The FLASH370i™ Family Of CPLDs and Designing with *Warp2*®

This application note covers the following topics: (1) a general discussion of complex programmable logic devices (CPLDs), (2) an overview of the FLASH370i™ family of CPLDs, and (3) using the *Warp2*® VHDL Compiler for the FLASH370i family.

Overview of CPLDs

CPLDs extend the concept of the PLD to a higher level of integration to improve system performance, use less board space, improve reliability, and reduce cost. Instead of making the PLD bigger with more input terms and product terms, a CPLD architecture is composed of multiple PLDs or logic blocks (LABs) connected together with a programmable interconnect matrix (PIM). Multiple Logic Array Blocks (LABs) provide comparable speed to a PLD because the basic propagation path is through one LAB and each LABs product term array is comparable to a PLD array. Multiple LABs provide the higher integration. The number of LABs in a CPLD is typically between 2 for the smaller CPLDs and 16 for the larger ones. In addition to LABs interconnected by the PIM, are the input/output macrocells and the dedicated input macrocells. *Figures 1* and *2* show the CPLD generic block diagram and the logic block diagram respectively.

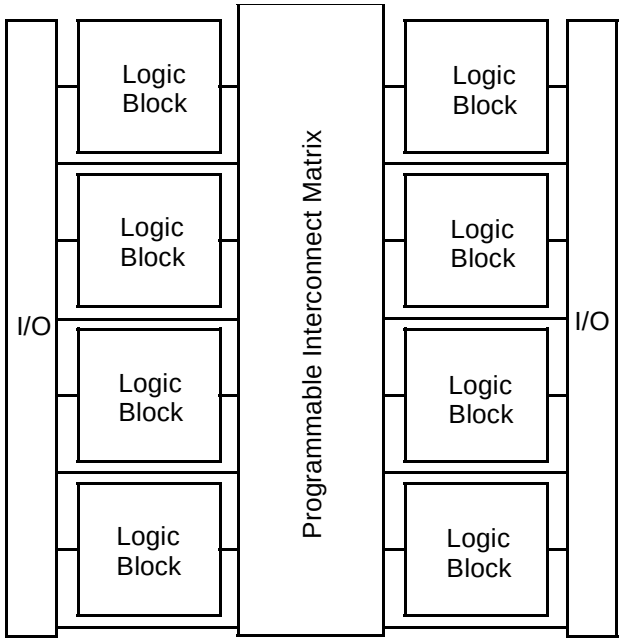

Figure 1. Generic Block Diagram

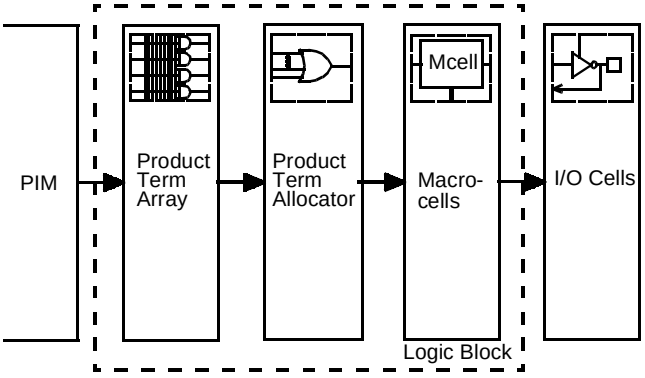

Figure 2. Logic Block Diagram

The architectural components of the LAB are: (1) the product term array, (2) the product term allocator, and (3) the macrocell. The product term array is the same in the CPLD as in the PLD except that the inputs into the array can now also come from the PIM. The product term allocator is a new concept in the CPLD where product terms are not fixed to a macrocell with its associated input/output pin but can be routed to different macrocells depending on where they are needed. The result is a more efficient allocation of product terms and higher integration. Implementation of the product term allocator varies across CPLD vendors which is more fully discussed in the section describing the features of the FLASH370i family.

The macrocell accepts the single output of the product term allocator which is the ORing of a variable number of product terms. In some macrocells this input feeds into a two input XOR gate with the other input potentially carrying the Q feedback. This configures the D flip flop to a T flip flop which can provide an improvement in capacity for certain designs such as counters. After the XOR gate, the macrocell is configurable as registered, combinatorial, and in some cases latched. There are two kinds of macrocells which are input/output dedicated and buried. Dedicated macrocells output to the input/output macrocell and also provide feedback into the product term array. Buried macrocells only provide feedback into the product term array.

The function of the PIM is to distribute the needed fraction of the total available resources, all outputs from the LAB and possibly also dedicated inputs and inputs/outputs, to the appropriate LAB. There are two common methods of PIM implementation: array based interconnect and mux based interconnect.

Figure 3 shows the data path of communication between two LABs using the array based interconnect. In the array based interconnect, each output of the LAB can potentially connect

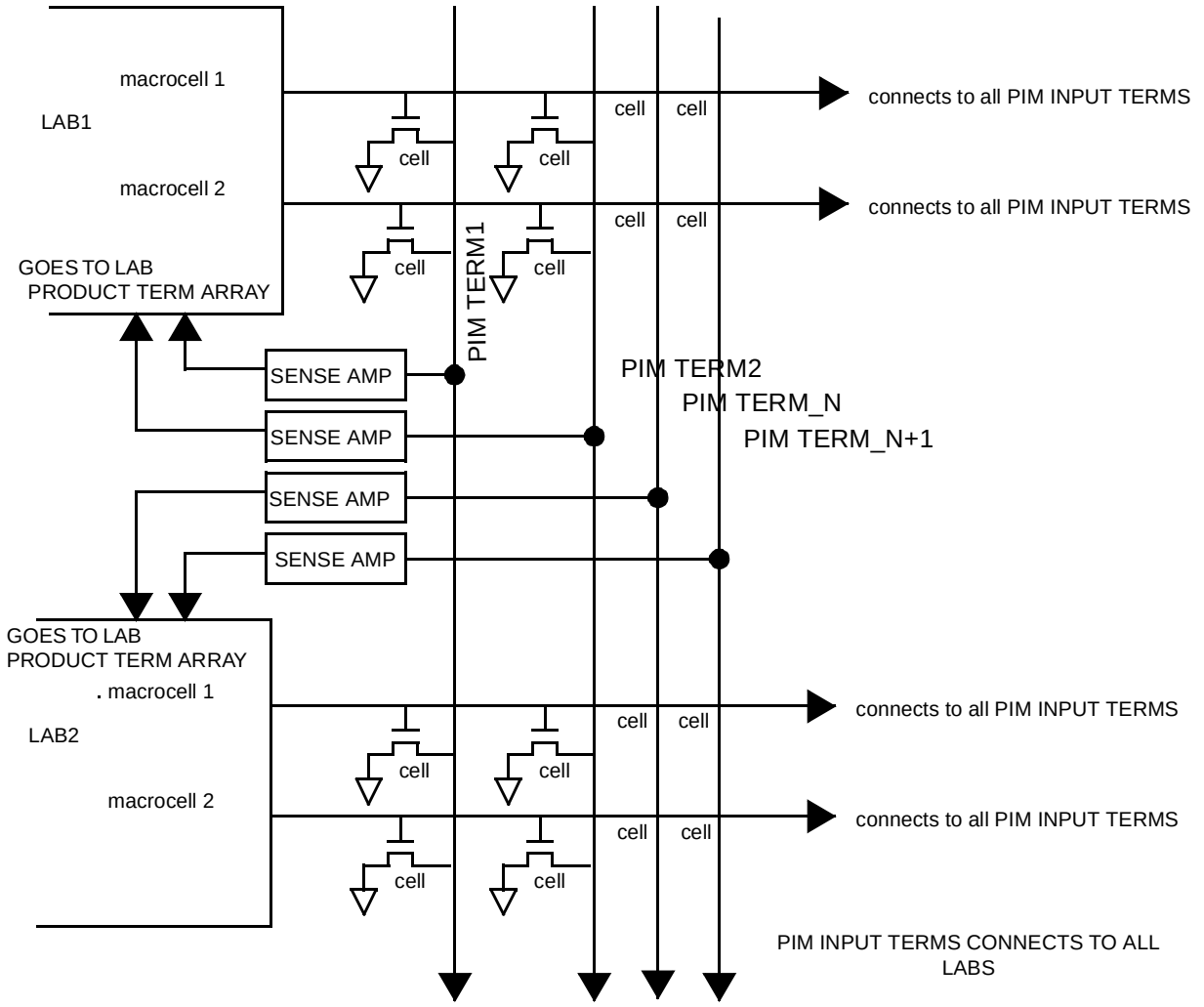

Figure 3. Array-Based Interconnect

to any number of PIM input terms through a memory element. Each PIM input term is assigned to a specific LAB and functions as an input term into the LABs product term array. In this example only four PIM input terms are shown two going to LAB1 and two going to LAB2. There is a sense amp per input term to detect the logic level, buffer the signal, and drive it into the LAB. The true and complement of the PIM signal feed into the product term array (not shown in the figure). Since every LAB output can connect to any PIM input, the interconnect is considered 100 percent routable. It never limits the ability of the device to fit logic. A macrocell output can connect to one or multiple PIM input terms. The major drawback from using a memory element as an interconnect is the slower propagation delay than the muxed based interconnect.

Figure 4 shows the data path of communication between two LABs using the muxed based interconnect. In the muxed based interconnect a mux chooses one of a number of potential PIM input terms into the LAB. The PIM input terms differ from the array based interconnect in that they are output from a 1 of n (where “n” is the number of inputs of the mux) mux instead of the output of a wired nor memory array. The inputs into the muxes are all the outputs of the LABs as well as dedicated inputs and input/output pins. *Figure 3* shows two PIM input terms output from two 4-to-1 muxes. In this example, macrocell 2 from LAB1 and macrocell 2 from LAB2 both show 2 chances to route into the muxes with other inputs having only 1 chance. The wider the mux (the number of inputs into the mux) the more likely all desired inputs into each LAB will be successfully routed and the more chances each signal gets to route into a LAB. The disadvantage of larger muxes is a larger slower propagation delay through the PIM and increased die size. Implementations of mux-based interconnect vary in the size of the mux.

Features of the FLASH370i CPLDs

The FLASH370i family of CPLDs offers densities from 2 to 8 LABs. *Figure 5* shows the block diagram of the CY7C374i/5i with 8 LABs. The even numbers of the family (372i,374i) bury half of the macrocells for maximum integration with the same pinout as the (371i,373i,375i) respectively. *Table 1* shows the family members offered.

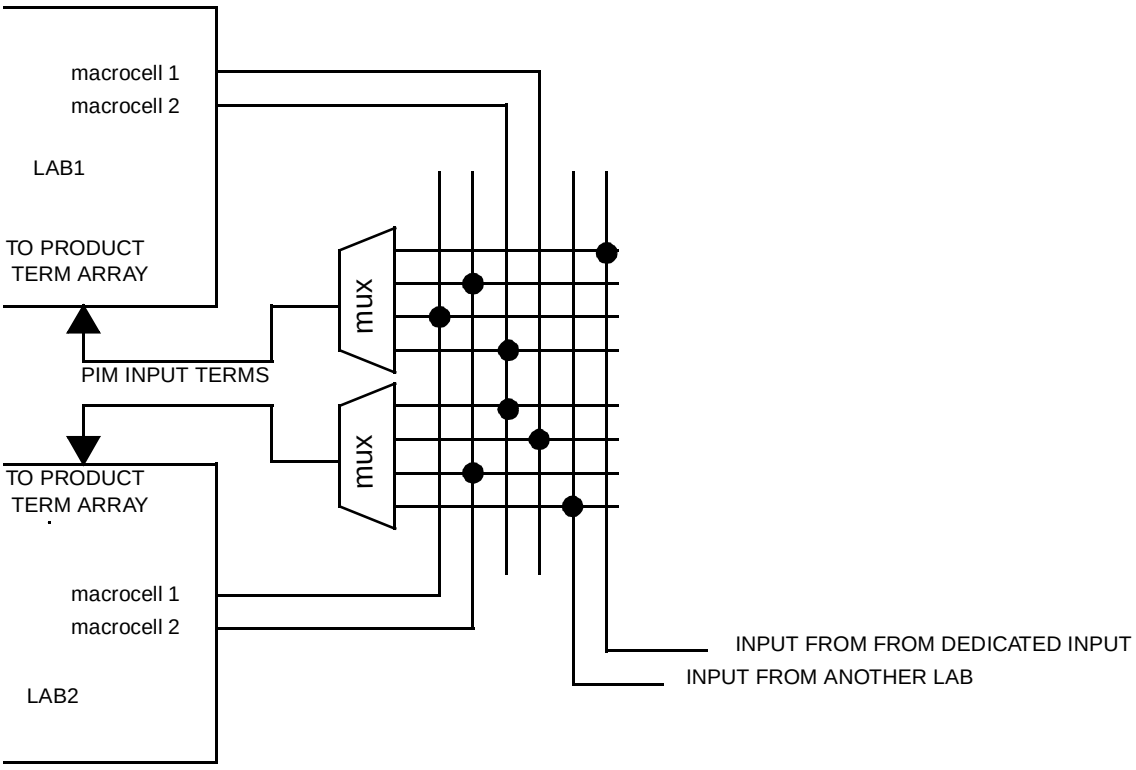

Figure 4. Mux-Based Interconnect

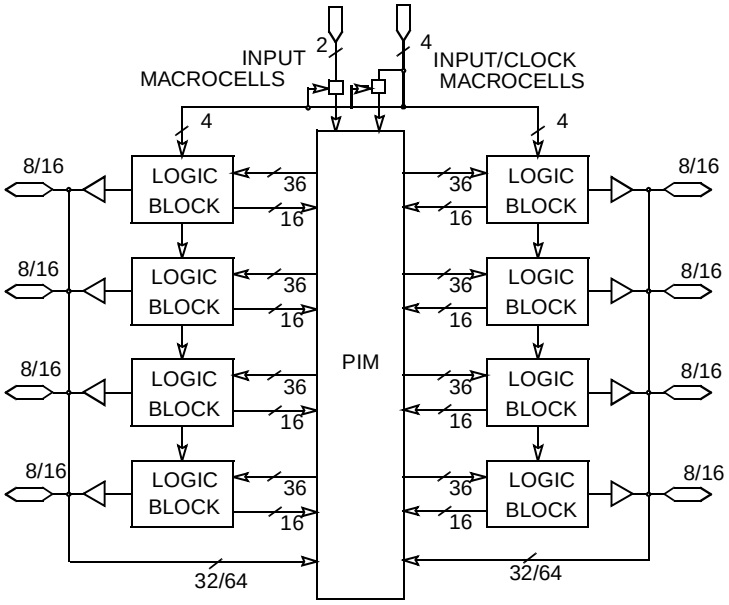

Figure 5. CY7C374i/5i Block Diagram

Table 1. FLASH370i Family Members.

Feature	CY7C371i	CY7C372i/3i	CY7C374i/5i
Macrocells	32	64	128
Dedicated Inputs	6	6	6
I/O pins	32	32/64	64/128
Dedicated Inputs Usable as Clocks	2	2/4	4/4
Speed (t_{PD})	8.5 ns	10 ns	12 ns
Primary Packages	44–PLCC	44/84–PLCC 100–TQFP	84–PLCC 100/160–TQFP

Figures 6 and 7 show the product term array, product term allocator, macrocells, and input/output macrocells for the FLASH370i family. Each LAB features 36 inputs, which can adequately handle 32-bit operations plus control signals with one pass through the LAB. The product term array features the true and complement polarities of each PIM output signal for a total of 72 inputs. 80 standard product terms are provided to the product term allocator which allocates from 0 to 16 product terms to each of the 16 macrocells. Additionally, 6 special product terms are also generated in the product term array. They are an asynchronous preset, asynchronous reset, and two groups of 2 bank output enable product terms.

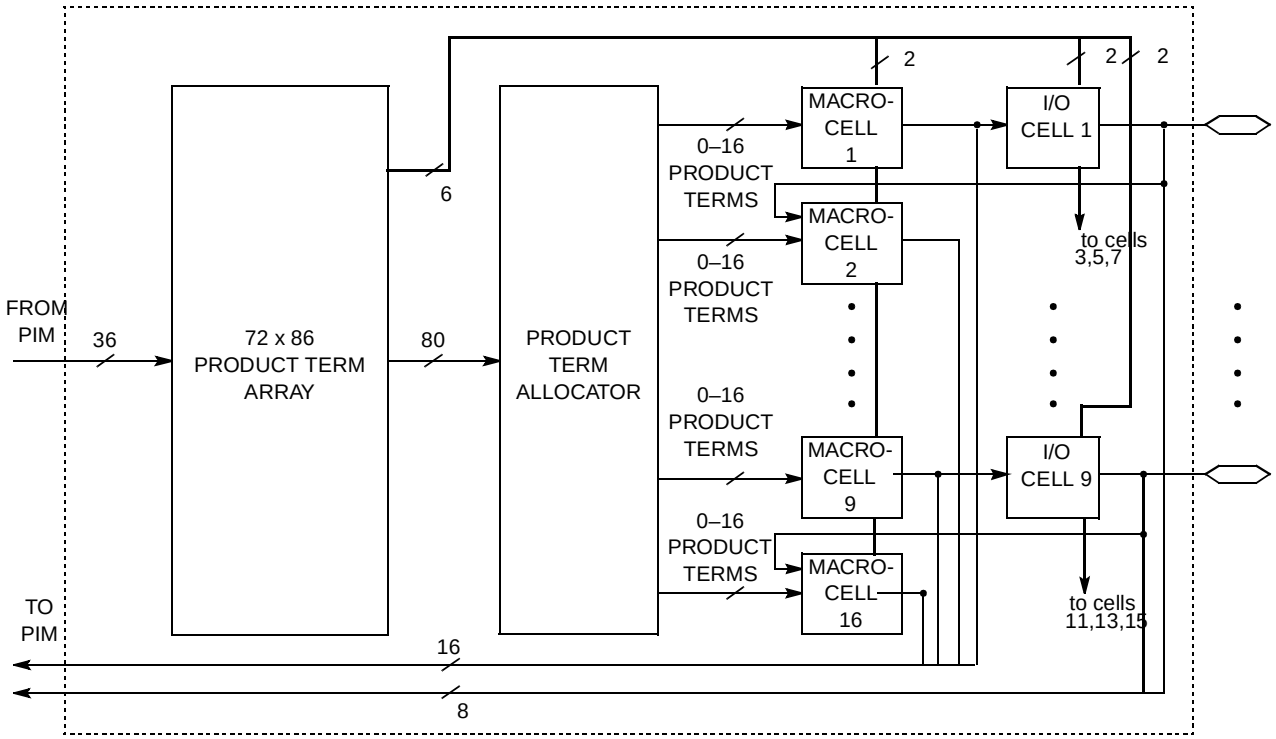

Figure 6. Logic Block for CY7C372i and CY7C374i (Register Intensive)

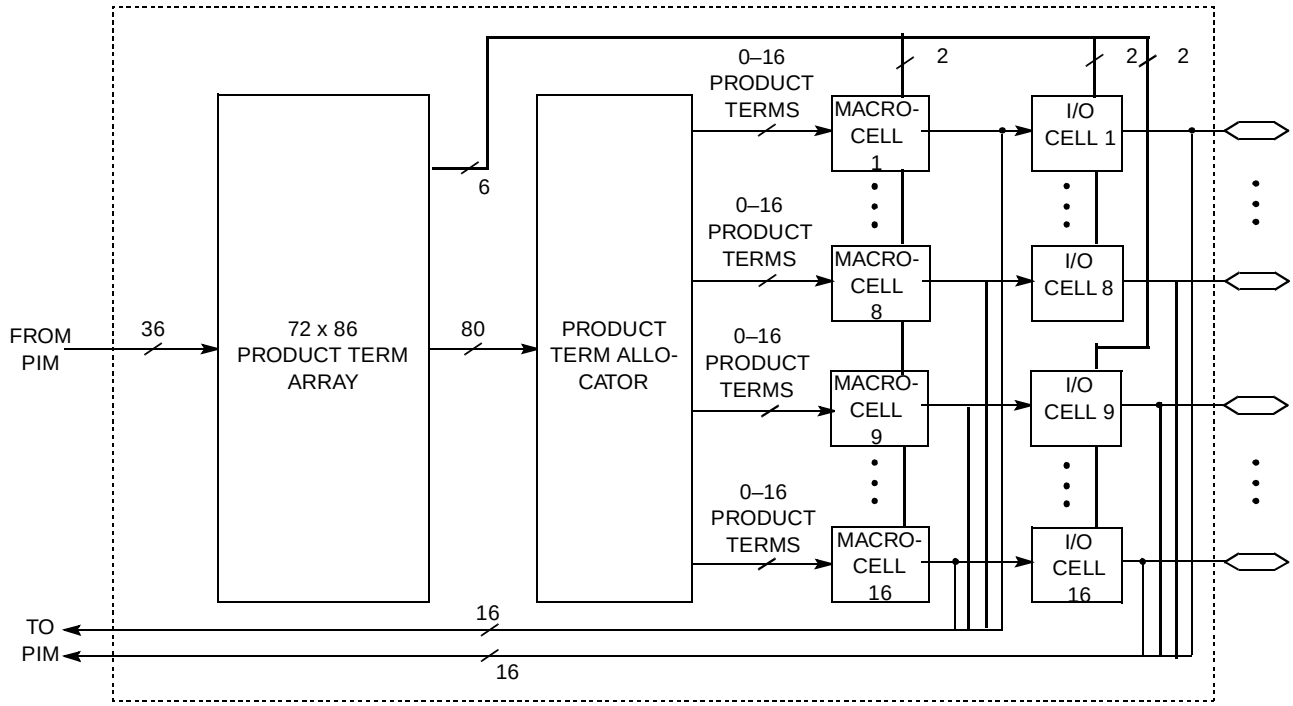

Figure 7. Logic Block for CY7C371, CY7C373, CY7C375, and CY7C377 (I/O Intensive)

The output macrocell (*Figure 8*) provides a selection of four output controlling options: (1) control from one output enable, (2) control from a second output enable, (3) permanently enabled, or (4) permanently disabled. Each LAB contains 4 output enable product terms, 2 for the upper 8 macrocells and 2 for the lower 8 macrocells.

The state macrocell (*Figure 8*) contains options to register, latch, or send data through combinatorially. For the input/output macrocell there is an additional output polarity mux to improve capacity before the signal goes to the input/output macrocell. For buried macrocells there is an additional mux which can configure the state register as an input register. If the buried macrocell is configured as an input, zero product terms will be allocated from the array. In *Figure 8* architecture bit C7 can choose the feedback from the input/output pin as the input into the register instead of from the product term array.

There is one asynchronous preset and reset product term for each LAB. There are polarity muxes for the clocks, preset and reset. Each macrocell can choose among two clocking options for the CY7C371i/372i and four clocking options for the CY7C373i/374i/375i. All macrocells in a LAB receive the same polarity of the clock, set and reset. Polarities are configurable per LAB. *Figure 8* shows the input/output macrocell and input/output plus buried macrocell.

Figures 9 and *10* show the input/clock and input macrocells. The input macrocell provides the flexibility to let the input enter combinatorially, latched, single registered, or double registered (for maximum metastability performance). For the CY7C371i/372i there are two input/clocks pins and four input pins. For the CY7C373i/374i/375i there are four input/clock pins and two input pins. For added flexibility, each clock can be configurable for either positive or negative polarity.

In order to fully understand the operation of the FLASH370i product term allocator, two important aspects of product term allocator design need to be introduced: product term steering and product term sharing. Steering refers to the assignment of a product term resource to a macrocell. In the traditional PLD there is no steering flexibility. Each macrocell has assigned product terms that can only be used by that macrocell. In many designs each macrocell requires a different number of product terms putting an emphasis on the ability to allocate product terms individually on an as needed basis. Product term sharing refers to a product term being used by multiple macrocells. The logic equations for different macrocells sometimes contain the same minterm. Instead of generating this same minterm multiple times, it is generated on only one product term and shared across macrocells, thereby improving capacity.

Figure 11 is a conceptual representation of the FLASH370i product term allocator. The product term allocator functions like a segmented OR array by ORing from 0 to 16 product terms for each macrocell. Product terms can be steered and shared on an individual basis. This architecture has several advantages over other implementations that steer product terms away from one macrocell to serve another.

Figure 12 is a conceptual representation of the MACH™ product term allocator. It shows no ability to share product terms across macrocells. Each cluster of four product terms can route to only one macrocell. The product terms are routed in groups of four which is a much higher granularity of product term allocation and not as efficient.

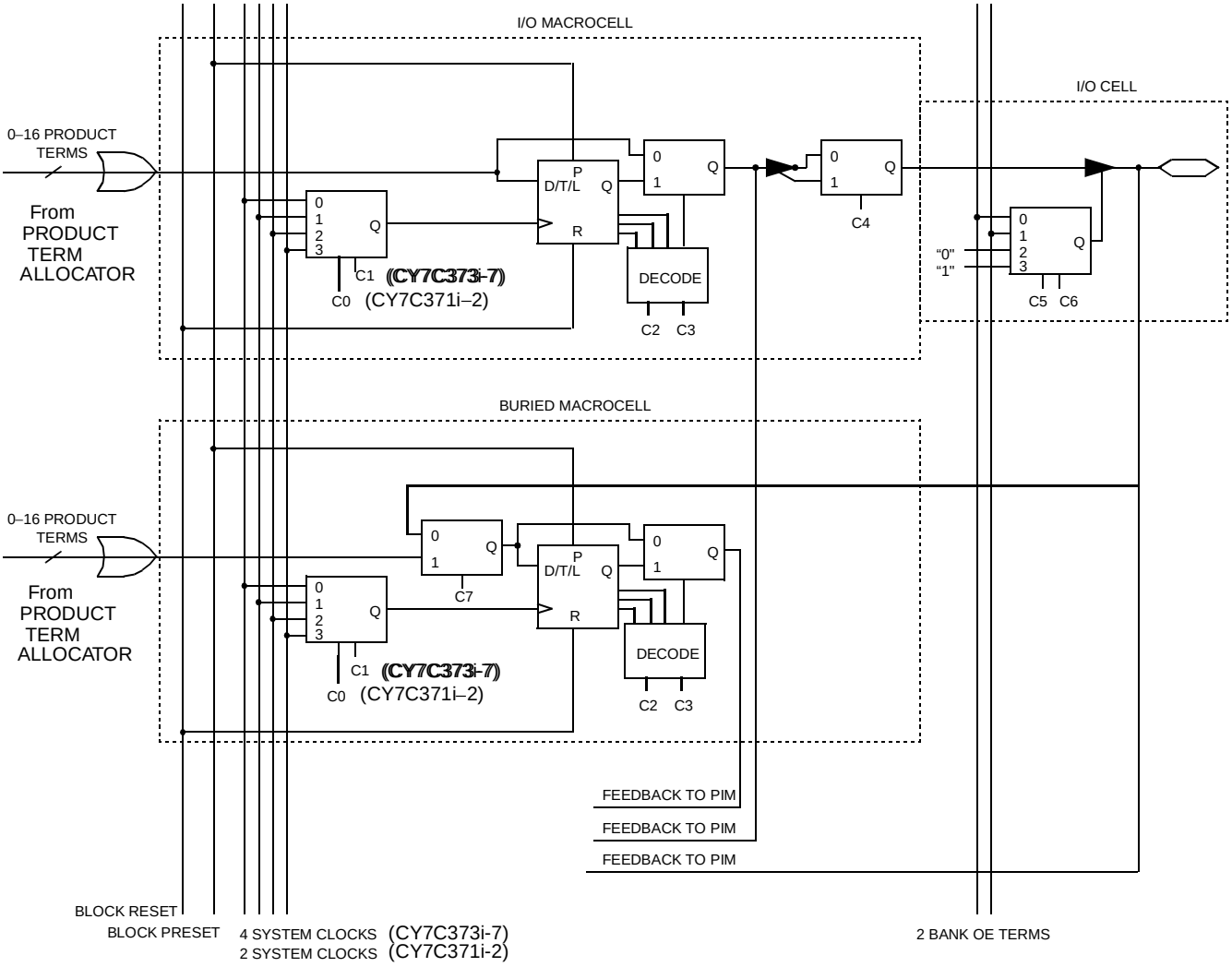

Figure 8. I/O and Buried Macrocells

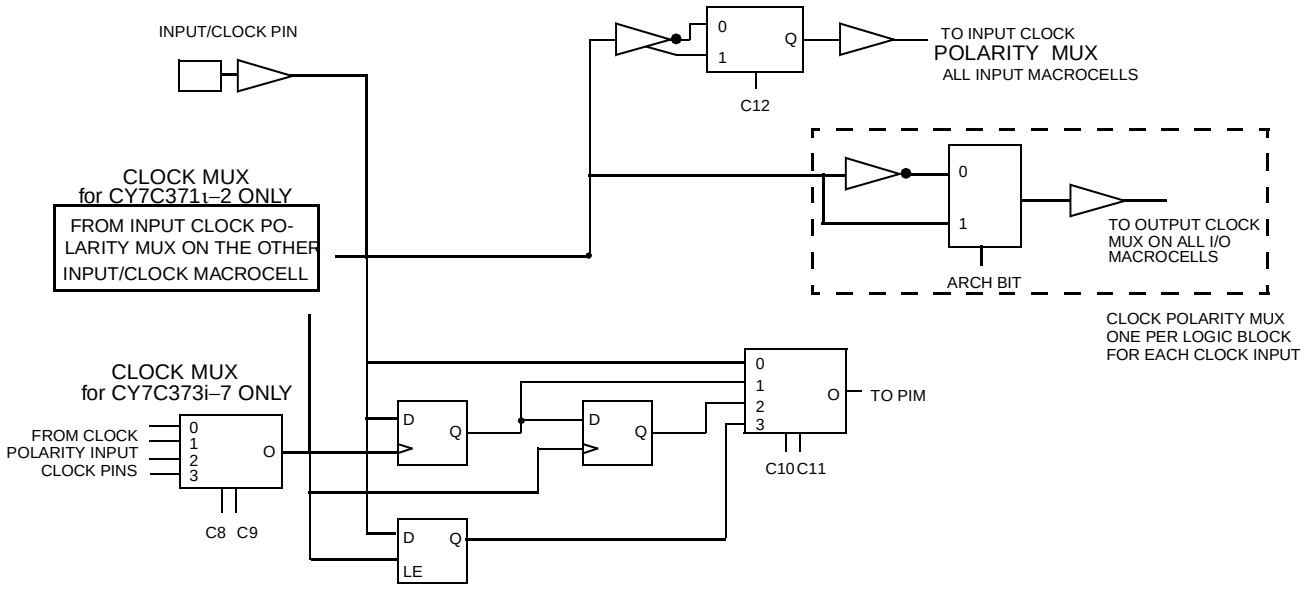

Figure 9. Input/Clock Pins

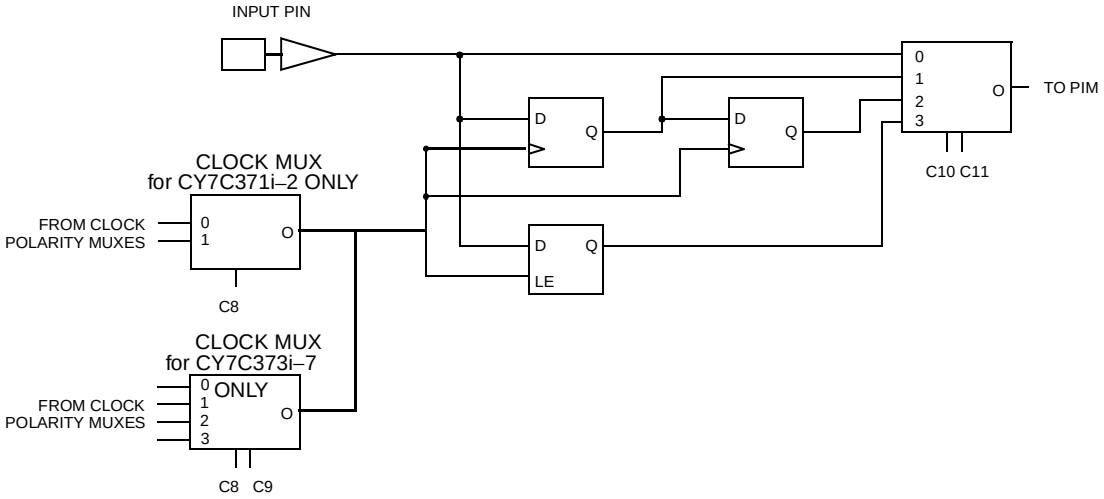

Figure 10. Input Pins

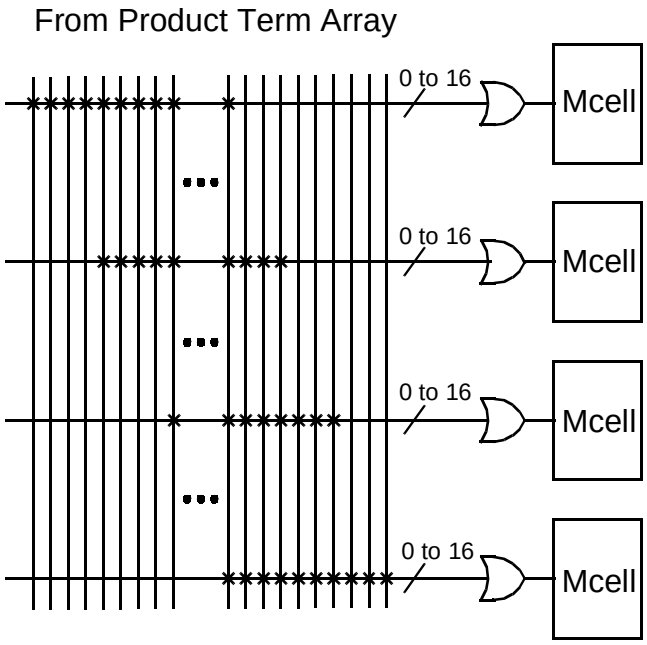

Figure 11. FLASH370i Product Term Allocator Representation

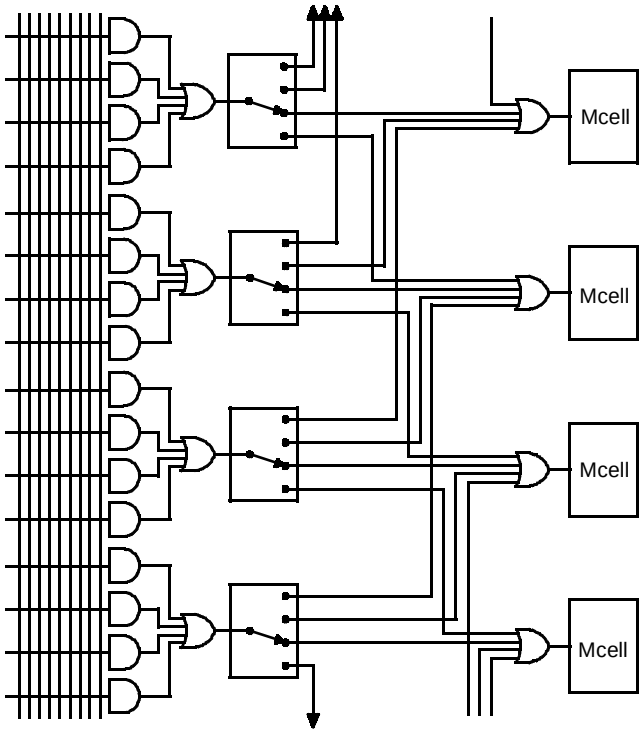

Figure 12. MACH Product Term Allocator Representation

To demonstrate this inefficiency, consider a macrocell that needs five product terms to implement its logic. Two product term clusters with a total of eight available product terms are needed. This wastes the resources of three product terms from the borrowed cluster since these product terms can not be rerouted to another macrocell.

The MAX7000™ product term allocator representation (*Figure 13*) shows the use of expander terms

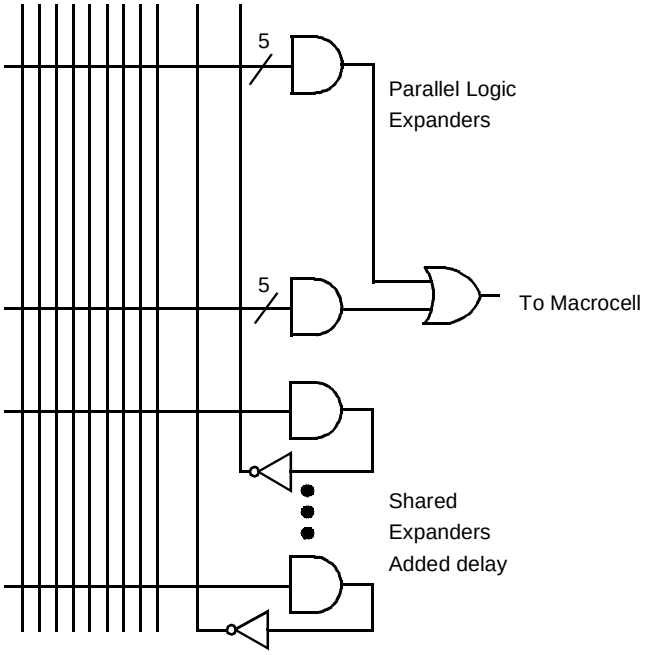

Figure 13. MAX Product Term Allocator Representation

allow two passes through the array which can produce very high capacity. These expanders are also shared among all product terms in the LAB. The problem with using the expanders is in the additional propagation delay of two passes through the array. This complicates the timing model and links the performance of the device to the use of the expander product terms. As with the MACH product term allocator, the MAX7000 allocator also has five product term clusters. It therefore suffers from the same problem of product term wasting when more than one cluster is routed to a macrocell.

The FLASH370i product term allocator provides the most effective method of steering and sharing product terms. The propagation of signals through the product term allocator is independent of the number of product terms allocated to each macrocell. Additionally the flexibility of this product term allocator, with the PIM, enables a change in the design without a modification to the external pinout of the device. There is no need for input and output switch matrices, which add extra delay and degrade performance.

The timing model of the FLASH370i family is far simpler than for other CPLD solutions for two reasons. First, all input signals into the LAB pass through the PIM. This includes all input/outputs, feedbacks from macrocell outputs, and dedicated inputs. Secondly, the propagation time through the product term allocator is independent of the number of product terms allocated to a macrocell. As a result, there are no expander delays, no dedicated versus input/output pin delays, no penalties for using up to 16 product terms, or no delay penalties for steering and or sharing product terms. The FLASH370i family of products provides timing as predictable as PLDs like the 22V10.

The PIM in the FLASH370i was designed to approach the 100 percent routability of the array based interconnect but not made so wide that performance and die size suffered.

Using *Warp*™ to Design with the FLASH370i

Development software is extremely important for ease of use and efficiency of resource allocation when designing with CPLDs. Cypress offers two software packages that will fully support the FLASH370i family of products as well as all other PLDs and state machine PROMs. *Warp2* provides full VHDL language support which is becoming the industry standard for describing hardware design. A functional simulator is also provided. *Warp3*® additionally includes schematic capture and exact timing simulation capability.

The simplified timing model of the FLASH370i often makes exact timing simulation unnecessary because performance can be predicted directly from the datasheet. Therefore the functional simulator of *Warp2* may be a cost effective design solution. With *Warp* no manual intervention for fitting the designs into the devices are necessary. In addition to *Warp*, customers also have third party support from a variety of vendors.

Warp products take in VHDL designs and automatically fit them into the chosen device. The following section explains how to exploit the special features of the FLASH370i with VHDL. A thorough treatment of VHDL constructs is found in the *Warp2* Reference Manual. Topics covered here are: (1) using the single/double registered options for the dedicated inputs, and registering signals from the IO pins, (2) using the clock polarity mux feature, (3) describing registered versus latched versus combinatorial outputs, (4) using the output enable feature, (5) using the asynchronous preset/reset feature, and (6) Using the buried registers as for the (372/4/6).

To register the dedicated inputs one or two signals must be defined to represent the additional nodes for one and two registers respectively. Appendix A demonstrates how to use single and double registered inputs for a 4 bit loadable counter. In *proc2*, RESET1 and RESET2 are the outputs of the first and second registers. It requires 2 passes through *proc2* to activate RESET2. Signal RESET2 is then used in *proc1* to perform the reset. *Proc2* additionally registers the data to be loaded with the statement `regin <=temp.dat`. The signal REGIN is then used in process *Proc1* to load the counter with the statement `temp.cnt <= regin`. If the same clock is used for the inputs as for the state registers, then the statements in process *proc2* could be incorporated into *proc1* and only one process is needed. The assignment of the entity output pins is handled by the instantiation of the *bufoe* component (called in the statement `use work.rtlpkg.all`), which takes the signal TEMP.CNT as input and transfers it to the output (in this case called COUNT) when the output enable control (called OUTEN) is HIGH. Registering the inputs from the input/output pins is better suited for the 372/374/376 members of the family since the signal does not need to go through the PIM and logic block.

Clocking on the falling instead of the rising edge of the clock is simply done by changing the statement `wait until (clk = '1')` to `wait until (clk = '0')`. Events occurring on the rising and falling edge of a clock can be incorporated into the same design by defining a separate process for the event, provided that sufficient logic blocks are available.

VHDL describing combinatorial and registered outputs is identical to other part implementations as with the FLASH370i. The registered equations must be inserted inside a process and after a `wait until clock=` statement.

Appendix B shows an example of how to implement the combinatorial macrocell option with maximum usage of output enable flexibility for the CY7C371. A total of eight different input signals control the output enable functionality. The entire function is handled by the *bufoe* component where the input into the buffer is the external input pin. No signals are necessary.

The latch option is unique to the FLASH370i family. Appendix C shows an example of how to latch a signal using the IF-THEN-ELSE construct. In this example the signal is latched when the clock is HIGH by setting the signal value to itself with the statements `signala <= signala` and `signalb <= signalb`. When the clock is LOW the path is combinatorial and the signal value gets the input. This is handled in the code `if clk='0' then signala <= inputa; signalb <= inputb`. Two signals are defined, SIGNALA and SIGNALB, to latch the data when the clock is in the right polarity (in this case HIGH).

Appendix D shows the full registered configuration. As in Appendix C, the signals SIGNALA and SIGNALB are defined and the function of the register is defined within a process. On the rising edge of the clock, SIGNALA gets INPUTA and SIGNALB gets INPUTB.

Appendix E uses latches for the output enable control. Signals need to be generated from the array and are passed as the output enable parameter into the *triout* component. This function behaves similarly to the *bufoe* but does not include the feedback parameter.

Appendix F shows how to use the buried registers to implement the least significant bits in a counter. A bit vector signal is defined to represent all the register states. Those states that are needed as outputs are assigned to the entity output pins outside of the process with the statement `count (0 to 11) <= fullcnt (4 to 15)`. If output enable control is desired then this last statement is omitted and the signal to output assignment is handled with the *bufoe* component.

Appendix G is the same as Appendix F except that the registers are reset asynchronously. The format of the process is much different from Appendix F but functions exactly the same except for the asynchronous instead of synchronous reset. The process uses a “sensitivity list” that includes all the parameters that will activate the process. The synchronous part of the process is initiated by the statement `clk'event and clk='1'` instead of `wait until clk='1'`. The asynchronous preset/reset is similar to other Cypress PLDs except for the additional polarity mux feature that enables active HIGH or LOW. To specify clock polarity, the VHDL construct for active HIGH is `if reset = '1' then` and for active LOW is `if reset = '0' then`.

Appendix A. inregcnt

```
-- The bufoe port map parameters are:
-- bufoe port map(signal going to the input of the tristateable buffer,
--     tristate control signal,
--     the output signal that is the entity output pin,
--     the feedback signal from the entity input/output pin)
-- In this example the last entry is “open” meaning no feedback.

USE work.bv_math.all;          -- necessary for inc_bv();
USE work.rtlpkg.all;         -- necessary for bufoe

ENTITY inregcnt IS
       PORT   (clk, clkin, reset, load, outen: IN bit;
         count: INOUT x01z_VECTOR(0 TO 3));
END inregcnt;

ARCHITECTURE behavior OF inregcnt IS
TYPE bufRec IS                 -- record for bufoe
       RECORD                  -- inputs and feedback
            cnt: bit_vector(0 TO 3);
            dat: bit_vector(0 TO 3);
END RECORD;
SIGNAL temp: bufRec;
SIGNAL regin: bit_vector(0 to 3);-- for registering input loaded data
SIGNAL reset1, reset2:bit;      -- for registering the reset input
CONSTANT counterSize: integer := 3;
BEGIN
g1:    FOR i IN 0 TO counterSize GENERATE
            bx: bufoe PORT MAP(temp.cnt(i), outen, count(i), temp.dat(i));
       END GENERATE;
proc1: PROCESS
       BEGIN
            WAIT UNTIL (clk = ’1’);
            IF reset2 = ’1’ THEN -- uses the double registered signal
                 temp.cnt <= “0000";
            ELSIF load = ’1’ THEN
                 temp.cnt <= regin; -- uses the single registered signal
            ELSE
                 temp.cnt <= inc_bv(temp.cnt);  -- increment bit vector
            END IF;
       END PROCESS;
-- Proc2 single registers the load operation and double registers the reset -- oper-
ation.  Note the two clkin’s are needed for the double register.
proc2: PROCESS
       BEGIN
         WAIT UNTIL (clkin = ’1’);
          regin <= temp.dat; --single register for data load
          reset1 <= reset; --single register the reset signal
          reset2 <= reset1;--double register the reset signal
       END PROCESS;
END behavior;
```

Appendix B. usecomb

```
--uses the full functionality of the oe features of the 371.
--macrocell is in combinatorial mode

USE work.rtlpkg.all;

ENTITY usecomb IS
       PORT (outen1, outen2, outen3, outen4, outen5, outen6, outen7,
            outen8; IN bit; inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END usecomb;

ARCHITECTURE behavior OF usecomb IS
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(inputa(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(inputa(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(inputa(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(inputa(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(inputb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(inputb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(inputb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(inputb(i), outen8, outb(i+6), open);
       END GENERATE;
END behavior;
```

Appendix C. uselatch

```
--uses the full functionality of the oe features of the 371.
--macrocell in latched mode

USE work.rtlpkg.all;

ENTITY uselatch IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
       outen8: IN bit;
            inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END uselatch;

ARCHITECTURE behavior OF uselatch IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;--the clk input is an active low latch enable
--the if then construct must be within a process.
PROCESS
  BEGIN
     IF clk='0' then
       signala <= inputa;
       signalb <= inputb;
     ELSE
       signala <= signala;
       signalb <= signalb;
     END IF;
  END PROCESS;
END behavior;
```

Appendix D. usereg

```
--macrocell in registered mode

ENTITY usereg IS
       PORT   (clk, outen1, outen2, outen3, outen4, outen5, outen6, outen7,
       outen8: IN bit; inputa, inputb: IN bit_vector(0 to 1);
            outa,outb: INOUT x01z_vector(0 to 7));
END usereg;

ARCHITECTURE behavior OF usereg IS
SIGNAL signala, signalb: bit_vector(0 to 1);
BEGIN
g1:    FOR i IN 0 TO 1 GENERATE
            bx1: bufoe PORT MAP(signala(i), outen1, outa(i), open);
            bx2: bufoe PORT MAP(signala(i), outen2, outa(i+2), open);
            bx3: bufoe PORT MAP(signala(i), outen3, outa(i+4), open);
            bx4: bufoe PORT MAP(signala(i), outen4, outa(i+6), open);
            bx5: bufoe PORT MAP(signalb(i), outen5, outb(i), open);
            bx6: bufoe PORT MAP(signalb(i), outen6, outb(i+2), open);
            bx7: bufoe PORT MAP(signalb(i), outen7, outb(i+4), open);
            bx8: bufoe PORT MAP(signalb(i), outen8, outb(i+6), open);
       END GENERATE;    --the clk input is a rising edge triggered clock for
                        --the register
--the wait until construct must be within a process.
PROCESS
  BEGIN
    WAIT UNTIL clk='1';
    signala <= inputa;
    signalb <= inputb;
  END PROCESS;
END behavior;
```

Appendix E. uselatch2

```
--This file shows the use of the triout component to perform the
--output enable function.

--COMPONENT triout
-- port (
--   x: IN bit; -- input to buffer
--   oe: IN bit; -- output enable
--   y: OUT bit); -- output
--END component

--The oe control is a function of the dedicated inputs and is latch
--controlled.

USE work.rtlpkg.all;         --to instantiate triout component

ENTITY uselatch2 IS
  PORT (clk1, clk2, in_oe1, in_oe2: IN bit;
        inputa, inputb: IN bit_vector(0 to 1);
         outa,outb: INOUT x01z_vector(0 to 7));
END uselatch2;

ARCHITECTURE behavior OF uselatch2 IS
SIGNAL signala, signalb: bit_vector(0 to 1);
SIGNAL sig_en1, sig_en2, sig_en3, sig_en4: bit;
BEGIN
g1:  FOR i IN 0 TO 1 GENERATE
       bx1: triout PORT MAP(signala(i), sig_en1, outa(i));
       bx2: triout PORT MAP(signala(i), sig_en2, outa(i+2));
       bx3: triout PORT MAP(signala(i), sig_en3, outa(i+4));
       bx4: triout PORT MAP(signala(i), sig_en4, outa(i+6));
       bx5: triout PORT MAP(signalb(i), sig_en1, outa(i));
       bx6: triout PORT MAP(signalb(i), sig_en2, outa(i+2));
       bx7: triout PORT MAP(signalb(i), sig_en3, outa(i+4));
       bx8: triout PORT MAP(signalb(i), sig_en4, outa(i+6));
     END GENERATE;

--The clock latches the data when high and is combinatorial when low
oecontrol: PROCESS
            BEGIN
              IF clk1= '0' then
                 sig_en1 <= not(in_oe2) and not(in_oe1);
                 sig_en2 <= not(in_oe2) and in_oe1;
                 sig_en3 <= in_oe2 and not(in_oe1);
                 sig_en4 <= in_oe2 and in_oe1;
              ELSE
                 sig_en1 <= sig_en1;
                 sig_en2 <= sig_en2;
                 sig_en3 <= sig_en3;
                 sig_en4 <= sig_en4;
              END IF;
            END PROCESS;

latch: PROCESS
```

Appendix E. uselatch2 (continued)

```
            BEGIN
              IF clk2= '0' then
                signala <= inputa;
                signalb <= inputb;
              ELSE
                signala <= signala;
                signalb <= signalb;
              END IF;
            END PROCESS;
END behavior;
```

Appendix F. buriedreg

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.

USE work.bv_math.all;               -- necessary for inc_bv();

ENTITY buriedreg IS
       PORT   (clk, reset: IN BIT;
            count: INOUT bit_vector(0 TO 11));
END buriedreg;

ARCHITECTURE behavior OF buriedreg IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
PROCESS
     BEGIN
       WAIT UNTIL (clk = '1');
            IF reset = '1' THEN       -- synchronous reset
                 FOR i IN 0 TO 15 LOOP
                      fullcnt(i) <= '0';
                 END LOOP;
            ELSE
                 fullcnt <= inc_bv(fullcnt);
            END IF;
     END PROCESS;
count(0 to 11) <= fullcnt(4 to 15);
END behavior;
```

Appendix G. buriedreg2

```
-- The purpose of this example is to show how to use the
-- buried registers to create a 16 bit counter.  The 12
-- most significant bits are assigned to i/o registers
-- and the 4 least significant bits go to the buried registers.
-- This example also demonstrates how to do an asynchronous reset.

USE work.bv_math.all;              -- necessary for inc_bv();

ENTITY buriedreg2 IS
  PORT   (clk, reset: IN BIT;
     count: inout bit_vector(0 TO 11));
END buriedreg2;

ARCHITECTURE behavior OF buriedreg2 IS
SIGNAL fullcnt : bit_vector(0 to 15);
BEGIN
   PROCESS(clk,reset)--sensitivity list
         BEGIN
           IF reset = '1' THEN
             fullcnt <= x"0000";-- asychronous reset, the x stands for hex
           ELSIF (clk'event and clk = '1') then
             fullcnt <= inc_bv(fullcnt);-- synchronous count
           END IF;
         END process;
   count(0 to 11) <= fullcnt(4 to 15);     -- assignssignals to entityoutputs
  -- and defines buried registers
END behavior;
```

MAX7000 is a trademark of Altera Corporation.
MACH is a trademark of Advanced Micro Devices, Inc.
Warp2 and *Warp3* are registered trademarks of Cypress Semiconductor Corporation.
Warp and FLASH370i are a trademarks of Cypress Semiconductor Corporation.