O Cell (PIC): Logic, Clocking, Routing, and External Device Interface

Abstract

This application note describes the features and advantages of the *ORCA* Series 3 FPGA programmable I/O cell (PIC). The Series 3 PIC architecture is presented in detail. Methods for efficiently implementing designs using the PIC architecture's resources are presented, using both synthesis tool inferencing and source code instantiation. Several novel applications which utilize some of the PIC's special features are presented, as well as methods for implementing them.

Introduction

Modern digital electronic systems are being asked to perform increasingly difficult tasks at greater speed, with lower power, and in less space. Today's systems frequently utilize one or several high-speed microprocessors, high-speed memory, and fast bus architectures in order to squeeze the maximum system speed out of the smallest area. These devices are also migrating to lower voltages due to ever decreasing device geometry and power consumption limitations. Also, due to shrinking design times and system flexibility needs, more designers are turning to programmable logic solutions. The latest crop of PLDs and FPGAs have huge numbers of internal gates, registers, and routing resources, all of which can operate at extremely high core speeds. To take full advantage of these internal capabilities in a high-speed digital system, the design of the device's interface to the external system (I/O) must take all of the internal and external requirements into account and allow for very high-speed data transfers into and out of the device.

The *ORCA* Series 3 FPGA family incorporates a new programmable I/O cell (PIC) which takes all of these requirements into account. It has been designed to allow for very high system speeds. It can interface to external devices with several different voltage levels. It has selectable power management and system noise management features, and several special built-in logic functions to directly interface to external buses.

PIC Features and Benefits

The Series 3 PIC architecture has numerous features which can be very beneficial to the designer. Generally speaking, the Series 3 PIC allows for a higher frequency system interface than many other FPGAs due to its register/latch and zero-hold capabilities. It allows for greater flexibility in the electrical characteristics of the I/O signals. It can save on logic, registers/latches, and routing resources that would otherwise be implemented in internal logic. It can save power by allowing a lower-voltage core to interface to a higher-voltage system (such as a 3.3 V core in a 5 V system). It can also save power and reduce noise/ground bounce by using different output driver modes. Due to routing resources, it could save the designer time and effort by allowing pins to be locked much earlier in the design cycle. Finally, the PIC can save cost by potentially allowing the design to fit into a smaller array than would otherwise have been possible. Some of the PIC's specific features and their corresponding benefits can be seen in the following tables.

Table of Contents

Contents **Page**

List of Tables

List of Figures

PIC Features and Benefits (continued)

Table 1. Input Features and Benefits

Input Features	Benefits
TTL or CMOS compatible input levels	Flexibility to interface to different external devices.
5 V tolerant OR3Txx/OR3LxxxB	Flexibility to operate in mixed voltage environments.
Programmable delay	Ability to create fast zero-hold inputs.
100 kΩ pull-up/50 kΩ pull-down	Inputs can be defaulted without external resistors.
Input registered modes (latch, FF, LFF, direct in)	Fast registered/latched inputs and/or zero-hold inputs without using PLC resources.
Normal/inverted clock	Fast, flexible input clocking without using PLC resources.
Two inputs (IN1, IN2) per I/O pin	Flexibility to deMUX inputs, bring both signals into the device.
Clock input per I/O pin	Allows any input to drive an internal tree routing structure for use as clocks on other high fan-out signal.

Table 2. Output Features and Benefits

Output Features	Benefits
Two outputs (OUT1, OUT2) from array per I/O pin	Flexibility to MUX outputs.
Selectable drive current	Fast outputs or low-power outputs.
Normal or fast open drain	Fast-shared-interrupt outputs.
Fast, slewlim, sinklim buffers	Fast outputs or low-power, low EMI outputs.
100 kΩ pull-up/50 kΩ pull-down	Outputs can be defaulted without external resistors.
Output from PIO FF or general routing	Optional fast registered outputs without using PLC FFs.
Registered 3-state signal	Fast-output enable without using PLC resources.
ECLK or SCLK FF resources	Fast, flexible output clocking without using PLC.
Normal/inverted clock resources	Fast, flexible output clocking without using PLC.
Output logic	Gated clock outputs, pulses, decodes, etc.
Output MUXes	Flexibility to MUX outputs.

Table 3. Control Signal Features and Benefits

Control Signal Features	Benefits
CE—active-high or -low, or always on	Allows for fast, flexible control of registered I/O without using PLC.
LSR—active-high or -low	Allows for fast, flexible set/reset of registered I/O without using PLC.
LSR—asynchronous or synchronous	Allows for fast, flexible set/reset of registered I/O without using PLC.
LSR—ce_over_lsr, lsr_over_ce (sync)	Allows for fast, flexible set/reset of registered I/O without using PLC.
GSR—enable/disable	Allows for fast, flexible set/reset of registered I/O without using PLC.

Table 4. Grouping and Routing Features

Grouping and Routing Features	Benefits
Four PIO per PIC, two PICs per pair	Easier to create nibble and byte-wide oriented I/O buses.
Routing structure similar to PLC	Flexible routing allows pin locking of device prior to place and route.

PIC Architecture Description

Overview

The *ORCA* Series 3 PICs are located around the perimeter of the FPGA device (top, bottom, left, and right edges). An *ORCA* Series 3 PIC interfaces to four bond pads and contains the necessary routing resources to provide an interface between I/O pads and the PLCs. Each PIC is composed of four programmable I/Os (PIOs) and significant routing resources to adjacent PICs and PLCs, as well as to the routing resources of the entire device. PICs are grouped in pairs for purposes of routing. This allows the PIC's architecture to handle nibble and byte wide buses very efficiently. Each PIO contains input buffers, output buffers (3-statable), routing resources, latches/FFs, and logic and can be configured as an input, output, or bidirectional I/O. Each PIO can be configured for TTL or CMOS level input mode. Each PIO can automatically accept 3.3 V or 5 V referenced levels. Each PIO output can be adjusted for speed and current drive capability.

Programmable Input/Output (PIO)

Figure 1 shows a Series 3 PIO. Each PIO within a PIC can be logically divided into two blocks: input logic and output logic. These two blocks share some common resources, namely the I/O pad of the device, pull-up/pull-down resistors, a system clock, an Express clock, controls for clock enable, local set/reset, and global set/reset. If the pad is designed as bidirectional I/O, then both blocks (input and output) of the PIO will be used. Also, note that if the I/O pad is used only as an output, the input block remains active and its resources, such as pull-ups and pull-downs, can be used.

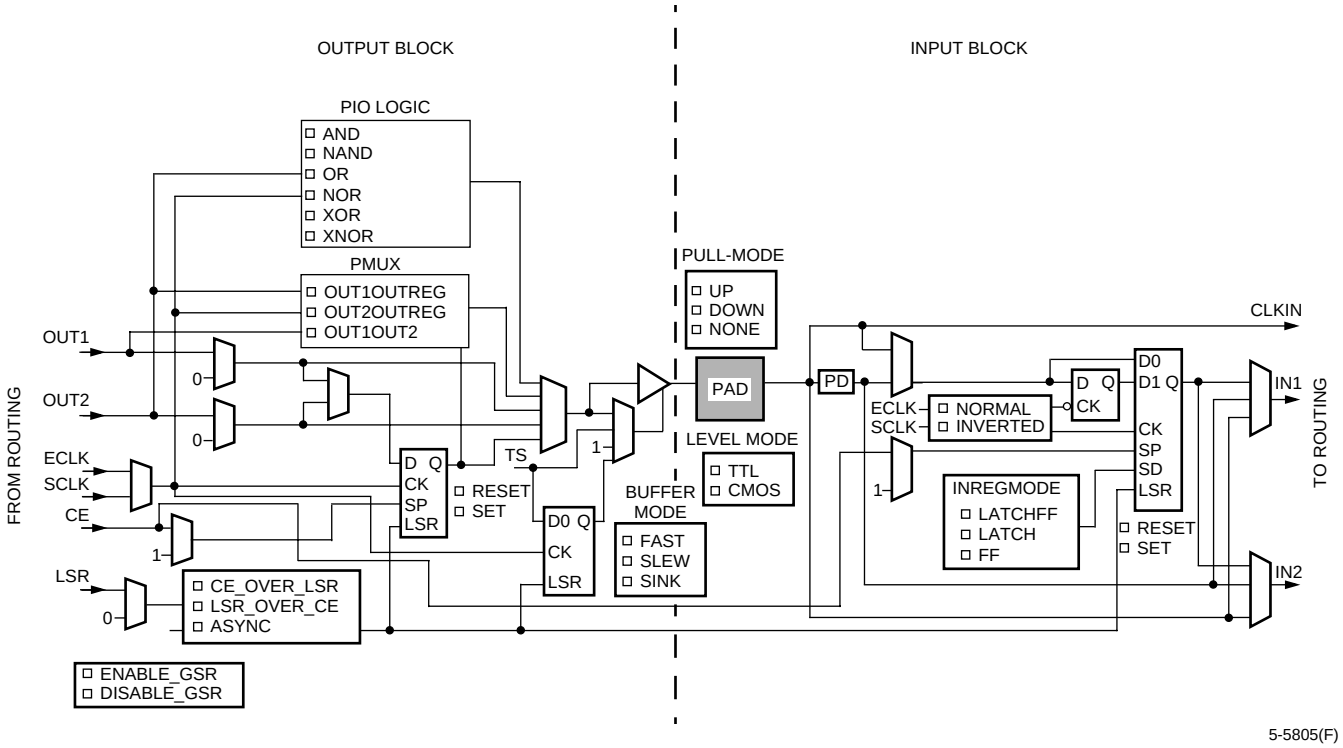


Figure 1. OR3Txx/OR3LxxxB

PIC Architecture Description (continued)

PIO Inputs

Each PIO input has six major options associated with it. These options are automatically selected when the corresponding library elements are inferred or instantiated. They can also be manually selected in the EPIC editor. These options are as follows:

- 3Cxxx input level: TTL or CMOS
- 3Txx input level: 5 V tolerant (5 V PCI compliant) or 3 V PCI compliant (clamped)
- 3LxxxB input level: 5 V tolerant (5 V PCI compliant) or 3 V PCI compliant (clamped)
- Input speed: fast or delayed
- Float value: pull-up (100 kΩ), pull-down (50 kΩ), none
- Input register mode: latch, FF, fast zero-hold FF, none (direct input)
- Input register/latch clock sense: inverted or noninverted
- Input selection: IN1, IN2, and/or clock input

PIO Outputs

Each PIO output has 10 major options associated with it. These options are automatically selected when the corresponding library elements are inferred or instantiated. They can also be manually selected in the EPIC editor. These options are as follows:

- Output drive current: 12 mA sink/6 mA source or 6 mA sink/3 mA source
- Output function: normal or fast open-drain
- Output speed: fast, slewlim, sinklim
- Output source: FF direct out, general routing
- 3-state source: FF direct out, general routing
- Output polarity: active-high or -low
- 3-state polarity: active-high or -low
- Output FF clock source: express, system
- Output register clock sense: inverted or noninverted
- Logic options: MUX options: OUT1/OUT2, OUT1/OUTREG or OUT2/OUTREG; logic options: AND, NAND, OR, NOR, XOR, and XNOR (between OUT1/OUT2)

PIC Architecture Description (continued)

PIC Routing Resources

Each PIC provides routing resources between the four PIOs, adjacent PICs, adjacent PLCs, and to the rest of the PICs and PLCs in the device. These resources include the following:

- PIC pairs and adjacent PLCs
- Output switching block: one per PIC, connects PIO outputs and controls to PIC routing resources
- Clock spine switch block: one per PIC pair, connects PIC clock resources to global clock spines
- pSW: two groups of eight lines per PIC, connecting to PIOs in groups of four
- px1: five lines per PIC, traverse one PIC, broken by a CIP in middle of PIC
- px2: five lines per PIC, traverse 2 PICs
- px5: 10 lines per PIC, traverse 5 PICs
- pxH: eight lines per PIC, traverse half of a side of the array
- pxL: 10 lines per PIC, traverse entire side of the array
- Routing buffers

Figure 2 shows the left PIC of a top edge PIC pair and part of the right PIC. Along the bottom of the image is a portion of a PLC.

PIC Architecture Description (continued)

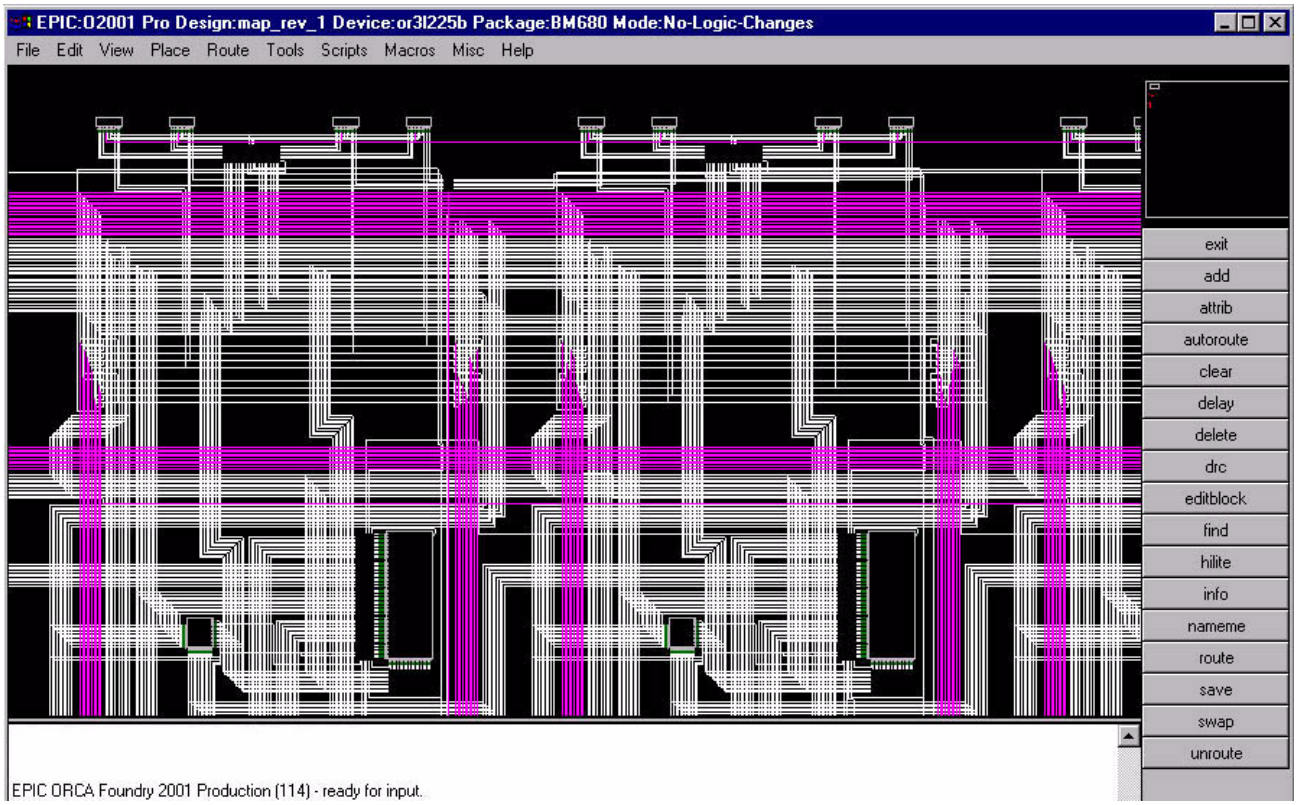


Figure 2. Series 3 PIC in EPIC

PIC Clocking

All four PIO within a PIC share a common local system clock (SCLK), while all PICs on a given side of the device share a common express clock (ECLK). These clocks can be used to control the PIO input register/latch, input latch, output register, output logic, and output MUX. All four PIO within a PIC also share a common clock enable, which can control the input register/latch and output register of each PIO. Both the clock and the clock enable can be configured for active-high or active-low operation. The common system clock for a PIC can originate from the system clock spine or from one of the local pSW routing segments.

PIC Architecture Description (continued)

Also, the direct CLKIN inputs from each PIO in the PIC pair (eight PIOs) and the local pSW segments are routed to the system clock spine switching block, whose output drives the clock spines. This allows any I/O pin or internal logic to drive the internal clock network allowing up to 40 global clocks per device. The common ECLK for a device side originates from the CLKCNTRLx function block which resides in the middle of the side of the FPGA on which the PIC itself resides. This CLKCNTRLx function block is driven by the dedicated ECLK input pad on that side of the FPGA or from the secondary ECLK pins in the lower-left and upper-right corners. These pin restrictions must be followed for the ECLK pins, but they do yield improved system performance, especially for I/O setup times and clock-to-out times. The overall Series 3 clock routing diagram for ECLK and system clock are shown in Figure 3. This express clock delivers the fastest clock to output delays from a registered output.

PIC Set/Reset

All four PIOs within a PIC share a common local set/reset. This signal can be synchronous or asynchronous. If synchronous, it can be set for CE_OVER_LSR or LSR_OVER_CE operation. The LSR can be configured for active-high operation or active-low operation. Global set/reset can also be disabled.

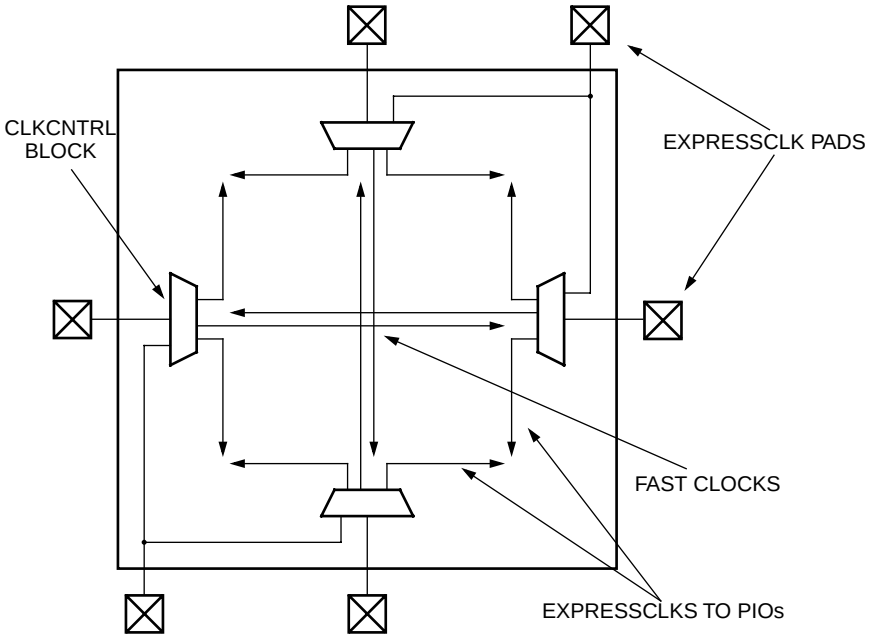


Figure 3. Clock Routing

Design Implementation

To use the Series 3 PIC's features in an HDL design, you must determine whether the synthesis tools are inferring the desired PIC resources from the generic HDL source code, and whether these PIC resources are being connected to internal resources correctly. This can be done by viewing the synthesis output netlist, which for the *ORCA* Foundry tool flow must be an EDIF file. You can view this EDIF file with any text editor. The EDIF file should contain PIC library cell declarations and instantiations with names identical to those listed in the Series 3 macro library (see cell tables below). Below is an EDIF netlist which uses several Series 3 PIC cells. The actual design is not important, but notice the cell declarations in the top portion of the EDIF file and the instances in the lower portion.

```
(edif pictest
    :
  (cell IFS1P3DX (cellType GENERIC)                ⇐ PIC INPUT FLIP-FLOP
    CELL PORTS HERE
    CELL PROPERTIES HERE
  (cell OFS1P3DX (cellType GENERIC)                ⇐ PIC OUTPUT FLIP-FLOP
    CELL PORTS HERE
    CELL PROPERTIES HERE
  (cell OB12F (cellType GENERIC)                   ⇐ FAST 7OUTPUT BUFFER
    CELL PORTS HERE
    CELL PROPERTIES HERE
(cell OSMUX21 (cellType GENERIC)                   ⇐ PIC OUTPUT MUX
    CELL PORTS HERE
    CELL PROPERTIES HERE
(cell ILF2P3DX (cellType GENERIC)                  ⇐ PIC INPUT LATCHED FLIP-FLOP
    CELL PORTS HERE
    CELL PROPERTIES HERE
(cell IBM (cellType GENERIC)                       ⇐ INPUT BUFFER
    CELL PORTS HERE
    CELL PROPERTIES HERE

       OTHER CELLS DECLARED HERE
 (library work
  (edifLevel 0)
  (technology (numberDefinition ))
  (cell pictest (cellType GENERIC)
   (view structure  (viewType NETLIST)
    (interface
     (port clk (direction INPUT))
     (port eckr (direction INPUT)
      (property loc (string "131")))               ⇐ ECKR PAD LOCATION
     (port ckena (direction INPUT))
    OTHER PORTS HERE
    SYNTHESIS DESIGN PROPERTIES HERE                        INSTANTIATIONS
       (contents                                                    ⇓
     (instance right_xclk (viewRef NETLIST  (cellRef CLKCNTLR (libraryRef orca3c ))))
     (instance ireg_ckena (viewRef NETLIST  (cellRef IFS1P3DX (libraryRef orca3c ))))
     (instance g1_0_oreg_dout_a (viewRef NETLIST (cellRef OFS1P3DX (libraryRef orca3c ))))
     (instance g1_0_out_buf (viewRef NETLIST  (cellRef OB12F (libraryRef orca3c ))))
     (instance g1_0_adlatch (viewRef NETLIST  (cellRef IFS1S1D (libraryRef orca3c ))))
     (instance g1_0_out_mux (viewRef NETLIST  (cellRef OSMUX21 (libraryRef orca3c ))))
     (instance addr_dec (viewRef NETLIST  (cellRef SAND8 (libraryRef orca3c ))))
     (instance ireg_din_b (viewRef NETLIST  (cellRef ILF2P3DX (libraryRef orca3c ))))
     (instance ix201 (viewRef NETLIST  (cellRef GSR (libraryRef orca3c ))))
     (instance ix205 (viewRef NETLIST  (cellRef OB6 (libraryRef orca3c ))))
     (instance ix222 (viewRef NETLIST  (cellRef IBM (libraryRef orca3c ))))
    OTHER CELLS INSTANTIATED HERE
    NETS DECLARED HERE
 (design pictest (cellRef pictest (libraryRef work ))))
```

Design Implementation (continued)

If you do not wish to view the EDIF file in a text editor, you may have another option. Most synthesis vendors' tools support a schematic viewer. This viewer reads in your EDIF netlist and displays the design in schematic form, including cell references. For example, in Exemplar's *Galileo Extreme** you can invoke a tool named Netscope, which will display the entire netlist in schematic form. It even has an option to invoke a text editor to view the ASCII text version of the netlist simultaneously. With it you can highlight a symbol in the schematic viewer, and it will automatically highlight the cross-referenced instantiation in the EDIF file.

Some synthesis vendors' tools might not infer these elements. If the synthesis tools are not correctly inferring the desired PIC resources (or not at all), then the designer must resort to instantiating most (or all) of the PIC resources directly into the source code.

Many of the newest synthesis tool releases do infer at least some PIC resources. For example, Exemplar's *Leonardo Spectrum** tool will infer PIC registers from generic VHDL source code. However, depending on how the source code is written, there may be concerns as to how those PIC registers connect to internal logic. For example, if the source code defines some registered complex logic function, and that register output is declared as an output pad, *Leonardo Spectrum* will most likely place the register in a PIC cell. However, the complex logic will be implemented inside PFU resources in the core. There may be significant routing delays to get from the PFU output to the PIC register input.

You may also need the registered output to feed back to other internal logic. It could be possible to rewrite the source code to force synthesis to place the logic and register together in a PFU, but then the register output must route to a nonregistered I/O. Or you could leave the register in the PIC, declare the pad as an in/out bidirectional pad and use the input register in the same PIO to bring the signal back into the device.

This trade-off between internal registering and PIC registering can be eliminated by double-registering synchronous signals destined for output. Thus, the complex logic has a short route inside a PFU to be registered, and this register can feed other internal logic as well as the PIC register which drives the fast output. No in/out pad is required (unless the signal is supposed to be a bidirectional bus already). The VHDL code below illustrates some of these concepts.

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- Specifically with Spectrum tool inferring PIC cells
library IEEE;
use IEEE.std_logic_1164.all;

entity spectrum_test is
   port ( clk, resetn:                    in   std_logic;
      ain, bin, cin:              in   std_logic;
      aout, bout, cout, dout:              out std_logic;
      eio:                        inout std_logic;
         tri:                             in   std_logic;
      fout:                               out std_logic );
end spectrum_test;

architecture Behavioral of spectrum_test is
      signal reg_a, reg_b, reg_c: std_logic;
      signal reg_d, reg_e, reg_f: std_logic;
begin
      synch: process (clk, resetn)
      begin
            if (resetn = '0') then
              reg_a <= '0'; reg_b <= '0'; reg_c <= '0';
              reg_d <= '0'; reg_e <= '0'; reg_f <= '0';
              aout  <= '0'; bout <= '0'; cout  <= '0';
              dout  <= '0'; fout  <= '0';
            elsif (clk'EVENT and clk = '1') then
              reg_a <= ain;
              aout  <= reg_a;
              reg_b <= bin;
              bout  <= reg_b;
              reg_c <= cin;

              if (reg_a = '1' and reg_b = '1' and reg_c = '1') then  cout
<= '1';
              else  cout <= '0';
              end if;

              if (reg_a = '1' and reg_b = '0' and reg_c = '0') then  reg_d
<= '1';
              else  reg_d <= '0';
              end if;
              dout <= reg_d;

              if (reg_a = '0' and reg_b = '1' and reg_c = '1') then  reg_e
<= '0';
              else  reg_e <= '1';
              end if;

              reg_f <= eio;
              fout  <= reg_f;
            end if;
      end process synch;
      eio <= reg_e when tri = '1' else 'Z';
end Behavioral;
```

* *Galileo Extreme* and *Leonardo Spectrum* are trademarks of Exemplar Logic, Inc.

Design Implementation (continued)

When this code is synthesized by *Leonardo Spectrum*, the following is implemented:

1. reg_a, reg_b and reg_c are all input registers in separate PIOs.
2. reg_a drives another register in a different PIO, which drives the aout pad. reg_b drives another register in a different PIO, which drives the bout pad.
3. The combinatorial logic (LUT) which drives cout is placed in a PFU. However, the register is placed in a PIO, and this register drives the cout pad.
4. reg_d is a register inside a PFU, driven by combinatorial logic (LUT) in the same PFU. reg_d then drives a register in a separate PIO which drives the dout pad.
5. eio is a bidirectional pad, driven by reg_e. reg_e is an output register in the same PIO as the eio pad. However, the combinatorial logic (LUT) which drives reg_e is placed in a PFU. reg_f is an input register in the same PIO as the eio pad. reg_f drives another register in a different PIO. This register drives the fout pad.

Notice that reg_d could fan out to many places inside the array without adversely affecting douts clock-to-output timing.

PIC Library Cells

There are numerous *ORCA* Series 3 macro library cells that may be used to map to PIC resources such as input buffers, output buffers, bidirectional buffers, PIO gates, PIO multiplexers, PIO input flip-flops, PIO output flip-flops, PIO input latches, and PIO input latched flip-flops. Some of these library cells may not be inferred by the synthesis tools, depending on which version of synthesis tools are being used. Therefore, in many cases, the user must instantiate these library cells into his/her source code to achieve the desired function. Shown below is a list of the available I/O and PIO cells for Series 3, and a checklist of features:

I/O Cells for Use with Series 3

Table 5. Input Buffer Cells

Cell	TTL	CMOS	Pull-Up	Pull-Down	Delay
IBM	—	√	—	—	—
IBMS	—	√	—	—	√
IBMPD	—	√	—	√	—
IBMPDS	—	√	—	√	√
IBMPU	—	√	√	—	—
IBMPUS	—	√	√	—	√
IBT	√	—	—	—	—
IBTS	√	—	—	—	√
IBTPD	√	—	—	√	—
IBTPDS	√	—	—	√	√
IBTPU	√	—	√	—	—
IBTPUS	√	—	√	—	√

Notes:
Pull-ups are 100 kΩ.

Pull-downs are 50 kΩ.

Delay is dependent on speed grade and array, but guarantees zero-hold.

Design Implementation (continued)

Table 6. Output Buffer Cells

Cell	Sinklim	Slewlim	Fast	3-State	Pull-Up	Pull-Down
OB6	√	—	—	—	—	—
OB12	—	√	—	—	—	—
OB12F	—	—	√	—	—	—
OBZ6	√	—	—	√	—	—
OBZ6PD	√	—	—	√	√	—
OBZ6PU	√	—	—	√	—	√
OBZ12	—	√	—	√	—	—
OBZ12PD	—	√	—	√	√	—
OBZ12PU	—	√	—	√	—	√
OBZ12F	—	—	√	√	—	—
OBZ12FPD	—	—	√	√	√	—
OBZ12FPU	—	—	√	√	—	√

Notes:
Sinklim is 6 mA sink and 3 mA source.
Slewlim is 12 mA sink and 6 mA source.
Fast is 12 mA sink and 6 mA source.
Pull-ups are 100 kΩ.
Pull-downs are 50 kΩ.

Table 7. Bidirectional Buffer Cells

Cell	CMOS Input	TTL Input	Sinklim	Slewlim	Fast	3-State	Pull-Up	Pull-Down
BMZ6	√	—	√	—	—	√	—	—
BMZ6PD	√	—	√	—	—	√	—	√
BMZ6PU	√	—	√	—	—	√	√	—
BMZ12	√	—	—	√	—	√	—	—
BMZ12PD	√	—	—	√	—	√	—	√
BMZ12PU	√	—	—	√	—	√	√	—
BMZ12F	√	—	—	—	√	√	—	—
BMZ12FPD	√	—	—	—	√	√	—	√
BMZ12FPU	√	—	—	—	√	√	√	—
BTZ6	—	√	√	—	—	√	—	—
BTZ6PD	—	√	√	—	—	√	—	√
BTZ6PU	—	√	√	—	—	√	√	—
BTZ12	—	√	—	√	—	√	—	—
BTZ12PD	—	√	—	√	—	√	—	√
BTZ12PU	—	√	—	√	—	√	√	—
BTZ12F	—	√	—	—	√	√	—	—
BTZ12FPD	—	√	—	—	√	√	—	√
BTZ12FPU	—	√	—	—	√	√	√	—

Notes:
Sinklim is 6 mA sink and 3 mA source.
Slewlim is 12 mA sink and 6 mA source.
Fast is 12 mA sink and 6 mA source.
Pull-ups are 100 kΩ.
Pull-downs are 50 kΩ.

Design Implementation (continued)

Table 8. Bidirectional Buffer with Delayed Input Cells

Cell	CMOS Input	TTL Input	Sinklim	Slewlim	Fast	3-State	Pull-Up	Pull-Down
BMS6	√	—	√	—	—	√	—	—
BMS6PD	√	—	√	—	—	√	—	√
BMS6PU	√	—	√	—	—	√	√	—
BMS12	√	—	—	√	—	√	—	—
BMS12PD	√	—	—	√	—	√	—	√
BMS12PU	√	—	—	√	—	√	√	—
BMS12F	√	—	—	—	√	√	—	—
BMS12FPD	√	—	—	—	√	√	—	√
BMS12FPU	√	—	—	—	√	√	√	—
BTS6	—	√	√	—	—	√	—	—
BTS6PD	—	√	√	—	—	√	—	√
BTS6PU	—	√	√	—	—	√	√	—
BTS12	—	√	—	√	—	√	—	—
BTS12PD	—	√	—	√	—	√	—	√
BTS12PU	—	√	—	√	—	√	√	—
BTS12F	—	√	—	—	√	√	—	—
BTS12FPD	—	√	—	—	√	√	—	√
BTS12FPU	—	√	—	—	√	√	√	—

Notes:
Sinklim is 6 mA sink and 3 mA source.
Slewlim is 12 mA sink and 6 mA source.
Fast is 12 mA sink and 6 mA source.
Pull-ups are 100 kΩ.
Pull-downs are 50 kΩ.
Delay is dependent on speed grade, but guarantees zero-hold.

Table 9. PIC Cells for Use with Series 3

Cell	Type	System Clock	Express Clock
OEAND2	2 Input AND	—	√
OEND2	2 Input NAND	—	√
OENR2	2 Input NOR	—	√
OEOR2	2 Input OR	—	√
OEXNOR2	2 Input XNOR	—	√
OEXOR2	2 Input XOR	—	√
OSAND2	2 Input AND	√	—
OSND2	2 Input NAND	√	—
OSNR2	2 Input NOR	√	—
OSOR2	2 Input OR	√	—
OSXNOR2	2 Input XNOR	√	—
OSXOR2	2 Input XOR	√	—

Design Implementation (continued)

Table 10. PIC Multiplexer Cells

Cell	Type	System Clock	Express Clock
OEMUX21	2 to 1 MUX	—	√
OSMUX21	2 to 1 MUX	√	—

Table 11. PIC Input Flip-Flop Cells

Cell	System Clock (↑ Edge)	Clock Enable	Asynch. Preset	Asynch. Clear	Synch. Clear 1	Synch. Clear 2	Synch. Preset 1	Synch. Preset 2
IFS1P3BX	√	√	√	—	—	—	—	—
IFS1P3DX	√	√	—	√	—	—	—	—
IFS1P3IX	√	√	—	—	√	—	—	—
IFS1P3IZ	√	√	—	—	—	√	—	—
IFS1P3JX	√	√	—	—	—	—	√	—
IFS1P3JZ	√	√	—	—	—	—	—	√

Notes:
All control signals are positive level.

Synch. Clear 1 is clear overrides enable. Synch. Clear 2 is enable over clear.

Synch. Preset 1 is preset overrides enable. Synch. Preset 2 is enable over preset.

Table 12. PIC Output Flip-Flop Cells

Cell	Express Clock (↑ Edge)	System Clock (↑ Edge)	Clock Enable	Asynch. Preset	Asynch. Clear	Synch. Clear 1	Synch. Clear 2	Synch. Preset 1	Synch. Preset 2
OFE1P3BX	√	—	√	√	—	—	—	—	—
OFE1P3DX	√	—	√	—	÷	—	—	—	—
OFE1P3IX	√	—	√	—	—	÷	—	—	—
OFE1P3IZ	√	—	√	—	—	—	√	—	—
OFE1P3JX	√	—	√	—	—	—	—	√	—
OFE1P3JZ	√	—	√	—	—	—	—	—	√
OFS1P3BX	—	√	√	√	—	—	—	—	—
OFS1P3DX	—	√	√	—	√	—	—	—	—
OFS1P3IX	—	√	√	—	—	√	—	—	—
OFS1P3IZ	—	√	√	—	—	—	√	—	—
OFS1P3JX	—	√	√	—	—	—	—	√	—
OFS1P3JZ	—	√	√	—	—	—	—	—	√

Notes:
All control signals are positive level.

Synch. Clear 1 is clear overrides enable. Synch. Clear 2 is enable over clear.

Synch. Preset 1 is preset overrides enable. Synch. Preset 2 is enable over preset.

Design Implementation (continued)

Table 13. PIC Input Latched Flip-Flop Cells

Cell	System Clocked Flip-Flop (↑ Edge)	System Clock Enable	Express Clocked Latch (Negative Level)	Asynch. Preset	Asynch. Clear	Synch. Clear 1	Synch. Clear 2	Synch. Preset 1	Synch. Preset 2
ILF2P3BX	√	√	√	√	—	—	—	—	—
ILF2P3DX	√	√	√	—	√	—	—	—	—
ILF2P3IX	√	√	√	—	—	√	—	—	—
ILF2P3IZ	√	√	√	—	—	—	√	—	—
ILF2P3JX	√	√	√	—	—	—	—	√	—
ILF2P3JZ	√	√	√	—	—	—	—	—	√

Notes:
All control signals are positive level except Express clock input.

Synch. Clear 1 is clear overrides enable. Synch. Clear 2 is enable over clear.

Synch. Preset 1 is preset overrides enable. Synch. Preset 2 is enable over preset.

Table 14. PIC Input Latch Cells

Cell	System Clocked Latch Enable	Asynch. Preset	Asynch. Clear	Synch. Clear	Synch. Preset
IFS1S1B	√	√	—	—	—
IFS1S1D	√	—	√	—	—
IFS1S1I	√	—	—	√	—
IFS1S1J	√	—	—	—	√

Note: All control signals are positive level.

I/O Cell and PIC Cell Instantiation in VHDL

If your synthesis tool will not infer the PIC resources you desire, you must instantiate them in your design. In order to instantiate any component into a VHDL design, the component black box must first be declared in the architecture. For example:

```
-- VHDL netlist for testing 3C/3T PIC architecture: Example 1
library IEEE;
use IEEE.std_logic_1164.all;

entity pic_example1 is
    port ( clk, eckr       : in  std_logic;
                ckena, clear, tri : in  std_logic;
          datai, datao      : in  std_logic);

        attribute LOC : string;
        attribute LOC of eckr : signal is "131";  -- 208 pin package
end pic_example1;

architecture Structure of pic_example1 is
    -- internal signal declarations
    signal drhi, drlo: std_logic;
    signal eckr_int  : std_logic;
    signal datai_int : std_logic;
    signal rdata1, rdata2 : std_logic;
    signal datao_int : std_logic;
    -- local component declarations
        COMPONENT vhi                           -- logic '1' driver
        PORT( z: OUT std_logic := 'X');
        END COMPONENT;

        COMPONENT vlo                           -- logic '0' driver
        PORT( z: OUT std_logic := 'X');
        END COMPONENT;

        COMPONENT clkcntlr                              -- clock controller, right edge
        PORT(               clkin : IN std_logic := 'X';
              shutoff : IN std_logic := 'X';
              clkout : OUT std_logic := 'X');
        END COMPONENT;

        COMPONENT ibmpus                                -- CMOS input buffer, pullup & delay
        PORT(   i:  IN std_logic := 'X';
                        o:  OUT std_logic);
        END COMPONENT;

        COMPONENT ilf2p3dx                      -- PIC input latched FF
        PORT(   d    : IN std_logic := 'X';
              sp   : IN std_logic := 'X';
              eclk: IN std_logic := 'X';
              sclk: IN std_logic := 'X';
              cd   : IN std_logic := 'X';
              q    : OUT std_logic := 'X' );
        END COMPONENT;

        COMPONENT fd1p3dx                               -- PLC flip-flop
        PORT(   d : IN std_logic := 'X';
              sp: IN std_logic := 'X';
              ck: IN std_logic := 'X';
              cd: IN std_logic := 'X';
              q : OUT std_logic := 'X';
```

Note: *IEEE* is a registered trademark of The Institute of Electrical and Electronics Engineers, Inc.

I/O Cell and PIC Cell Instantiation in VHDL (continued)

```
        qn: OUT std_logic := 'X');
    END COMPONENT;

    COMPONENT ofs1p3dx                    -- PIC output FF, System clock
    PORT(   d : IN std_logic := 'X';
        sp: IN std_logic := 'X';
        sclk: IN std_logic := 'X';
        cd: IN std_logic := 'X';
        q : OUT std_logic := 'X');
    END COMPONENT;

    COMPONENT obz12fpd                    -- fast output buffer,3-state, pull-down
    PORT(   i:  IN std_logic := 'X';
        t:  IN std_logic := 'X';
        o:  OUT std_logic);
    END COMPONENT;
begin
:
end Structure;
```

Note a convenient source of PIC components (or any *ORCA* components for that matter) can be found in a VHDL source library file called orcacomp.vhd. Entire component declarations can be copied from this file directly into the design source code.

The component may then be instantiated as many times as desired in the design. For example, . . .

```
-- VHDL netlist for testing 3C/3T PIC architecture: Example 1
library IEEE;
use IEEE.std_logic_1164.all;

entity pic_example1 is
:
end pic_example1;

architecture Structure of pic_example1 is
   -- internal signal declarations
   -- local component declarations
begin
   -- component instantiation statements
      drive_hi: vhi  port map (drhi);
      drive_lo: vlo  port map (drlo);
      right_xclk: clkcntlr  port map (eckr, drlo, eckr_int);
      in_buffer: ibmpus  port map (datai,datai_int);
      in_reg: ilf2p3dx  port map(datai_int, ckena, eckr_int, clk, clear, rdata1);
      pic_reg: fd1p3dx  port map(rdata1, ckena, clk, clear, rdata2);  -- qn not mapped
      out_reg: ofs1p3dx  port map(rdata2, ckena, clk, clear, datao_int);
      out_buffer: obz12fpd  port map(datao_int,tri,datao);
end Structure;
```

If the design is being synthesized, the Foundry mapper will see the black-box components and use the appropriate library elements. However, for simulation, the components must be configured using the supplied simulation libraries. Set up the simulator to point to the correct directory. (For MTI, use the vsystem.ini or modelsim.ini file to do this). Then use a configuration block to tell the simulator which library elements to use for each instance. For example, . . .

```
-- VHDL netlist for testing 3C/3T PIC architecture: Example 1
-- vsystem.ini or modelsim.ini contains : “ORCA3 = $FOUNDRY/vhdl/data/orca3/mti/work”
--      where $FOUNDRY points to the Foundry tools base directory
library IEEE, ORCA3;
```

I/O Cell and PIC Cell Instantiation in VHDL (continued)

```
use IEEE.std_logic_1164.all;
use ORCA3.orcacomp.all;

entity pic_example1 is

end pic_example1;

architecture Structure of pic_example1 is

end Structure;

configuration Structure_CON of pic_example1 is
   for Structure
      for all: vhi      use entity ORCA3.vhi(V);      end for;
      for all: vlo      use entity ORCA3.vlo(V);      end for;
      for all: clkcntlr use entity ORCA3.clkcntlr(V); end for;
      for all: ibmpus   use entity ORCA3.ibmpus(V);   end for;
      for all: ilf2p3dx use entity ORCA3.ilf2p3dx(V); end for;
      for all: fd1p3dx  use entity ORCA3.fd1p3dx(V);  end for;
      for all: ofs1p3dx use entity ORCA3.ofs1p3dx(V); end for;
      for all: obz12fpd use entity ORCA3.obz12fpd(V); end for;
  end for;
end Structure_CON;
```

I/O Cell and PIC Cell Instantiation in *Verilog**

Declaring and instantiating components into a *Verilog* design is even easier. Simply declare *Verilog* modules from the library with instance names, and map the ports accordingly. (The *Verilog* library modules are each stored as a separate *Verilog* file (*module_name*.v) in */foundry_directory*/verilog/data/orca3.) Here is an example of the same circuit rewritten in Verilog:

```
// Verilog design for testing 3C/3T PIC: Example 1
module PIC_EXM1(CLK,ECKR,CKENA,CLEAR,TRI,DATAI,DATAO);
  input  CLK,ECKR,CKENA,CLEAR,TRI,DATAI;
  output DATAO;
  reg    RDATA1, RDATA2;
  wire   DRHI, DRLO, ECKR_INT, DATAI_INT, DATAO_INT;
  // instantiate components
  VHI VH1 (DRHI);
  VLO VL1 (DRLO);
  CLKCNTLR CL1 (ECKR, DRLO, ECKR_INT);
  IBMPUS IB1 (DATAI, DATAI_INT);
  ILF2P3DX ILF1 (DATAI_INT, CKENA, ECKR_INT, CLK, CLEAR, RDATA1);
  FD1P3DX  FF1 (RDATA1, CKENA, CLK, CLEAR, RDATA2);  // QN not mapped
  OFS1P3DX OF1 (RDATA2, CKENA, CLK, CLEAR, DATAO_INT);
  OBZ12FPD OB1 (DATAO_INT,TRI,DATAO);
endmodule
```

To simulate this design, you will need to include the libraries of all Series 3 modules you wish to instantiate. On a *UNIX*† workstation, you can modify your design to add the following compiler directives to the top of your *Verilog* source code (where */foundry_directory* is the path to the *ORCA* Foundry tools base directory):

```
`timescale 1 ns / 100 ps
`include "/foundry_directory/verilog/data/orca3/VHI.v"
`include "/foundry_directory/verilog/data/orca3/VLO.v"
`include "/foundry_directory/verilog/data/orca3/CLKCNTLR.v"
`include "/foundry_directory/verilog/data/orca3/IBMPUS.v"
`include "/foundry_directory/verilog/data/orca3/ILF2P3DX.v"
```

* *Verilog* is a registered trademarks of Cadance Design Systems, Inc.
† *UNIX* is a registered trademark of X/Open Company, Ltd.

I/O Cell and PIC Cell Instantiation in VHDL (continued)

```
`include "/foundry_directory/verilog/data/orca3/FD1P3DX.v"
`include "/foundry_directory/verilog/data/orca3/OFS1P3DX.v"
`include "/foundry_directory/verilog/data/orca3/OBZ12FPD.v"

module PIC_EXM1(CLK,ECKR,CKENA,CLEAR,TRI,DATAI,DATAO);

end module
```

Or, you can just specify the library directory when compiling for simulation at the command prompt. For example, in MTI:

```
vlog -y $FOUNDRY/verilog/data/orca3 +libext+.v design.v
```

where $FOUNDRY points to your Foundry tools' base directory, and design.v is your top level source code.

Express Clock Instantiation in VHDL and *Verilog*

If PIC output flip-flops are used, the designer has the option of using an Express clock or a system clock. Also, if PIC input latched flip-flops are used, the designer must use Express clock to enable the latch. Also, several of the PIC gates and one of the PIC multiplexers accept express clock as an input. However, to use an Express clock, the designer must remember to instantiate the correct clock controller (CLKCNTLx) or programmable clock manager (PCM) in the design and LOC its corresponding input pad (ECKx or SECKxx). The choice of clock controller depends on which edge of the array the PICs in question reside on. If there are PICs on all four edges of the device which require Express clock, then either of the following must occur:

1. Instantiate all four clock controllers and LOC all four ECKx pads.
2. Instantiate both PCMs and LOC both corner SECKxx pads.
3. A combination of 1. and 2. which guarantees that all PICs are getting an Express clock.

Here is a short example of how to instantiate a CLKCNTLR cell and LOC its ECKR pad using attributes in VHDL source code. This design was targeted at an OR3T55 in a 208-pin SQFP package.

```
-- VHDL netlist for demonstrating 3C/3T express clock instantiation
library IEEE, ORCA3;
use IEEE.std_logic_1164.all;
-- use ORCA3.orcacomp.all;

entity pictest is
   port (    clk: in  std_logic;
       eckr: in  std_logic;                          ⇐ ECKR PORT LISTED
           -- other ports declared here
             );
      attribute LOC : string;
      attribute LOC of eckr : signal is "131";              ⇐ ECKR PAD LOCATION ATTRIBUTE
end pictest;

architecture Structure of pictest is
   -- internal signal declarations
            signal drlo    : std_logic;
            signal eckr_int: std_logic;
     -- other signals declared here
     -- local component declarations
      COMPONENT clkcntlr                            ⇐ RIGHT CLOCK CONTROLLER COMPONENT
      PORT(  clkin : IN std_logic := 'X';
             shutoff : IN std_logic := 'X';
              clkout : OUT std_logic := 'X' );
      END COMPONENT;

      COMPONENT vlo
```

I/O Cell and PIC Cell Instantiation in VHDL (continued)

```
      PORT( z: OUT std_logic := 'X'  );
      END COMPONENT;
            -- other components declared here
begin
   -- component instantiation statements
      drive_lo: vlo
      port map (drlo);

      right_xclk: clkcntlr                                    ⇐ RIGHT CLOCK CONTROLLER INSTANTIATED
      port map (eckr,
              drlo,                  -- assumes direct port mapping
              eckr_int);
            -- other components instantiated here
end Structure;
```

Here is the same example using *Verilog* source code. The ECKR pin location must be specified using the synthesis tools.

```
// Verilog design for testing 3C/3T PIC: Example 1
module PIC_EXM1(CLK,ECKR, …);
  input  CLK,ECKR, …;
  output …;
  reg       …;
  wire      DRLO, ECKR_INT, …;

  // instantiate components
  VLO VL1 (DRLO);
  CLKCNTLR CL1 (ECKR, DRLO, ECKR_INT);
   :
endmodule
```

Methods to Assign Special Properties to PIC Resources

The I/O registers, latches, buffers, pads, and other resources in each PIC frequently need additional information assigned to them in order to allow the Foundry tools to implement a functioning design with the desired timing. Things such as I/O pad locations, frequency, input setup times, clock-to-output times, output loads, etc, may need to be specified. The primary method of passing this information to the Foundry tools is through the preference file. However, other methods may exist to pass this information into the preference file, such as via VHDL attributes, synthesis directives, or by using a logical preference file. One advantage of the logical preference file is the ability to use wildcards to specify net names.

Methods to LOC I/O Pins

Source Code

There are several ways to assign the input and output pins of your design to the physical pad locations of the device. If you are using VHDL, you can make these assignments in the source code using attribute statements and the LOC property. The synthesis tool will pass this information into the EDIF file, and the mapper will convert them into the preference file. For example, to assign input name to pad 77:

```
entity pictest is
   port (  name: in  std_logic;
           -- other ports declared here
             );
      attribute LOC : string;
      attribute LOC of name : signal is "77";
end pictest;
```

In *Verilog*, there is no method of assigning pad locations to your I/O.

Preference File

The preference LOCATE command can be inserted into the preference file (.prf) to dictate where an I/O pad should be placed. For example, consider the following line in a preference file:

```
LOCATE COMP “name” SITE “77”;
```

This will force the Foundry tools to place the pad for signal name at pin 77 of the device.

Logical Preference File

There is no way to locate I/O pads in a logical preference file.

Synthesis Tools

Your synthesis tools will usually also have a method of entering I/O pad assignments using some commands and/or constraint file. This information should be converted into a Foundry preference file (.prf) for use by the Foundry tools, or inserted into the EDIF file during synthesis, in which case the mapper will convert them to preferences. Refer to your synthesis vendor’s documentation.

Methods to Assign Special Properties to PIC Resources (continued)

Methods to Specify PIC Timing Constraints: Input Setup (to a PIC Input Register)

An input setup time specifies a setup time requirement for registered input ports relative to a clock net.

Preference File

The OFFSET IN command can be used in the preference file to dictate an input's setup time. For example:

```
OFFSET IN COMP "inport_name"  5.0 NS BEFORE COMP "clk";
```

Logical Preference File (Use lp2prf to Translate into a Preference)

```
INPUT_SETUP inport_name 5 NS CLKNET = "clk";
```

Methods to Specify PIC Timing Constraints: Clock to Output (from a PIC Output Register)

A clock-to-output time specifies a maximum allowable output delay for registered output ports relative to a clock net.

Preference File

The OFFSET OUT command can be used in the preference file to dictate an output's clock-to-output time. For example:

```
OFFSET OUT COMP "outport_name"  20.0 NS AFTER COMP "clk";
```

Logical Preference File (Use lp2prf to Translate into a Preference)

```
CLOCK_TO_OUT outport_name 20 NS CLKNET = "clk";
```

Methods to Specify PIC Timing Constraints: Frequency/Period (from/to a PIC register)

A frequency or period value specifies the minimum frequency (or maximum period) at which sequential circuits must operate. When applied to PIC registers, it can apply to the timing between a PIC register and a PLC register, or between PIC registers. Note that this preference takes into account all logic and routing delays and any clock skew, so it is the most reliable method of preferencing a synchronous circuit's speed of operation. A FREQUENCY or PERIOD command can be used in the preference file to dictate a synchronous circuit's required clock rate. For example, for a fully synchronous 50 MHz design with clock input clk:

Preference File

```
FREQUENCY NET "clk" 50 MHz;
or
PERIOD NET "clk" 20 NS HIGH 10 NS;
```

Logical Preference File (Use lp2prf to Translate into a Preference)

```
FREQUENCY PORT "clk" 50 MHz;
or
PERIOD PORT "clk" 20 NS HIGH 10 NS;
```

Methods to Assign Special Properties to PIC Resources (continued)

Methods to Specify PIC Timing Constraints: Multicycle (from/to a PIC Register)

A multicycle value specifies a relaxation of a previously specified FREQUENCY or PERIOD preference on a synchronous path(s). For example, if one portion of a synchronous design may only need to operate at one-half the clock frequency of the rest of the design, then a 2X multicycle value may be assigned to those paths. When applied to PIC registers, remember that you wish to create a preference from FF to FF, not to an I/O pad. For example, suppose we want a multicycle preference from a PLC register to a PIC output register:

Preference File

MULTICYCLE “M1” START COMP “PFU_n” NET “net_name” END COMP “outport_name” 2 X;

M1 is just a label. PFU_n is a physical PFU name, net_name is a physical net name, and outport_name is an actual output port name.

Logical Preference File (Use lp2prf to translate into a preference)

MULTICYCLE FROM CELL “plcreg” TO CELL “pic_outreg” 2 X;

In this correct implementation, plcreg and pic_outreg are instance names of registers. The synthesis tool may create register instance names that differ drastically from names in the source code. Also note that this is **not** the same as the following incorrect line:

MULTICYCLE FROM CELL “plcreg” TO PORT “outport_name” 2 X;

This line creates a timing preference from a PLC register output all the way to an output pad named outport_name. This will create a define path preference and a maxdelay preference on that path in the .prf file when lp2prf is executed.

Other PIC Timing Information

Preferences, logical preferences, VHDL properties, and synthesis vendor commands also exist for specifying the following timing values. Refer to the *ORCA* Foundry documentation CD or your synthesis vendor’s documentation for information on how to implement these values.

maxdelay—Specifies a maximum total delay for a net or path. When applied to PIC timing, it is recommended that it should only be used for asynchronous I/O timing, either between some internal PLC logic and a PIC resource, or between a PIC resource and an external pad. If a PIO is being used synchronously, use the previously described synchronous preferences instead.

maxskew—Specifies a maximum signal skew between a driver and loads on a specified clock signal. When applied to PIC timing, it refers to the ECLK or SCLK signal which drives the corresponding PICs.

Other I/O Features

Preferences, logical preferences, VHDL properties, and synthesis vendor commands also exist for specifying such values as output load, slew rate, and input delay mode. Refer to the *ORCA* Foundry documentation CD or your synthesis vendor’s documentation for information on how to implement these values.

Methods to Assign Special Properties to PIC Resources (continued)

Applications

Microprocessor and/or Bus Interface with MUXed/deMUXed Address/Data

The Series 3 PIC is well designed for applications requiring an interface to external multiplexed address and data buses, such as may be found on various types of microprocessor buses (*Intel i960**) and system buses (PCI, VME64). In one case, the PICs may have to receive a multiplexed address/data bus and demultiplex it for internal use. In another case, the PICs may have to multiplex internal address and data to drive an external bus. In a bus with master/slave arbitration, the PICs may have to do both multiplexing and demultiplexing using bidirectional I/O.

* *Intel* and *i960* are registered trademarks of Intel Corporation.

Case 1: Input Demultiplexer (same clock for address and data)

To demultiplex an incoming address/data bus, the PIO latch or FF can be used to store one bit of the address on one edge of the SCLK, while an adjacent PLC FF stores a bit of the data on the other edge. This is shown in Figure 4. For example:

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- Input Demultiplex

library IEEE, ORCA3;
use IEEE.std_logic_1164.all;
      COMPONENT ifs1s1d              -- PIC input latch
      PORT(d: IN std_logic := 'X';
            sclk: IN std_logic := 'X';
            cd  : IN std_logic := 'X';
            q   : OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT fd1p3dx              -- PLC flip-flop
      PORT(d : IN std_logic := 'X';
            sp: IN std_logic := 'X';
            ck: IN std_logic := 'X';
            cd: IN std_logic := 'X';
            q : OUT std_logic := 'X';
            qn: OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT vhi
      PORT(  z: OUT std_logic := 'X' );
      END COMPONENT;

      COMPONENT vlo
      PORT(  z: OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT sand8                        -- SLIC and8
      PORT(a: IN std_logic := 'X';
            b: IN std_logic := 'X';
            c: IN std_logic := 'X';
            d: IN std_logic := 'X';
            e: IN std_logic := 'X';
            f: IN std_logic := 'X';
            g: IN std_logic := 'X';
            h: IN std_logic := 'X';
            z: OUT std_logic := 'X');
      END COMPONENT;
```

Methods to Assign Special Properties to PIC Resources (continued)

```
begin
   -- component instantiation statements
      drive_hi: vhi port map (drhi);
      drive_lo: vlo port map (drlo);

      g1: for i in 0 to 7 generate
        pio_latch: ifs1s1d port map (adin(i), clk, drlo, abus(i));

        plc_reg: fd1p3dx port map  (adin(i), csel, clk, drlo, dbus(i)); -- qn output not specified
      end generate;

      addr_dec: sand8
      port map (abus(0), abus(1), abus(2), abus(3),
               abus(4), abus(5), abus(6), abus(7),
               csel);
end Structure;
```

Case 2: Input Demultiplexing (seperate data clock and address enable)

If an address enable is used to latch the address instead of the clock, it is done by replacing the SCLK signal at the PIO latch with the address enable signal. Note that performing the address latch function in the FF and clocking the data into the PIO FF is also valid for Case 1 and Case 2. This allows trade-offs of I/O performance of these two functions.

```
use ORCA3.orcacomp.all;

entity demux1 is
   port ( clk: in  std_logic;
         adin: in  std_logic_vector(7 DOWNTO 0);
      -- outputs listed here
      );
end demux1;

architecture Structure of demux1 is
   -- internal signal declarations
   signal drhi, drlo    : std_logic;
   signal abus, dbus    : std_logic_vector(7 DOWNTO 0);

   -- local component declarations
      COMPONENT ifs1s1d                  -- PIC input latch
      PORT(   d   : IN std_logic := 'X';
              sclk: IN std_logic := 'X';
              cd  : IN std_logic := 'X';
              q   : OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT fd1p3dx                  -- PLC flip-flop
      PORT(           d : IN std_logic := 'X';
              sp: IN std_logic := 'X';
              ck: IN std_logic := 'X';
              cd: IN std_logic := 'X';
              q : OUT std_logic := 'X';
              qn: OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT vhi
      PORT(  z: OUT std_logic := 'X' );
      END COMPONENT;
```

Methods to Assign Special Properties to PIC Resources (continued)

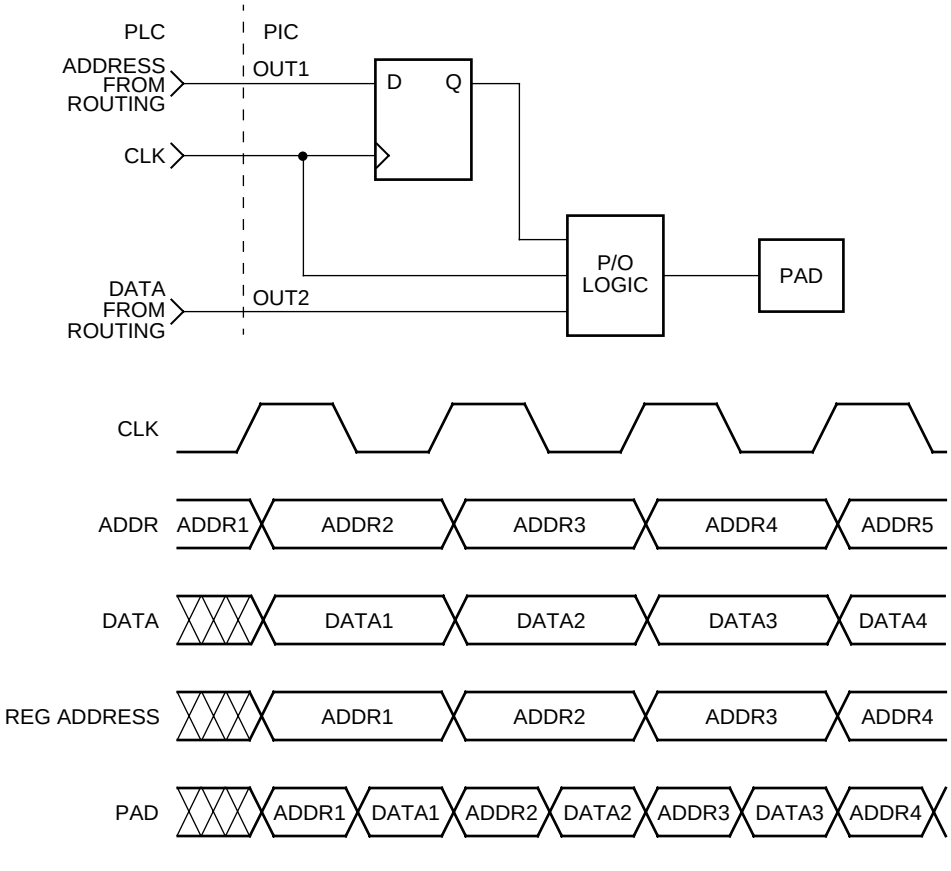
```
      COMPONENT vlo
      PORT(  z: OUT std_logic := 'X');
      END COMPONENT;
begin
   -- component instantiation statements
      drive_hi: vhi port map (drhi);
      drive_lo: vlo port map (drlo);

      g1: for i in 0 to 7 generate
        pio_latch: ifs1s1d port map (adin(i), clk, drlo, abus(i));

        plc_reg: fd1p3dx port map  (adin(i), drhi, clk, drlo, dbus(i)); -- qn output not specified
      end generate;
end Structure;
```

Case 3: Output Multiplexing

To multiplex separate buses into a single address/data output bus, use the OSMUX21 or OEMUX21 element and route the address bit to OUT1 and the data bit to OUT2. Depending on what phasing you want between address and data, you can use the PIO output FF to delay the address or data by another ECLK or SCLK cycle. This puts the PIO in OUT1OUTREG mode or OUT2OUTREG mode. This is shown in Figure 5.



5-5797(F)

Figure 4. PIC Output Multiplexing

Methods to Assign Special Properties to PIC Resources (continued)

Here is a VHDL example of how to instantiate such a circuit:

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- Output Multiplexer, OUT2OUTREG mode

library IEEE, ORCA3;
use IEEE.std_logic_1164.all;
use ORCA3.orcacomp.all;

entity omux1 is
   port ( clk: in  std_logic;
        adout: out std_logic_vector(7 DOWNTO 0);
       -- other inputs and outputs listed here
       );
end omux1;

architecture Structure of omux1 is
   -- internal signal declarations
   signal drhi    : std_logic;
   signal drlo    : std_logic;
   signal abus    : std_logic_vector(7 DOWNTO 0);
   signal rabus   : std_logic_vector(7 DOWNTO 0);
   signal dbus    : std_logic_vector(7 DOWNTO 0);
   -- local component declarations
      COMPONENT vhi
      PORT(  z: OUT std_logic := 'X' );
      END COMPONENT;

      COMPONENT vlo
      PORT(  z: OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT osmux21              -- PIC output Mux
      PORT(   d0: IN std_logic := 'X';
              d1: IN std_logic := 'X';
            sclk: IN std_logic := 'X';
              z : OUT std_logic := 'X');
      END COMPONENT;

      COMPONENT ofs1p3dx             -- PIC output FF
      PORT(   d : IN std_logic := 'X';
              sp: IN std_logic := 'X';
            sclk: IN std_logic := 'X';
              cd: IN std_logic := 'X';
              q : OUT std_logic := 'X');
      END COMPONENT;
begin
   -- component instantiation statements
      drive_hi: vhi port map (drhi);
      drive_lo: vlo port map (drlo);

      g1: for i in 0 to 7 generate
        pio_oreg: ofs1p3dx port map (abus(i), drhi, clk, drlo, rabus(i));
        pio_mux: osmux21 port map (rabus(i),dbus(i),clk,adout(i));
      end generate;
end Structure;
```

Methods to Assign Special Properties to PIC Resources (continued)

Fast Address Decode Using SLICs Next to PICs

An address input from an external bus can also be decoded very quickly using the SLICs in the PLCs adjacent to the PICs. The decode bit(s) could then be used to select various internal registers for reads or writes. Using the input demultiplexer example above (CASE 1), the address latch outputs from the PIOs could be routed to a nearby SLIC for decoding, and the decode bit(s) could drive the clock enable of the PLC data register to be written to, as seen in Figure 4.

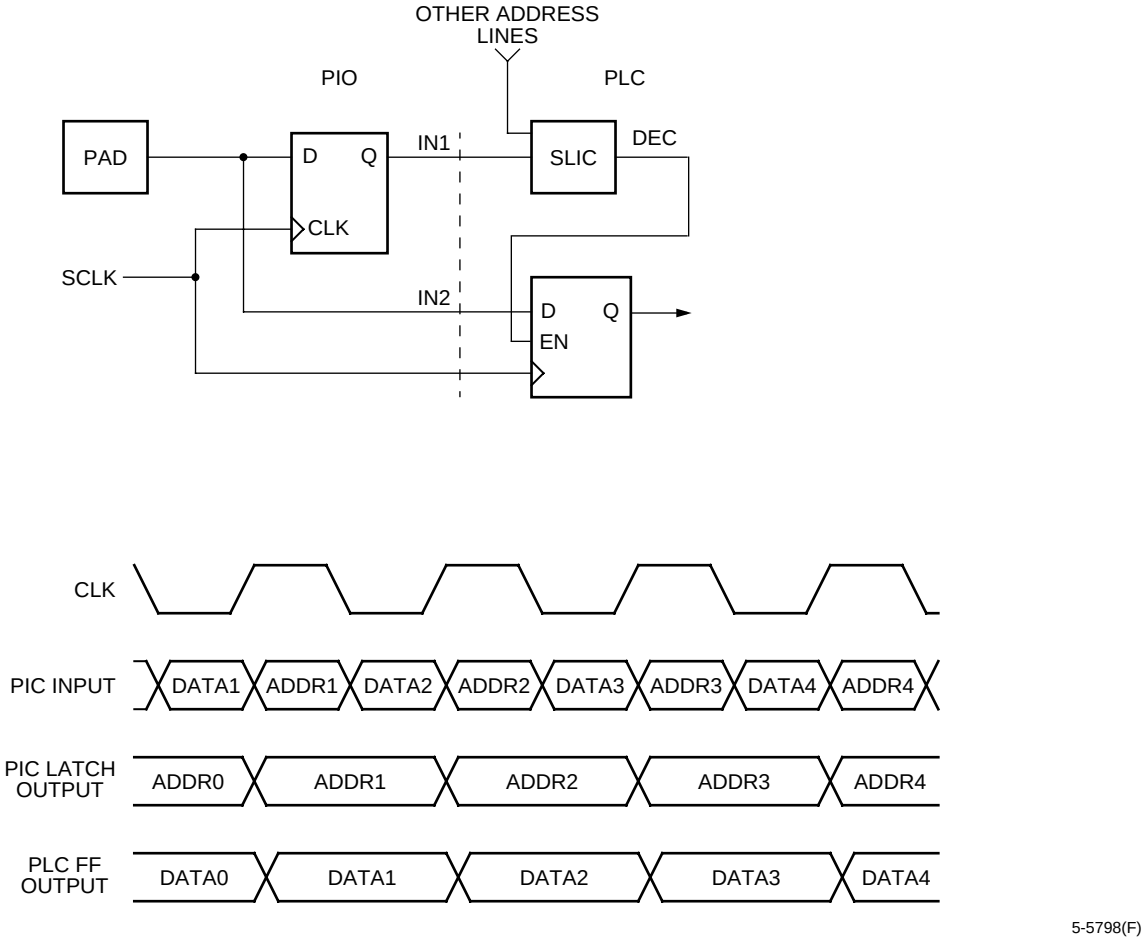


Figure 5. PIO Input Demultiplexing

To instantiate such a circuit using VHDL:

```
library IEEE, ORCA3;
use IEEE.std_logic_1164.all;
use ORCA3.orcacomp.all;

entity demux2 is
   port ( clk: in  std_logic;
        adin: in  std_logic_vector(7 DOWNTO 0);
      -- outputs listed here
      );
end demux2;
```

Methods to Assign Special Properties to PIC Resources (continued)

```
architecture Structure of demux2 is
   -- internal signal declarations
   signal drhi, drlo  : std_logic;
   signal abus, dbus   : std_logic_vector(7 DOWNTO 0);
   signal csel    : std_logic;

   -- local component declarations
```

Zero Hold and Programmable Delay Input

The designer can use the zero-hold flip-flop mode or the programmable delay input capabilities of the PIO to implement a zero hold input capture. The zero-hold flip-flop mode requires the combined use of both the SCLK and ECLK to drive the PIO flip-flop's clock pin and the latch clock, respectively.

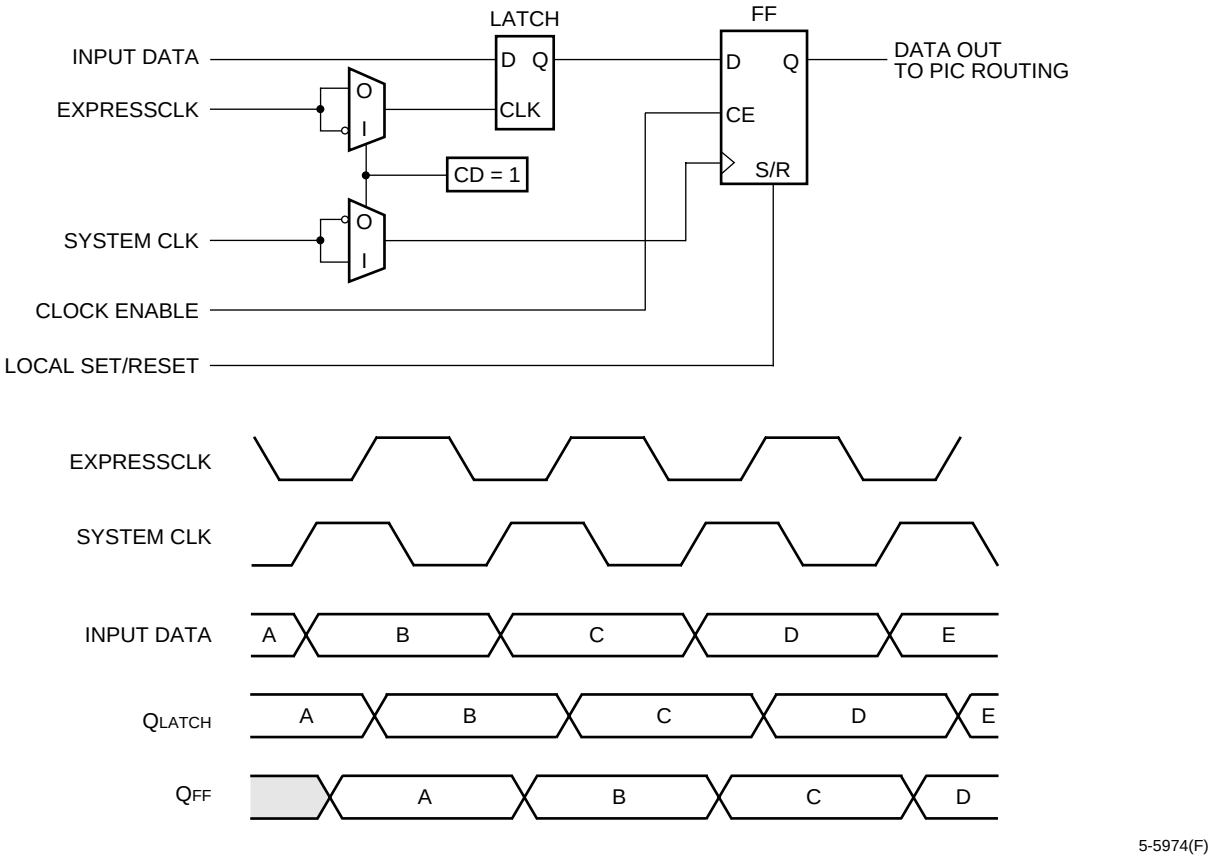


Figure 6. Fast Capture (Zero-Hold FF)

Methods to Assign Special Properties to PIC Resources (continued)

To instantiate this zero-hold circuit in *Verilog*:

```
// Verilog design for testing 3C/3T PIC: Zero Hold
module NOHOLD (SCLK,ECKR,DATAI,DATAO);
  input  SCLK,ECKR,DATAI;
  output DATAO;
  wire       DRHI, DRLO, ECKR_INT, DATAI_INT;
  // instantiate components
  VHI VH1 (DRHI);
  VLO VL1 (DRLO);
  CLKCNTLR CL1 (ECKR, DRLO, ECKR_INT);
  IBMS IB1 (DATAI,DATAI_INT);
  ILF2P3DX ILF1 (DATAI_INT, DRHI, ECKR_INT, SCLK, DRLO, DATAO);
endmodule
```

The ECLK pad location must be assigned in the preference file. There is one ECLK pin at the center of each side of the array, with a secondary ECLK pin available in the lower-left and upper-right corners. Also, clock enable and reset have been defaulted to simplify the example.

If an ECLK is not used for the clock, then a programmable delay input needs to be used with the PIO input FF. These are used by instantiating an input buffer whose name ends with an "s", such as IBMs. There are no clock restrictions with using this type of I/O buffer to obtain a zero hold, but the setup time to the FF will be slightly degraded vs. using a zero-hold flip-flop.

High-Speed 3-State Enable/Disable FF

In the Series 3 architechture, an extra FF in every PIO is available to register the 3-state enable/disable signal, thus allowing very fast bus turn around times.

High-Speed Clock-To-Output (T_{CO})

It has been noted already that by using a PIC register with an Express clock and a fast output buffer, extremely fast synchronous outputs can be generated. For example, in a OR3T80-7, the expected clock-to-output delay from an ECLK middle input pin to the PIC output pin (fast buffer, same side as ECLK, 50 pF load) is 4.5 ns (see data book, Timing Characteristics). Subsequent speed grades (-8, etc.) will naturally be faster. If external clock skew is minimized and external devices with fast setup times are used, this registered output can enable system speeds of +100 MHz.

CAUTION: If this output approach is used with wide buses, care must be taken to minimize ground bounce.

Fast Open-Drain Output—Shared Interrupt Line

The PIO output buffers can easily be configured for open-drain outputs, for applications such as a wired-OR interrupt signal to a microprocessor. The input signal to the 3-statable buffer can be directly tied to the buffer's 3-state control signal. Therefore, if the signal at the input to the buffer is a logical 0, the output is low, but if the input is a logical 1, the output is forced to high impedance. This open drain can easily be implemented in the source code and inferred by a synthesis tool as follows:

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- Open drain output, inferred

library IEEE;
use IEEE.std_logic_1164.all;

entity odrain is
   port (din:   in  std_logic;
         dout1: out std_logic;
         dout2: out std_logic);
end odrain;
```

Methods to Assign Special Properties to PIC Resources (continued)

```
architecture Behavioral of odrain is
  signal mid : std_logic;
begin
  mid   <= din;
dout1 <= '0' when (mid = '0') else 'Z';
  dout2 <= mid when (mid = '0') else 'Z';
end Behavioral;
```

In this example, dout2 will implement the fast open-drain output correctly, routing the mid signal through the PIC's OUT1 or OUT2 data path to the 3-state buffer's input and control pin in parallel. Dout1 will accomplish a similar function, but is not a true fast open-drain output. Dout1 will tie a VLO library element's output through OUT1 or OUT2 to the 3-state buffer's input and connect mid to the buffer's control pin via a separate route. Therefore, use the dout2 method.

An open-drain can also be instantiated using any of the OBZxxx output buffers. For example:

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- Open drain output, instantiated

library IEEE;
use IEEE.std_logic_1164.all;

entity odrain2 is
    port (din:  in  std_logic;
          dout: out std_logic);
end odrain2;

architecture Structural of odrain2 is
  component ibm
  port( i:  IN std_logic := 'X';
        o:  OUT std_logic);
  end component;

  component obz6
  port(   i:  IN std_logic := 'X';
          t:  IN std_logic := 'X';
          o:  OUT std_logic);
  end component;

  signal mid : std_logic;
begin
  ibuf: ibm  port map (din,mid);
  obuf: obz6 port map (mid,mid,dout);
end Structural;
```

Output Logic (AND, NAND, etc.)—OUT2 with ECLK or SCLK

The logic block inside each PIO allows the designer the ability to combine signals at the I/O to increase the speed of signals being sent off-chip. Either SCLK or ECLK can be gated with the OUT2 signal. Logic functions include AND, NAND, OR, NOR, XOR, and XNOR. Two examples of this are an address decode, where two signals are gated through an AND gate, and Parity generation where two signals are gated through an XOR gate. One of the two signals in these examples would come from one side of the AND/XOR tree and would connect to the OUT2 pin (available for each PIO), while the other signal would come from the other side of the AND/XOR tree and would connect to either the SCLK pin (available for each PIC) or the ECLK pin (available for each side of the device). This circuit could also be used to generate short external pulses through gated clocks.

Since this is an asynchronous circuit, great care must be taken to control the timing between signals to avoid unwanted glitches (see Note below). For example, the designer may want an internal flip-flop's output to be ANDed with its own SCLK and then drive this gated clock directly out to some external device.

Methods to Assign Special Properties to PIC Resources (continued)

To instantiate this PIO logic (AND function) in VHDL:

```
-- VHDL netlist for testing 3C/3T PIC architecture
-- PIO logic

library IEEE;
use IEEE.std_logic_1164.all;

entity piolog is
    port (in1:  in  std_logic;
          in2:  in  std_logic;
          and-out: out std_logic);
end piolog;

architecture mixed of piolog is
   component osand2
   port(a: IN std_logic := 'X';
        sclk: IN std_logic := 'X';
        z: OUT std_logic := 'X' );
end component;

begin

   gate: osand2 port map (in1,in2,and-out);
end mixed;
```

Synthesis tools do not infer PIO logic at this time.

Note: Gated clocks should be a last resort. It is a much better design practice to use nongated clocks and drive a clock enable pin to control data flow whenever possible. The same signal which drives the SCLK or ECLK pad of the FPGA should drive the external device's clock pin, and a synchronous signal from PIC output register should drive the external device's clock enable. This maintains a fully synchronous design which will be easier to debug and less prone to clock glitches.

Conclusion

The *ORCA* Series 3 PIC's numerous logic and routing features make it a flexible, high-speed interface between the external system and the Series 3's core logic array. The PICs are an integral part of the Series 3 architecture which allow these FPGAs to operate in a variety of modern high-speed digital applications.

January 2002
AP99-042FPGA

Lattice Semiconductor Corporation